FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

How to Efficiently Collaborate in Model Versioning

A Guideline to Reduce and Resolve Conflicts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Magistra der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Melanie Kapellner
Matrikelnummer 0106958

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung

Betreuung: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Dipl.-Ing. Dr. Martina Seidl
Mag. Petra Brosch

Wien, 30. August 2010 _____ _____

(Unterschrift Verfasserin) (Unterschrift Betreuung)

Danksagung

Besonderer Dank gilt meiner Mutter, die mich in jeder Lebenslage unterstützt, und einer besonderen Freundin, die mich gerade in dieser Lebenslage sehr unterstützt hat, und ohne die diese Diplomarbeit nicht möglich gewesen wäre.

Vielen Dank auch an meine Betreuerinnen für die intensive und ausdauernde Betreuung, außerdem an meinen Vater, Marion, Matthias, Jody, und an alle, die in irgendeiner Weise zur Entstehung dieser Arbeit beigetragen haben.

Abstract

The advances in software development lead to ever increasing complexity of software systems. Therewith rises the demand for technologies and techniques to shield the developers from that system complexity. For decades researchers and practioners have created abstractions to that end. Early programming languages shielded the programmers from the complexity of programming with machine code. Continuously, new programming languages and environments emerged and still emerge. At the moment, model-driven development, the next abstraction level, is on the rise, leveraging models to central artefacts of software development.

Nowadays, software is developed in teams. Models provide the advantage to be additionally used as an inter-expert language, hence facilitating communication. Consequently, a model is often edited by multiple persons. Version control provides collaborative support on a technical level by enabling concurrent work. In version control, mechanisms are refined to improve software development in teams. Current versioning tools, however, mainly provide support for code-centric software engineering. Versioning of models still faces major challenges. Models may not be as easily divided into parts and partitioned for several developers to work on as source code is. Due to parallel development, the versioning of models is especially prone to conflicts. The resolution of conflict situations is a precarious and preferably collaborative process.

Progress in model-driven development research seems to be restricted to the technical level, while social aspects like human behaviour and cooperation are often neglected. This is surprising, since software developers spend more than 60% of their time with collaborative activities, including meetings and other communication, coordination of tasks, and collaboratively executing these tasks. Well working collaboration in a team leads to an overall better outcome. It is therefore important to actively shape collaboration to gain the huge potential benefits achieved through efficient teamwork.

This thesis provides a general survey on possibilities to improve the model-driven development process in regard to collaborative aspects. We therefore analyse means and methods of collaboration in theory and practice, both in software engineering and various other fields. We highlight the importance of attaching a greater value to collaboration in research and in practice. Since collaboration is especially finical in conflict situations, we provide a specific example for enhanced collaboration in conflict resolution in model versioning.

Kurzfassung

Die technologischen Fortschritte in der Softwareentwicklung erlauben die Erstellung immer größerer und komplexerer Softwaresysteme, die wiederum nur durch geeignete Abstraktionsmechanismen zu bewältigen sind. Daher suchen Forscher und Anwender seit Jahrzehnten nach neuen Technologien und Techniken, um den Entwickler von dieser Komplexität abzuschirmen. Die modell getriebene Softwareentwicklung (model-driven software development) stellt die nächste Ebene der Abstraktion dar. Hierbei werden Modelle zu zentralen Artefakten der Softwareentwicklung.

Software wird heutzutage meist in Teams entwickelt. Modelle selbst fördern die Zusammenarbeit, da sie zusätzlich als gemeinsame Sprache zwischen verschiedenen Experten dienen. Durch die gemeinsame Entwicklung von Modellen durch mehrere Personen müssen diese der Versionskontrolle unterzogen werden. Die Mechanismen der Versionierung, welche die gleichzeitige Arbeit erst ermöglicht, sind stetiger Weiterentwicklung unterworfen. Obwohl diese Techniken für die code-zentrierte Softwareentwicklung sehr gute Unterstützung bieten, hält die Versionierung in der modell getriebenen Softwareentwicklung noch einige Herausforderungen bereit. Die Lösung von Konflikten, die durch gleichzeitige Bearbeitung von Modellen entstehen, ist ein besonders heikler Prozess, wobei jedoch die besten Ergebnisse durch die gemeinsame Problemlösung erzielt werden.

Die Fortschritte in der modell getriebenen Softwareentwicklung scheinen reduziert auf technische Aspekte, während soziale Faktoren, also jene Faktoren, die von Menschen abhängen, weitgehend vernachlässigt werden. Diese Tatsache ist überraschend, verbringen doch Softwareentwickler mindestens 60% ihrer Zeit mit kollaborativen Aktivitäten wie Kommunikation, Koordination, und die gemeinsame Ausführung von Aufgaben. Gut funktionierende Zusammenarbeit in einem Team führt in jeder Hinsicht zu einem besseren Ergebnis. Die potentiell sehr hohen Gewinne durch effiziente Kollaboration ergeben sich nur aus der aktiven Gestaltung der Zusammenarbeit.

In dieser Diplomarbeit präsentieren wir Möglichkeiten, die Prozesse in der modell getriebenen Softwareentwicklung durch die Förderung der Kollaboration zu verbessern. Bestehende Mittel und Methoden der Zusammenarbeit in Theorie und Praxis werden hierfür analysiert, sowohl in der Softwareentwicklung als auch in anderen Bereichen. Wir zeigen die Notwendigkeit auf, der Kollaboration in Forschung und Praxis einen höheren Stellenwert beizumessen. Für den besonders heiklen Prozess der Konfliktlösung in der Modellversionierung präsentieren wir ein spezifisches Anwendungsbeispiel für verbesserte Kollaboration, welche zu höherer Qualität in der Software und effizienteren Entwicklungsprozessen führt.

Contents

CHAPTER 1

Introduction

First, we want our readers to know what motivated this thesis, and what is to be expected. The first chapter therefore contains the incentive, problem definition, research objectives, and last, the structure of the thesis.

1.1 Incentive

As software engineering (SE) advances, coming along with increasing requirements on software, the systems in development get larger and more complex, consisting of a large number of components [37]. The growing complexity makes high level representations of the system to develop inevitable [35]. Despite the significant advances in code-centric software engineering concerning programming languages and IDEs, the development of these complex systems still requires herculean effort, and can overwhelm the available implementation abstractions [48].

A solution can be found in model-driven engineering (MDE), where models replace code as the primary software artefacts, and are used throughout the whole SE life-cycle [37, 84, 84, 115]. Models provide abstractions above those available at the code level. Complex systems are described at multiple levels of abstraction, forming a wide variety of perspectives, and thereby aiming to shield the developers from the complexities [48]. MDE aims to provide support for modelling structural and functional aspects, next to system attributes [48]. By introducing a separation of concerns through different types of models, MDE makes software engineering activities more systematic [115].

The MDE approach, aiming to raise the level of abstraction, already gained foothold in industrial practice, facilitated by the Object Management Group (OMG)[1]. Technologies to support model-driven engineering are developed by leading vendors and open source projects [116]. While this development is on the rise, it still faces major challenges [112]. The proper infrastructure is not given yet. Current technologies provide rather basic or limited support for

[1] http://www.omg.org

MDE [48], and techniques used in code-centric software engineering can not always be transferred to model-driven software engineering.

The former mentioned growing complexity of software systems in development is only manageable in teams, and consequently raises the need for concurrent versioning, which is a well established practice to manage parallel development [35]. Models, as well as source code, require versioning management. Unfortunately, the line-oriented approach for comparing versions of code fragments in order to find potential conflicts in traditional software engineering is generally useless when applied on models. Potential conflicts between versions in parallel model-driven engineering—with the range of version control systems available—usually have to be solved manually, including pen and paper, and time-consuming meetings. When these conflicts are solved automatically, the results are not satisfying. Just going back to a former version or randomly choosing one of the conflicting parts over the other is no efficient way of conflict resolution. Still, these usually are the solutions offered by tools intended to aid with conflict resolution. Source code version control and change conflict resolution, no doubt very important in modern software development [10], is put into practice and works out well in traditional code development. Regarding MDE, these issues are subject to ongoing research.

The project AMOR (Adaptable Model Versioning) [4] aims to address not yet solved challenges in MDE by providing improved versioning mechanisms for modelling languages. The main goal is to build an adaptable versioning framework with precise conflict detection and intelligent conflict resolution. So far, conflict detection in model versioning is an error-prone task, often leading to conflicts either detected wrongly or not at all. AMOR strives to provide means for smart conflict resolution support, this specifically includes the graphical visualization of differences and suggestions for conflict resolution.

Usually, research—concerning enhancements like this—is focused only on technical aspects, while collaboration and social aspects are neglected completely. Collaboration in the described domain is facilitated by versioning management, allowing the developers to concurrently work on a project. Communication is accomplished by extern devices like the phone or e-mails, meetings cover the rest. Still, a lot of tasks are left to be solely done by one person, where instead collaboration would be beneficial. AMOR's approach highly supports the formerly mentioned process of inefficient conflict resolution in versioning management in MDE. That is, using patterns provided through already solved similar conflicts, and attending to the graphical visualization of the problem. Additionally, a collaborative resolution of the problem would be desirable. With the involvement of all the conflicting parties and not one person in charge of the solution, supplementary information is provided, which otherwise would be lost. This information includes motivation for changes made, and different opinions can be taken into account.

One first idea to realise this approach is the Collaborative Conflict Resolver (CCR) [19]. This extension of AMOR not only deals with solving conflicts but provides support to enhance the collaborative process in doing so.

In this work, we show the potential of such a component. We survey various collaborative tools and techniques, and strive to determine the most useful way to integrate those into the CCR.

1.2 Problem Definition

In large software engineering projects, usually a number of developers work together to create a common outcome [73]. The collaboration of different types of specialists is required [110]. They communicate, share information, and perform independent tasks as well as corporate tasks.

Neither the costs for bad nor the benefits of good collaboration are directly visible in the outcome. Activities and schedules want to be coordinated, and problems have to be communicated and solved. The costs, which are for example expressed in time, rise, if there are no efforts made to support the collaborative process. This has been known for a long time. Still, the actual efforts to improve this part of work barely match the benefits which can be gained. For example, systems created for optimistic versioning in software engineering only recently were enhanced to offer actual collaborative conflict resolution, aided by communication features. The necessary process of manual conflict resolution was, and often still is, left to be done with pen and paper, or—with tool support—left to one of the conflicting parties solely, resulting in inferior solutions.

In various fields, there have been different attempts to support the collaborative process. This has been done on a theoretical level, by studies concerning the usability of either certain additional tools [92], or policies on the social level [1, 43, 114]. Also, support was realised in off-the-shelf applications [24, 45, 61, 64], or added to existing software systems [26, 83].

The support for collaboration in these systems and studies is usually limited, either to small parts of the process, or to certain tools or methods of communication. These attempts often are too general, altogether offering poor support for collaboration, or too specific, related only to certain situations or limiting the possibilities through very strict guidelines.

Statistics concerning the usability of specific tools or methods provide very different and often not comparable conclusions, for the results depend on the size of the team, the main tasks, and numerous other criteria. The only common outcome is that tool support, or efforts to support collaboration in any other way, are always beneficial.

When teams are working with models in parallel, specific problems and conflicts arise. A supporting collaborative tool would have to be responsive to these particularities. Additionally, common requirements have to be fulfilled by such a tool. Common requirements, which do not only occur when working with models, but generally occur in collaborative processes and therefore have to be minded as well.

1.3 Research Objectives

The main goal of this thesis is the investigation of methods supporting efficient collaboration in the context of model versioning. These methods shall be encorporated into model versioning systems like AMOR [7].

The plan of action, or less declamatory expressed, the guideline we will produce, includes what to do when and how to efficiently improve the process through collaboration. Within this thesis, the following questions are elaborated.

- What means of communication can be identified, and how useful are those in general and for our case?

- At which point in the software development life-cycle, that is for which tasks and under what circumstances, are different ways of collaboration useful?
- How should methods of collaboration be set, and policies and rules established, for the chosen way of communication to work out best? Which social aspects have to be observed additionally?

We offer a guideline for organisational aspects, specifications on how to coordinate the collaborative work, i.e. how to collaborate. We show the importance and the advantages of good collaboration, and assemble means and methods to boost efficiency. By determining the general suitability and the dependence on circumstances, we create this guideline for well working cooperation.

1.4 Structure of this Thesis

The structure of this thesis is reflective of our approach. Chapter 2 describes the basics of our pursued goal and therefore the exact field where our research is settled, which includes model-driven engineering, version control in general and model versioning, as well as challenges in model versioning.

Chapter 3 deals with collaboration in general. Occurrence and importance of collaboration are explained, as well as influences and restrictions on collaboration. We divide means and methods for communication into categories according to their occurrence and therefore importance.

Chapter 4 shows a selection of conceptional and practically realised approaches. Relevant studies from various fields, including version control in general, teaching and learning, and games, among others, are considered.

Chapter 5 describes our general results. We subsume possibilities for collaboration, means and methods of communication, resulting from Chapter 4. Also the usefulness of these technologies and techniques for model versioning systems like AMOR are determined. We show how the found alternatives can be combined in the most effective way, and under which circumstances the practicabilities for certain alternatives change. Social aspects are discussed, ranging from user acceptance and the allocation of authorisations and responsibilities to the influences through distributed software engineering.

In Chapter 6 we discuss the appliance of our general results and introduce the project AMOR, which inspired this work. Before our conclusion, we present a specific proposal for the utilisation of our results by providing a detailed concept for the Collaborative Conflict Resolver (CCR), a component aiming to aid the users in collaborative conflict resolution in model versioning [19]. This includes the features to be integrated and the design of the user interface. We describe how conflict resolution works within the CCR, that is, the exact course of such a process, and point out additional use for the CCR.

CHAPTER 2

Background

We aim for better collaborative support in model versioning. This chapter provides relevant information concerning the background of the topic. The first section describes models, their origin and their use, including model-driven development. The second section introduces version control as part of software configuration management. Last, we discuss challenges left to be solved in model versioning. These challenges concern technical aspects, while other challenges on the social level will be discussed in the next chapter.

2.1 Models in Development

"A model is a simplification of reality" [14]. A model is a description or specification of a system and its environment, intended for a specific purpose, and is represented by a combination of drawings and text [2]. Building a description of a domain by modelling increases the understanding of this domain [46].

Models have different purposes. Fowler [47] distinguishes three modes of usage for models: sketch, blue print, and programming language. At first, models provide a way to communicate ideas and alternatives. These sketches aim to improve understanding rather than provide completeness. Models can be used as drafts, to document decisions for design. Fowler calls these drafts blue prints. Blue prints are developed by system designers, they are detailed and complete to procure a foundation and instructions for the implementation by the programmer. Sketches and blue prints can best be distinguished by their degree of completeness. In model-driven development, models are used as a programming language. Here, applications are generated automatically through models.

Within the last years, models steadily gained importance, first with the CASE-approach [76, 93], later in model-driven development. This section briefly discusses modelling languages and the CASE approach. Then, in more detail, we describe model-driven software development (MDSD), including its progression, its promises, and the actual state-of-the art.

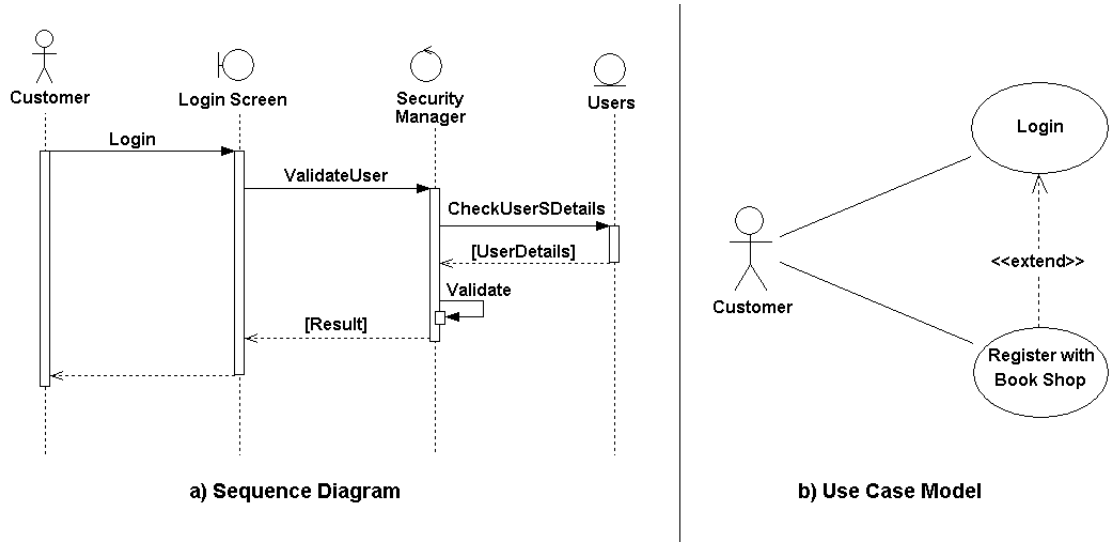


Figure 2.1: Examples of UML diagrams

Modelling Languages

When modelling a system, there is a need for a modelling language to express the description of the system [46]. A modelling language is an artificial language, consisting like any language in vocabulary—a set of modelling elements, and grammar—a set of rules on how to use those elements. A model, made with the predefined elements and rules, can describe a process, a state, or a system, among others. There is a vast number of existing modelling languages. Most graphical modelling languages are imprecise, the notation is rather defined by intuition instead of formal definitions. Informal methods still are useful to procure sketches and blue prints [47].

At the moment the de-facto standard for object-oriented modelling is the Unified Modeling Language (UML), which offers different viewpoints and levels of abstractions [97]. The Object Constraint Language (OCL) describes rules that apply to UML [82]. UML provides fourteen types of diagrams (UML 2.2) to represent static application structure, general types of behavior, and different aspects of interactions. Structure diagrams are Class Diagrams and Object Diagrams. Activity Diagrams and Use Case Diagrams model behaviour, and interaction can for example be represented in Sequence Diagrams.

Figure 2.1 gives some examples of UML models, redrawn from the Sparx Systems UML tutorial[1]. The Sequence Diagram shows object interactions over time, here a login process. The Use Case Diagram describes the proposed functionality of the system, namely the processes of a costumer first registering and then loging into the system. An example of an Activity Diagram is presented in Figure **??**, which describes a workflow of a system, including potential parallel activities.

The UML provides a set of graphical notations, with a metamodel as foundation [47]. Notation describes the graphical elements that models are comprised of, i.e. the syntax of the

[1] http://www.sparxsystems.com/uml-tutorial.html

modelling language. The metamodel of the Class Diagrams defines, how elements and concepts are connected. Elements of the Class Diagram are classes, associations, and multiplicities, among others. The importance of the metamodel changes with the intended use for the model. The metamodel is not important when drawing a sketch. When designing a blue print, it should be regarded. When using the UML as a programming language, the metamodel is highly important, for it defines the abstract syntax of the language. The formerly mentioned 14 diagram types of the UML 2 standard merely represent aspects of the metamodel. The UML standard states that certain elements are typical for certain diagram types, but they can also be used in combination with another diagram type [47].

The UML is the most popular, and yet, most criticised modelling language. Hansen et al. [55] identify three characteristics of the UML, that inhibit learning. UML is very extensive, there are several diagram types, more than 100 distinct metaclasses, and even more relationships between those. Next to extent and complexity, usability is the third described characteristic. The Unified Modeling Language is large and complex, yet it offers no support for common modelling practices like free-hand drawing and the inclusion of other diagramming styles [34]. These characteristics were the motivation for Hansen et al. to create Ideogramic UML [61]. UML's size and complexity not only inhibits learning the modelling language, but can present a problem to software developers and OMG working groups evolving the standard [49].

The CASE Approach

In computer-assisted, also computer-aided, software engineering (CASE), tools are used to automate software engineering and aid the developers in the process of designing and constructing software systems [110]. CASE tools, conceived in the 1980's, were designed to solve the problem of managing and manipulating diagrams and the associated repositories [76], and aid the system developer in specifying, designing, and constructing software systems [110]. CASE tools enable developers to express designs in terms of general-purpose graphical programming representations [93].

In research and literature, the CASE approach and CASE tools received much attention, but they were rarely adopted in practice. The reasons are various. Graphical representations offered by CASE tools were too generic and not customizable. Underlying platforms lacked the support for important properties, such as transparent distribution, fault tolerance, and security. Another reason may have been the missing support for concurrent engineering in CASE tools—only single-user or serialized work was possible [93]. More recently, CASE tools offer the ability to produce working software systems out of models [47, 76]. McGinnes [76] states that although modelling techniques have been beneficially transferred to the CASE environment, they were not necessarily designed with automation in mind, and important aspects for the modelling process, like communication, collaboration, and application of past experience, are not considered.

Model-Driven Software Development

Software programmers and researchers have been creating abstractions to help them program in terms of design intent and shield them from complexities of the underlying computing environment for the last five decades [93]. Early programming languages shielded the programmers

from the complexity of programming with machine code, early operating systems (OS) shielded them from complexities of programming directly to hardware. The advances in software development lead to ever increasing complexity of software systems, which highlight the inadequacies of modern programming languages [84]. Therewith rises the demand for methods and languages with a higher level of abstraction [49].

The abstraction power of models aids in the development of the increasingly complex software systems, and shields the developers from the rising complexity. Models provide multiple levels of abstraction, and a wide variety of perspectives. Structural and functional aspects can be described with models [48], and separation of concerns is provided by different types of models [115]. Model-driven engineering (MDE) aims to address the inability of modern programming languages to reduce the complexity and express domain concepts effectively [93].

In MDE, models—the abstract representations of source code—are major software artefacts [37,84,115], transformed into machine-executable forms [49]. The following section introduces model-driven software development (MDSD), and relevant aspects and approaches within MDSD, like model-driven architecture (MDA).

From Code-Centric to Model-Centric Development

Software systems grow larger and more complex through advances in software development and information processing technologies, therewith inadequacies of the abstractions provided by modern high-level programming languages surface [49], and high level representations of the system to develop are inevitable [35, 49]. In order to simplify the design and maintenance of software systems, and to adjust to the rapid change of technologies, models can be used. With modelling languages, the complexity of software systems in development is reduced, for models provide abstractions above those available at the code level. More importantly, the developers are shielded from intricate lower-level details [48, 49].

Due to these benefits of model-centric code development compared to code-centric development, there is a shift from the latter to the former kind of development [12, 89].

What Models in Development Promise

Much is expected of the shift from code-centric to model-centric development. This trend holds many promises about potential benefits [12, 73, 103]. Nothing less than an increase in productivity is expected [57]. In particular, the main promises model-driven development holds according to Hovsepyan et al. [59] are the following:

- **Save time.** The development with models saves time by code generation, it takes place on a level above the source code. Also, architectural elements can be reused.
- **Improve quality.** An improvement in quality is promised, by using well established and tested transformations.
- **Platform independence.** Code developed for a specific platform often has to be reimplemented for another, while models promise cross-platform development and enhanced platform migration.

- **Facilitate interaction and communication.** Different domain experts may communicate through models as a common language. Through models—with defined semantics—as common language, misunderstandings are reduced and thereby the communication facilitated.

Definition of Terms

In the following, we briefly discuss relevant terms in model-driven development.

Model-driven and model-based. In model-based development, a process is described with models, which then may be developed further. In model-driven development, models are transformed one to another, and reused amongst different platforms. Models, as central concepts, are the basis for an automated system development process [103].

Model-driven engineering. Models provide understanding of system behavior and implementations can be generated from models in MDE [49]. The kind of software engineering, where models are primary software artefacts, is called model-driven engineering. According to the Standard Glossary of Software Engineering Terminology (lEEE Std 610.121990) [62], software engineering is "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software". The same applies to model-driven engineering, a kind of software engineering.

Model-driven development. Model-driven development is part of MDE. Both include tasks such as requirements, design, testing or maintenance. The lEEE Standard 610.121990 [62] defines the software development process, as the process "by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use". Most authors, whether their writings concern MDE or any other kind of SE, do not differentiate between the terms engineering and development. Often, for example see [72], software development (SD) is described as part of software engineering (SE).

In this thesis, we use the term engineering in most cases. We rather refer to development, when the software development process is in the foreground, or the cited author did so for a reason.

Model-driven architecture. Model-driven architecture (MDA) is a software design approach to MDD from the Object Management Group (OMG), where models are first-class entities throughout the software life-cycle [48, 73], i.e. main development artefacts, used to describe various aspects of software, and thereby supporting model-driven engineering (MDE) [84]. Hence, one way to implement MDE is using the OMG's model-driven architecture (MDA) [49].

Model transformation and composition. Two essential management tasks in MDD are model transformation and model composition [84]. According to Oliveira et al. [84], transformation has been extensively researched, while model composition needs further investigation. For an integrated view of a system, models must be composed. The composition—a generic operation—is defined by the composition operation, a type of transformation [42]. Model composition consists of combining input model A and model B to generate an output model O, combining the content in both models [42, 84]. Oliveira et al. provide model transformation rules and a model composition strategy in [84].

State-of-the-Art in MDE

Mode-driven engineering, facilitated by the Object Management Group (OMG), the most representative organization in the definition of model-driven engineering (MDE) [2], has been around for several years, and gained foothold in industrial practice [103]. MDE is highly favoured in recent literature, yet available technology does not offer adequate support [48].

Models, like any other software engineering artefacts, have to be put under version control (VC). While concurrent versioning systems have been largely adopted to manage a source code repository, software models are inadequately managed. The VC systems usually are designed for source code and not reusable for other artefacts like models [11, 35]. Software model versioning is generally not supported by modelling tools, support often is limited to maintaining different versions of the files only [8, 35]. Also, some standards concerning MDE still have to be defined. The issues of the proper infrastructure not given yet will be discussed in Section 2.2 and Section 2.3, after introducing the reader to the basic concepts of version control.

2.2 Version Control

During the software development life cycle many activities must be carried out consistently through a number of development phases. The activities can be carried out more effectively when planned. Components produced in different phases of the life cycle pass through a number of different versions. The versions and their relationships have to be coordinated. Two very common activities, software configuration management (SCM) and version control, deal with the characteristics of complex systems [6]. SCM and VC activities are particularly important for software engineers, for during the system life cycle there is a growing number of versions of a single component [6]. Software configuration management focuses on forming and maintaining consistency of a system's performance. Version control facilitates storing and sharing the versions of one artefact. VC is a software configuration management activity.

Software Configuration Management

Increasing the number of developers involved does not necessarily decrease development time of a software engineering project. The more developers, the more coordination effort has to be made. Software development is not easily managed, for a modification can affect the entire system, and changes made can impact the work of other team members. Software Configuration Management (SCM) reduces the complexity of coordinating efforts [53], and aids the developers in controlling the configuration of the system [6].

"Change occurs throughout the software life cycle" [70]. SCM provides the foundation for effective control change, and, as a part of configuration management, tools and methods to maintain consistency, integrity, and traceability. Risks and costs of changes are minimised by identifying and establishing baselines, and tracking and controlling changes [63, 70]. A SCM plan specifies which activities have to be done, when, by whom, and what resources they require. The IEEE Standard for SCM plans [63] defines the minimum required contents of a Software Configuration Management plan, which apply to the entire life cycle. In the following we shortly describe the required content, also called the SCM classes of information.

Introduction. The introduction information includes purpose of the SCM plan, scope, definition of key terms, and references—altogether providing an overview of the SCM activities.

SCM management. Responsibilities and authorities to organizations and individuals for SCM activities, as well as their management, are part of the SCM management information.

SCM activities. This information includes all functions and tasks required to manage the configuration of the software system, both technical and managerial activities.

SCM schedule. The SCM schedule information includes coordination of identified activities, their sequence and dependencies. For example, how a key activity is related to a project milestone.

SCM resource. Environment, infrastructure, personnel, tools, and techniques among others are defined by the SCM resource information.

SCM plan maintenance. Activities necessary to ensure the continued SCM planning throughout the life cycle are included in the SCM plan maintenance information.

SCM and Models

Traditional software configuration management systems are operating on the abstraction of a file system, supporting textual artifacts like source code. The changes made are managed on a line-oriented level. The line-oriented approach is not suitable for models as main development artefacts. Kögel [70] concludes that "traditional SCM systems are inadequate for managing artefacts with graph structure and for supporting traceability". Still, there is a growing need for SCM techniques that are able to support models. For a classification and comparison of different SCM systems we kindly refer to [32].

SCM activities—whether they are planned or not—are part of and crucial to every software development project [63]. An important SCM activity is version control. VC facilitates storage and management of different consecutive or parallel versions of software artefacts and therefore allows parallel development.

Version Control Basics

Version control (VC) is an important activity in high-quality software production. It is needed to manage changes in documents, files, and programs. Version control aids when dealing with concurrent and consecutive versions. The structure used by version control systems (VCS) is similar to the structure in file systems. In VCSs file content and internal structure is considered, including details as classes, methods, control blocks and others [69]. With version control used in SE, different versions of textual artefacts are kept and organised, making it possible to fall back to previous versions, if for example the former version turned out to be better than the newer one. Another advantage is to be able to compare different variants. Changes can be retraced, even several development lines can be reconstructed in parallel. Even if there is not a team at work but a single person, using a tool to support version control is beneficial. Of course then no conflicts arise between versions of different team members.

Having consecutive versions allows the user to go back to a former version, or look up previous versions [75]. For example, if the consecutive version does not work, the first option

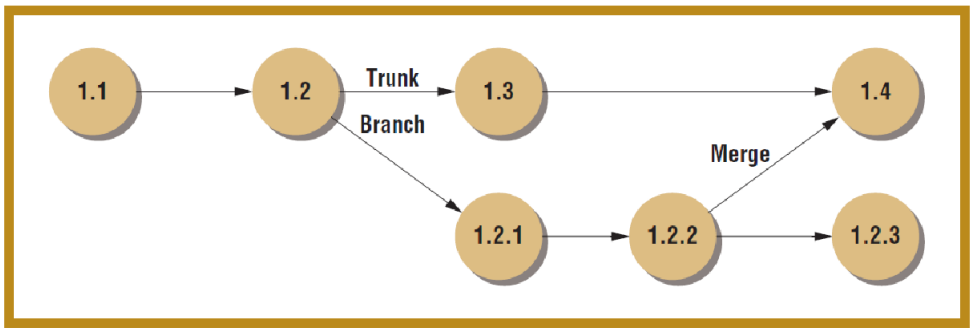


Figure 2.2: A CVS branching example [75]

may work like an undo-function, for it allows the user to disregard the newer—not working version—and go back to the former—working—version. The second option allows the user to look up the difference between the last working version and the newer one, figuring out, because of what differences the malfunction occured.

Version control does not only allow consecutive, but parallel development. Figure 2.2 illustrates an example of the combination of versioning possibilities. The main development line is called *trunk*, additionally branches offer the possibility to start parallel development lines, used for either a project still in progress while keeping the already delivered version or customizing a project without affecting the main development line. Any branch can be merged with the trunk again. Also concurrent versions allows multiple users to work simultaneously.

The simplest way of versioning is one version of an artefact replacing another as the current version, this is called a “sequence” (see Figure 2.3). Changes made in a component v1 lead to component v2. The version control system used still allows access to v1. Other version graphs are a “tree” and an “acyclic graph”, displayed in Figure 2.3. If the version of one component (v1) is changed into several other versions consecutively or parallel to versions v2 and v4, there are two or more branches. This structure is called a *tree*. These new versions (v2 and v4) can be developed further individually, and at any time merged into a single version again. An *acyclic graph* includes separation of the development line through several versions created of one former version, and merging those new versions into one version again. That is, the branches of a tree are brought together and form the acyclic graph. The merge describes the fusion process of two different versions of the same component. The cost for parallel modification of one artifact is a potential conflict when those versions are combined again.

In an acyclic graph versions of artefacts are—contrary to sequence and tree—merged together again. If independent parts of the component are different in the different versions, the versions can be merged. If not, conflicts between those versions arise. Merging conflicting versions to a single one can be accomplished by choosing one of the versions over the other one, for example the later committed version or the version of the user with the higher rank can be chosen. In this case the information of the other version, still stored in the version control system, is lost for the main development line. Actually merging versions includes consideration of the information included in both versions in question. In general, the consideration of all included

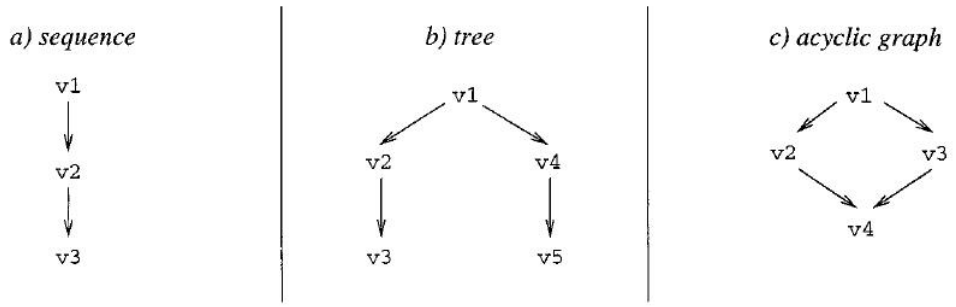


Figure 2.3: Version graphs: a) sequence, b) tree, c) acyclic graph [32]

information, especially the evaluation of the changes value in comparison, is a manual task.

Optimistic and Pessimistic Versioning

The *pessimistic approach* of versioning was incorporated by early version control systems, such as Revision Control System[2] (RCS) and Source Code Control System[3] (SCCS) [53]. Pessimistic versioning means that only one person may modify a fragment at a certain point in time.

In pessimistic versioning, as well as in optimistic versioning, the main software version is seperated from the parts different users are actually working on. The master copies, which are stored in a repository, are separated from the working copies, on which the programmers work (see Figure 2.4).

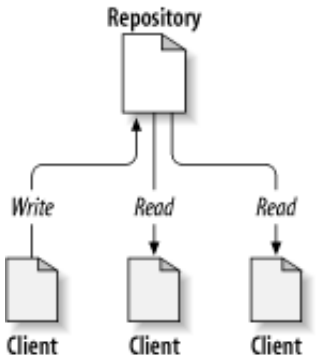


Figure 2.4: The repository and the working copies [31]

When using the pessimistic approach, checked out fragments are locked, thereby disabling anyone else to modify it until the checked out version is committed and therefore unlocked again. This is also called the *lock-modify-unlock model*. Figure 2.5 provides an example of pessimistic versioning. While Harry uses component A, that component is locked in the repository, and

[2]http://www.cs.purdue.edu/homes/trinkle/RCS
[3]http://sccs.berlios.de

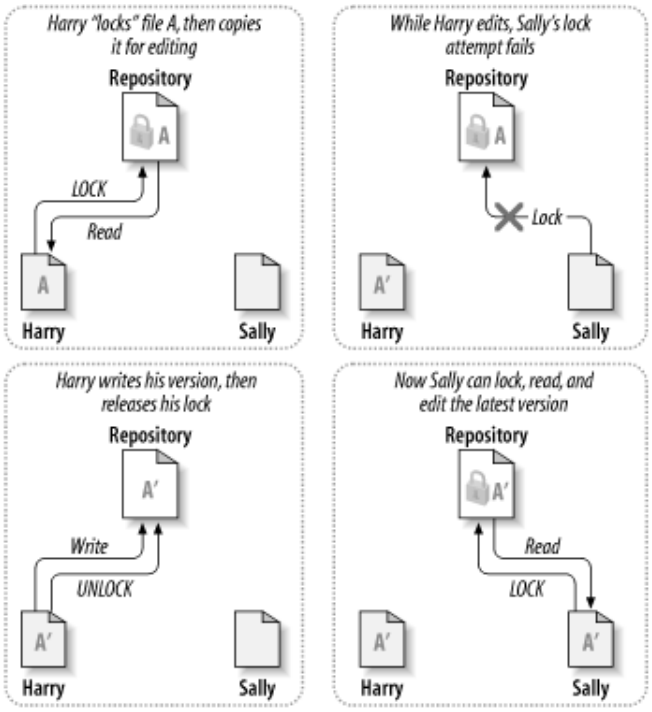


Figure 2.5: The lock-modify-unlock model [31]

Sally is not allowed to edit it. After Harry uploads (writes) his version back into the repository, Sally can check out component A—now the version modified by Harry—which is locked for Harry to use while used by Sally.

Conflicts can not materialise in the first place, with only stable changed fragments checked in and without the possibility for a second person to make conflicting changes. Although, locked versions can be checked out and viewed, but not modified [75]. In a larger development team parts of the software are therefore locked, and there is a high probability, different developers cannot fully attend to their tasks [73].

To avoid inhibition of co-workers tasks by locked parts of the system in development, users have to be able to modify parts of the system concurrently. In *optimistic versioning* several users can check-out the same fragment concurrently. The optimistic approach is also called the *copy-modify-merge model* [75]. The potential problem to avoid in optimistic versioning is displayed in Figure 2.6. Harry and Sally check out component A at the same time, and modify it. Harry first commits his new version A' to the repository. When Sally commits her version A", she would accidentally overwrites Harry's version A', if no dedicated conflict detection support is provided.

Due to the possibility to concurrently modify an artefact (A) in optimistic versioning, two different versions (A' and A") may be created, when a single version is required to continue the process. Figure 2.6 describes the accidental overwriting of a version with another. But a possible solution to deal with the parallel version A' and A" is to deliberately overwrite one of

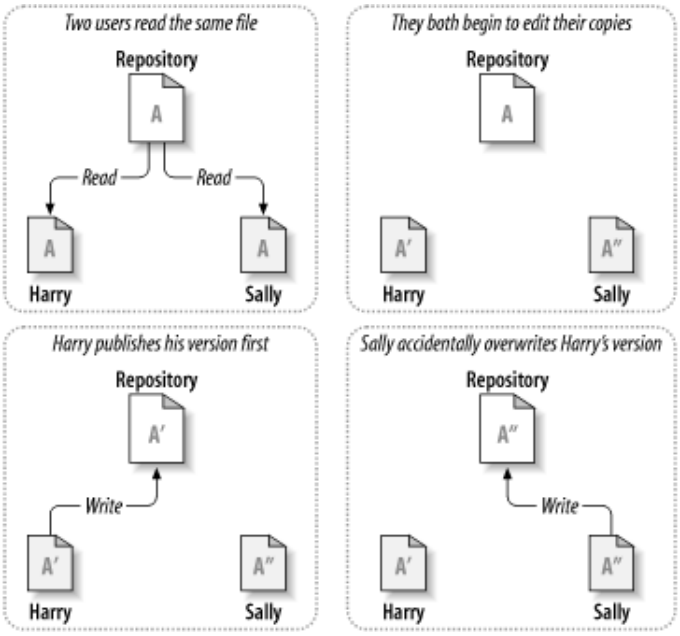


Figure 2.6: Optimistic version control without dedicated conflict detection support [31]

these versions. While users are allowed to concurrently check out artefacts, the repository may maintain only one version by overwriting. Which version to keep can be chosen for example according to the time of the check-in or the user privileges. This probably is the worst way to deal with concurrent versions, for one of the versions gets lost.

In the copy-modify-merge solution, there is a mechanism in place to prohibit accidental overwriting of other versions. Harry and Sally again concurrently check out component A (see Figure 2.7). Both copies are modified. Sally checks in her version A". Check-in of modified versions is naturally possible, if there has no other version been committed before. When a second version is committed, conflicts may arise, which have to be dealt with. These conflicts can be solved automatically, semi-automatically or manually. In this example, Harry carries out the merge manually. When Harry wants to commit his version A', he gets an "out-of-date" error, informing him about the modified component A in the repository, which now is A". Harry is not allowed to write his version into the repository. He checks out the new version of component A (A"), compares that version with his own modifications, and then creates a new merged version A* (See Figure 2.8). The merged version A* is published and Harry and Sally can both continue their work on component A*.

The copy-modify-merge solution introduces the combination of two parallel versions—a merge. Manually merging two versions usually is left to one of the parties involved. In the example above, Harry is solely in charge of creating the new version. Harry can chose to incorporate certain changes of his while disregarding conflicting changes Sally made. Since he does not know Sally's motivation behind her changes, his manual merge may still lead to advanta-

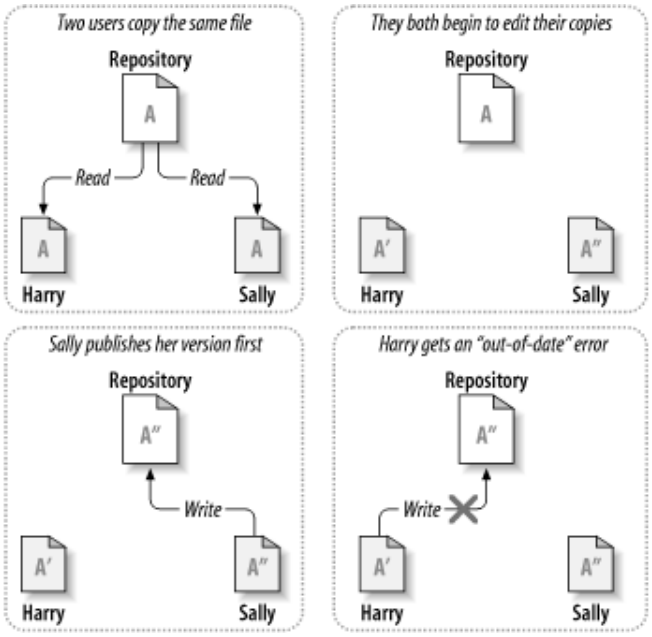


Figure 2.7: The copy-modify-merge solution part 1 [31]

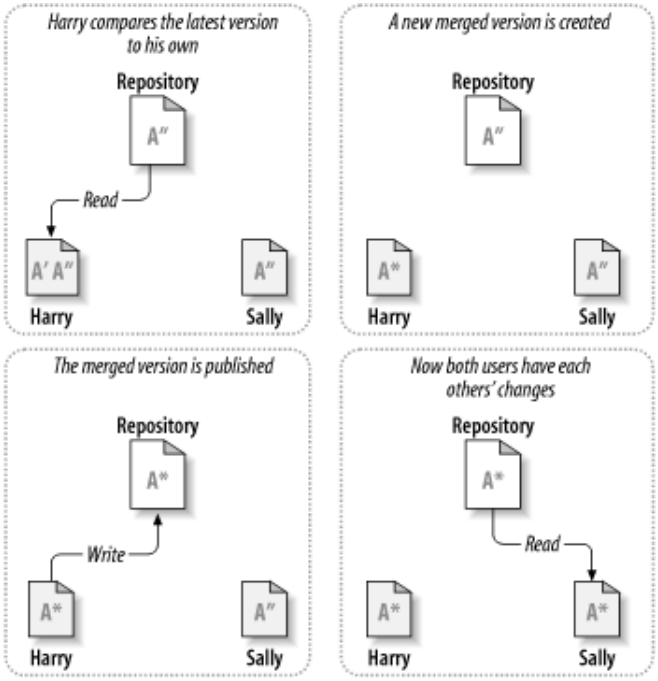


Figure 2.8: The copy-modify-merge solution part 2 [31]

geous progress being lost. Automatic merge of versions is possible, if there are no conflicting parts in those versions. A lot of versioning tools support automatic merge of versions. This automated merge support is mainly accomplished for line-oriented code-development, while for model-driven development these advancements can not be transferred because of the graph-based structure of models [95]. Versioning of models will be discussed in Section 2.2.

Tool Support for Versioning

Two important and challenging issues a team in software development is faced with periodically are to have access to previous software versions, and to be able to simultaneously work on the same artefact. To solve these issues the developers can either rely on version control tools or deal with it manually. Manual version control means the user makes regular backup copies. To support the version control process with a tool is recommended, for here the human factor can lead to errors and data loss [22]. Usually software development with a VCS is only marginally more complicated than without [102], but the advantages are obvious. Louridas [75] states that a software organisation that does not use a version control system is reckless and oblivious to the demands of actual project development. Tool support to handle different versions of programming code are available at any cost for any platform.

Tool support is indispensable for different reasons. Software development usually is a group effort, and version control systems help to coordinate the work of the team members. Supporting tools can strongly limit team members constricting each others work.

Concurrent Versions System

Concurrent Versions System (CVS) [86] is one of the oldest version control systems. Its development started in 1986, and it still is widely used. CVS is a standard to which other VC tools are compared [75]. We therefore name CVS as an example of version control in general.

CVS supports a shared repository. The repository can include any kind of file, for CVS is not concerned with compilers or programming. CVS maintains a history of changes, and provides notifications to users trying to record different changes to the same file, as well as log-files containing changes made [56].

Common commands used within the versioning processes in CVS [56] include *checkout*, to check out a copy of a file in the repository, and *commit*, to commit changes made in local copy to the repository. The work on files is not done directly within the repository. The working copies are checked out, changed and then the stable version is committed [86]. CVS checks if there has been a newer version committed. If there was no newer version committed, the user has to enter a log message describing the changes for documentation. If there was another version checked in in the meantime, merging the two versions may result in a conflict situation, which requires manual intervention [56, 86]. Before check-in, the *diff* command may be used to be provided with a list of changes made since the last commit. Independent development lines, called branches (see Figure 2.2), may be created and joined together at any given time, using *tag* and *update* commands.

With every committed file, the CVS creates a new version in the repository storing the current version of the file and the differences between the current version and the previous one.

Thus the repository contains all copies and maintains the complete history of any file. If looking for a specific previous version, it can be found by searching for a specific date, also one can request a version by a tag.

There is a vast number of alternatives to CVS. Subversion is a very popular version control system. Features of subversion are described on the official Subversion website[4], which also describes the similarity of the features provided by Subversion and CVS. Both tools use the copy-modify-merge solution earlier described. Other version control systems are for example provided by Microsoft[5] or IBM[6].

Desirable Features and Best Practices

If there is no version control tool used, backup copies of the software version have to be made manually before any changes are made. Some development teams may prefer manual version control, but tools for this purpose have many advantages. The VCSs available on the market can be stand alone applications and some are encapsulated in software development tools. If the software development tool does not provide version control, an adequate tool has to be chosen.

Candrlic et al. [22] describe desirable features for VCS and compare a few version control tools with manual version control. In a very small group version control does not pose a particular challenge. The programmers work on their parts of the program without disturbing each other and the communication is usually facilitated by working in the same room. Therefore manual version control is possible and works well until the number of developers grows. Table 2.1 lists the desirable features for a version control tool.

No.	Goal	A desirable feature of a version control tool
1	reliability	enabling automatic backup
2	work of multiple programmers	simultaneous maintenance
3	better simultaneous maintenance	new code recognition on the code line level
4	keeping a project history record	access to all the previous versions
5	easier work	intuitive graphic user interface
6	data for managers	monitoring software development
7	program reports	making statistical reports
8	monitoring team effectiveness	making team and programmer reports
9	faster adaptation to new versions	commentaries on implemented changes
10	faster adaptation to new versions	version marking
11	simultaneous work on multiple problems	connecting various project branches

Table 2.1: Desirable features of a version control tool [22]

[4] http://subversion.apache.org/

[5] http://msdn.microsoft.com/de-de/library/cc434922%28VS.71%29.aspx

[6] http://www-01.ibm.com/software/awdtools/clearcase/

As important as choosing the adequate tool for version control is the efficient use of it. Spinellis [102] offers some handy ideas for best practices to amend the utilisation. Not only source code should be put under version control, but also everything else like documentation, help files, and GUI elements. A VCS is beneficial for personal projects too, which might be a web page or a phone book. Whatever the version control is applied on, the name should be picked carefully, for some tools get confused when a file name is changed. When working in a development team, clear policies and procedures have to be defined and followed, because the actions in the VCS affect the other team members. When all releases are labeled, every user is provided with a concrete name to associate with. Last but not least, it is expedient to perform a separate commit for every change. Every commit provides a list of changes made, precisely displaying which lines the change affected, leading to better traceability.

Model Versioning

The specifics of models complicate the versioning, but the success of MDE in practice depends on it [4]. Model-centric version control and model transformation testing are important activities in MDD [74]. Traditional optimistic versioning systems follow a line-oriented approach, versioning text files. Those can be applied on models by serialising them, for example with the XML Metadata Interchange (XMI). But thereby the graph-based structure of models is not taken into account [73], making additional manual support for conflict detection and resolution necessary [74].

The line-oriented approach is well-known, but insufficient for versioning models, due to the structure and the semantics of models [4, 71]. The conditions under which models are edited and used in the development process are significantly different from those valid for text. These differences have to be considered. Merge tools for models need to offer the same quality as those available for textual merges [94]. A tool adequately supporting the model-based development process is needed [3]. According to Altmanninger et al. [4], the existing possibilities of model versioning suffer from three main problems: erroneous conflict detection, unsupportive conflict resolution, and the inflexibility regarding different modelling languages.

Barrett et al. [10] see the merging of multiple models into a single consistent one as the main practical problem to be solved in MDE. For the success of the model-driven paradigm, consistency maintenance is required, the models' syntax and semantics have to be considered to perform adequate conflict detection and resolution [3].

2.3 Challenges in Model Versioning

There have been many advances in model-driven development, and it already gained foothold in industrial practice. Still, major challenges have to be faced to make MDD a success [112], especially concerning the versioning of models.

Tool support. In the last four decades, profound experiences in using versioning systems for textual programming languages were gained [95], and a multitude of tools—like CVS and Subversion (cf. Section 2.2)—have been realised, which support detection and resolution of conflicting modifications [4]. For MDD to be a success, parallel development of model artefacts—

as it is state-of-the-art for textual documents—is required [95], but current technologies provide rather basic or limited support for MDE [48]. Modelling tools rarely provide support for concurrency and consistency, and available concurrent versioning tools do not provide adequate support for software models [9, 35]. Applying code versioning techniques for line-oriented source code to XMI-serializations of models (cf. Section 2.2) is not reasonable for optimistic versioning approaches [4]. Available version control systems either are bound to a specific modelling language, which makes them inflexible, or are applicable on any language, which usually brings poor versioning support with it [4]. When developing a system to support model-driven development, this has to be balanced. That is, on one hand the certain characteristics of different languages and on the other hand precise specifications for these has to be regarded to support them properly [73].

Conflicts in Model Versioning

In collaborative software engineering, concurrent accesses to related pieces of code or models are made. These accesses usually are coordinated by a version control system, and they may result in concurrent, potentially conflicting modifications [40, 89].

Conflicting fragments in model-driven development can be caused by one person making inconsistent changes to a fragment. These undesired effects can occur for example through independently developed subclasses, and the base class evolved in an unexpected way, affecting inheritance relationships [77]. In optimistic versioning, where more than one person manipulate the same artefact concurrently, different intentions lead to conflicting modifications not clearly compatible [32].

We have already established, that version control tools for textual programming languages are insufficient for MDD, for they do not take the logical structure of models into account. The structure and semantics of models are very different to line-based code. The graph-based structure and the models' syntax and semantics [3, 19] complicate the collaborative process of realisation, for the optimistic versioning, including conflict detection and conflict resolution, is not yet as mature as that of line-based source code. For example, when different model elements, that is syntactically different parts of the model, are edited concurrently, these modifications may not result in an obvious conflict. Still this modification may have a *conflicting side effect* [89]. Without consideration of the semantics, this conflict may not be detected.

Avoiding and Reducing Conflicts

Small changes in the source code—or a model—often have a large impact [77]. Reducing conflicts reduces development costs very much. Existing conflicts have to be solved, which requires time and effort. Obviously the developers would want to reduce the number of occurring conflicts and avoid as many as possible. There are a number of suggestions to reduce conflicts. Providing the user with the option of not performing the new, conflicting operation, is the simplest strategy, to avoid a conflict [41]. Another one is to set the granularity of the considered software artefacts to a small size, that is the amount of detail to which merge conflicts can be detected. With fine-grained version control, where the changes are as small as possible, the number and size of merge conflicts can be reduced [77].

Edwards' timewarp conflict model [41] includes a conflict tolerance to ease the users' burden by allowing certain conflicts to exist in a timeline. Users can continue the work, and solve conflicts later, if at all. Bruegge et al. [21] suggest that changes to the main development line should be restricted to bug fixes, and any other change made within development branches, while the number of these branches should be as small as possible, to minimize merge conflicts. Murta et al. [80] propose to add an optional lock command to a version control system with an optimistic concurrency policy. This lock command is applied on specific model elements, and it includes the reason why this element was locked. These are some of the proposals on the technical level, or concerning version control policies, to avoid and reduce potential conflicts.

The first step of conflict resolution is to avoid conflicts, if a conflict does not arise, it does not have to be handled. Avoiding conflicts is also accomplished by collaboration put into practice well. Most important here are awareness and communication possibilities, the latter combined with behavioural policies. Providing proper information leads to the team members being aware of each other's changes, and overlapping changes can be anticipated [77]. To find potential conflicts in time, tool support is needed. The distribution of the information raising awareness can be accomplished by providing means of communication.

CHAPTER 3

Collaboration

Collaboration is defined by working together to achieve shared goals [98]. Wong et al. [114] state that to the software engineering process collaboration within a team of expert professionals is central, and that software development projects include countless informal and formal meetings. Statistics about collaboration in the SE process are offered by Robillard et al. [90], classifying collaborative activities, and stating that software engineers spend most of their time with activities of this kind.

According to Frost [50], now that software development has become highly collaborative, interacting with colleagues takes up more time than stringing together lines of code, that is the actual core task. Also, there is a shift from ad-hoc to formal collaborative activities. All these collaborative activities sometimes cause more problems than they solve. Collaboration can be time-consuming and problematic despite all its benefits. The escalating number of meetings, e-mails, discussions, and coordination efforts leaves less than half of the workday for the main task at hand [25]. Therefore it is important to shape collaboration efficiently, for a reduction of its costs and to gain all benefits possible.

The following chapter introduces basic characteristics of collaboration and explains the importance of actively shaping collaboration by explaining the impact of good and bad acted out teamwork. Important aspects of communication are discussed before methods of collaboration. Depending on the point of time in a process, different means and methods of collaboration are desired. Last, we will specifically discuss collaboration in MDE.

3.1 Impact of Collaboration

Already 25 years ago 65 percent of research articles were jointly written in some fields [85]. Nowadays with more and more collaborative programs for text editing and improved communication and networking through globalisation, collaborative writing is easier than ever. New tools make collaboration easier, and larger projects demand even larger groups working together. This applies to various fields. Collaboration can be found everywhere, of course in software engineering and its subcategories like model-driven engineering too.

Working together to complete large software engineering projects in acceptable time is essential. A single persons's work is error-prone, while co-workers can catch each others mistakes [112]. Studies by DeMarco et al. [39] suggest that developers on large SE projects spend 70 percent of their time working with others. According to Jones [66], team activities account for 85 percent of the costs of large systems. With the growth of a software engineering project and the increasing number of people involved, time and costs needed for collaborative activities rise as well. To handle complex systems, collaborative design is more and more required, which means the simultaneous use of information [38]. The amount of time and resources needed, rises with the size of a project. This means the direct work—the actual development process—decreases, while the indirect work, coordination and conflict resolution processes, is increasing.

Software projects have for long had sizes not feasible by one person, but require collaboration of several persons. Collaboration, essentially working together towards a common goal, or in other words teamwork, therefore is no foreign word in software engineering. Maintaining communication alone needs a lot of resources. The need to respond to e-mails, attend meetings, participate in group discussions, and manage source control take up a lot of time [50]. Good collaboration can lower the costs, as much as bad collaboration can boost those costs for collaborative activities.

New technical improvements—techniques, tools, and methods—continuously emerge to improve the software development process and the final product. Software is developed by people and for people. Social aspects, like human characteristics, behavior, and cooperation, often are neglected, even though they have to be considered, for they are central to the development process. These aspects impact the business of software engineering at many levels and from many perspectives [36].

Influences on and Restrictions for Collaboration

Social aspects are not to be underestimated. Cheng et al. [28] divide the social aspects into awareness, interaction and roles. Social awareness means to be aware of the availability of someone to talk to, the availability for interruption, and their level of distraction. In geographically distributed software engineering, supporting social awareness is more difficult. Social interaction is actual conversation between two or more people about a certain subject. Interaction can be verbal, or involve drawing or pictures. Also it can be synchronous or asynchronous. Rules and expectations for the conversation are defined as social roles. To Cheng et al.'s definition of social awareness we add the awareness of responsibilities and authorisations, which are defined by the social roles. This means to be aware not only to whom one *can* talk to, but also to whom one *has to* talk to.

How a team collaborates depends on different variables. These variables influence social awareness, interaction and roles. Some of these variables are alterable, some not. Variables like organisational structure and culture may be set and not changeable. Country-specific cultural aspects—another variable—define how people work together, but can not be altered easily. Tool support is an easily changeable variable. Supportive tools may be chosen and tailored, not only to the influencing variables mentioned before.

There are four main variables influencing the collaboration within the team, pictured in Figure 3.1.

Tool support. Collaboration can be influenced by the supportive tools, either software or hardware. Tools can provide means for social interaction. Interaction, that is the actual communication, can be accomplished for example by tools supporting instant messaging, e-mails, or the phone. Social awareness—to know to whom one can talk and when—can be facilitated by tools providing contact lists—a kind of address book—extended by profiles and current status. The knowledge of the current status of a co-worker offers the availability of someone to talk to and for interruption. Profiles offer knowledge about responsibilities and authorisations.

Missing tool support on the other hand can restrict social interaction and awareness. For example, Ideogramic UML (see Section 4.4) is best used on electronic whiteboards. If the whiteboard is not available, the application Ideogramic UML is of less use. A video conferencing system can be restricted by the size of screens, and facilitated through larger screens [104]. For both examples, proper software and hardware are needed.

Geographical aspects. Geographically distributed SE limits social awareness. Co-workers in the same building more easily know about their availability for interaction. Is the team distributed to different countries, creating social awareness is more difficult and therefore has to be facilitated to a greater extent.

Cultural aspects. Cultural aspects influence social interaction on two levels. First, cultural background defines individual behaviour. Favela et al. [43] for example found in a study incorporating a US and a Mexican team (see Section 4.5), how different individual behaviour can be with a different cultural background. To facilitate collaboration and avoid frustration between teams or co-workers with different cultural background, these cultural aspects have to be considered. The individual disposition for openness, that is to provide information about current availability, is part of individual behaviour influenced by the cultural background. Second, cultural aspects influence organisational structure and culture, which in turn affects social interaction.

Organisational structure and culture. Collaboration is influenced by the organisational structure and culture of a company. The company sets rules and policies defining social interaction. The company's hierarchy, may it be steep or shallow, defines the social roles. Depending on the position within a team—the assumed social role—expectations concerning interaction are formed.

Recapitulatory, collaboration is influenced by tool support, geographical and cultural aspects, as well as organisational structure and culture. Social awareness, interaction, and roles, as defined in [28], in turn influence each other. Social roles define expectations and rules concerning interaction, and therefore interaction itself. Social interaction and social awareness are closely connected, influencing each other.

Consequences of Good and Bad Collaboration

In general, software engineers work in teams, still the typical tools for development are effectively for single users only. It is essential to collaborate within software engineering, but most software engineering tools such as editors, compilers, and debuggers lack the direct support for collaboration [33]. Collaborative tools are beneficial only if the collaboration in the first place

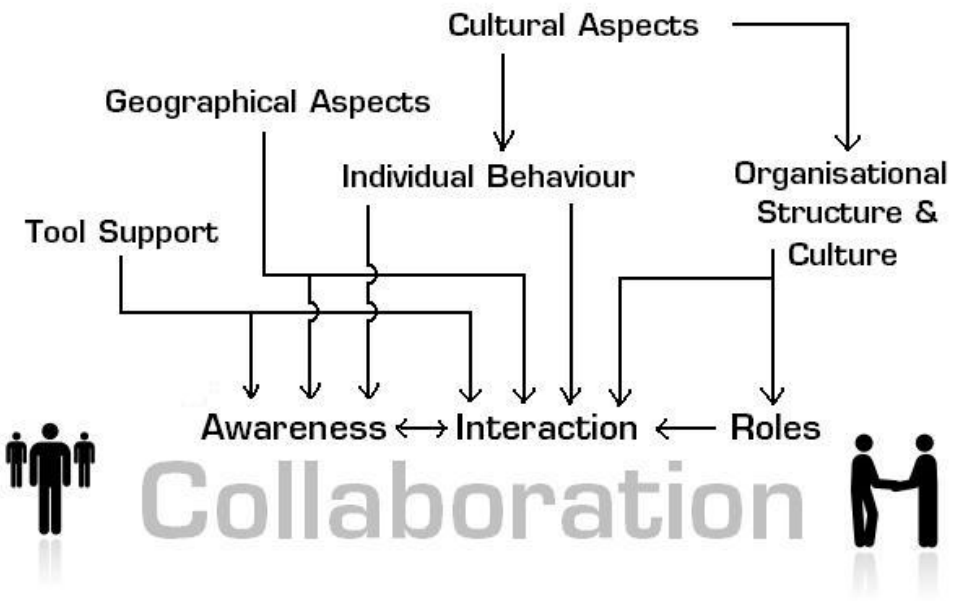


Figure 3.1: Influences on and restrictions for collaboration

has a firm social footing [45]. Good collaboration has many advantages, and bad collaboration has an even greater number of disadvantages. Good collaboration support should produce better outcomes through collective knowledge and wisdom. This requires that users have steady access to their own work and are aware of the work of others [114]. In the end, through supporting social interactions and saving time in coordination, well working cooperation raises productivity.

Improving collaboration has short-term and long-term benefits. Regarding the choice of tool support for communication and coordination of tasks and communication, there are two options. Either choose proper external tools for communication and coordination, which might be for example e-mail to communicate and a calendar for coordination, or choose a tool for the main task at hand with built-in collaborative support. An example for a tool with built-in collaborative support in software engineering is JAZZ (see Section 4.5). Either way, choosing the proper tool is highly beneficial for the project at hand. Independent of the choice for tools supporting collaboration, which is unquestionable but maybe short-term beneficially, is the proper use of these tools. Establishing rules for using the tools, and facilitating contact between team members also provides an improvement in collaboration for the project at hand as well as for following projects. If the team members know each other, communication is made easier to start with. Coworkers being aware of responsibilities and authorisations leads to less redundancies in work and less uncertainties concerning task allocation, thereby reducing potential conflicts.

Collaboration can be influenced negatively through wrong individual behaviour. An example can be found in collaborative writing. In Section 4.2 edit wars and the found solution for it in

Wikipedia are explained. Edit wars arise through users repeatedly undoing co-workers changes without communicating. We assume behaviour like this—in a professional environment—will happen much less often, but still occurs. Spinellis defines "commit wars" [101], the edit wars of software engineering, as a series of conflicting and mutually reversing, introduced by developers who disagree on how a particular element should be coded, sometimes starting with a hostile backout, which means to undo a commit without prior arrangement.

Within a company's team professional behaviour, although expected, naturally, is not always accomplished. Hence it is important to facilitate communication in a way that someone rather communicates than enforcing one's own opinion. In a professional environment, communication can be facilitated by providing thought-out tools and features. Thought-out means, that these tools and features are adequate for the intended situations and environments, and easy to use. An important factor, likely to be disregarded, are social aspects. On the social level, the company has to make sure its employees do not disregard others opinions. Establishing policies can help avoid this behaviour, and directs the use of the intended tools and features. For example, if e-mail is set to be a primary mean of communication, a rule to very regularly check the in-box facilitates communication and avoids frustration between co-workers.

3.2 Communication

Supporting the communication is vital to the collaborative process. Communication, as important as it is, can be regarded as a manual way of conflict resolution. As mentioned before, this includes the communication before a conflict arises in order to avoid the conflict at all. Good communication can not only reduce the amount of conflicts, but also very much improve the conflict resolution process.

Synchronous vs. Asynchronous Communication

Conversation is either *synchronous* or *asynchronous*. Synchronous communication is the concurrent participation in a communication. A chat or a face-to-face dialogue, a phone or a video conference are synchronous collaborative communications. Synchronous collaborative work includes a number of people working together on the same document in real time [114]. This type of collaboration usually is interactive, intensive and focused, and best supported by highly concurrent real-time synchronous access to information. When the mode of interaction is an asynchronous one, a person works relatively independent. A message in asynchronous communication must not be provided in real-time or at all. Sending an e-mail for example is considered asynchronous communication, for the message is sent by one person independent of time of receipt or answer.

Depending on the circumstances, synchronous or asynchronous communication is chosen. Important is, that both possibilities are given. Synchronous communication is usually preferred, because answers are given immediately. If answers are not required immediately, the person to contact is not available or not to be disturbed, asynchronous communication is preferable.

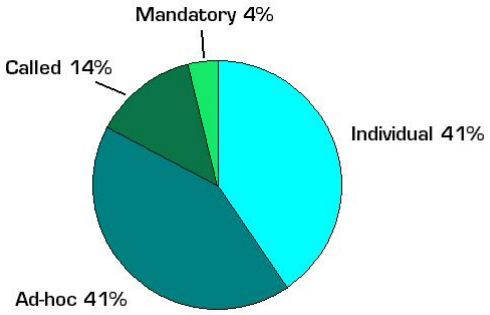


Figure 3.2: Relative time spent in the collaborative activities, redrawn from [90]

Spontaneous vs. Prearranged Communication

Communication can also be classified according to the preparation included. Robillard et al. [90] divide collaborative activities into four types: mandatory, called, ad-hoc and individual collaborative activities. Figure 3.2 shows how much these four activities are used in comparison to each other. Mandatory activities are the smallest part, individual and ad-hoc collaborative activities are used most.

Mandatory collaborative activities are formal meetings scheduled on a regular basis. They are usually planned long in advance and participants are required to attend.

Called collaborative activities occur when two or more team members decide to get together to do some technical work. Usually these meetings are not scheduled long in advance and only participating members will attend.

Ad-hoc collaborative activities occur when two or more team members work on the same subject at the same time and share on ad-hoc basis comments, or information on what they are doing. These activities usually are informal and sporadic, and a team member would interrupt his/her work to reply.

Individual activities occur when a team member is working on a task that is not shared at the same time by other team members and is then unlikely to interact on this subject with other team members.

Dividing the necessary communication into those groups and displaying the frequency of occurrence shows the need to support collaborative activities due to their numerical appearance. Thus the required features can be prioritized. Usually individual and ad-hoc communication are by far required most.

Because of these discoveries, the means for communication of choice to be made available for a team would first be for individual and ad-hoc collaborative activities. This means, if tool support is provided, for example instant messaging—synchronous and asynchronous—is preferable. Prioritising instant messaging does not decrease the need for other means of communication, for mandatory meetings video conferencing can be offered. The goal is to on one

hand provide the user with all means of communication required, that is for mandatory, called, ad-hoc, and individual activities. On the other hand, to prevent an overkill, prioritisation of these means of communication is required. The prioritisation takes place at the choice of means to communicate with and their accessibility, and by rules defining their use. For example, a team deciding to use e-mail for asynchronous communication and the phone for synchronous communication. Thereby the team prioritises means of communication for individual and ad-hoc communication, and these means are easy to access and to use. Called and mandatory collaborative activities, whether they are done face-to-face or using tool support, have to be arranged. The coordination of the called and mandatory activities is done in an ad-hoc or even individual fashion.

Making Arrangements for Communication

Empirical studies on large-scale SE projects have shown that coordination is one of the most difficult and pervasive problems [111]. Collaboration includes the coordination of activities and the coordination of communication.

There is spontaneous and prearranged collaboration. Prearranged collaboration can be a meeting, a simple conversation, or a task which has to be done in the group and therefore be scheduled. These tasks have to be arranged. An arrangement such as this consists of the process of arranging, and the storage of the information concerning the appointment to be looked up and remembered. An appointment can be agreed upon by different means, like the phone or e-mail. Additionally a calendar is needed to entry appointments. Electronic calendars come in handy, for they automate some processes like alignment of data and messages.

Calendar files—with the *.ics* file extension—can be used in different operating systems (Windows, Mac OS and Linux) and opened with different programs (Apple iCalendar, Mozilla Sunbird, Google Calendar and more) [44]. Calendar files can easily be sent by e-mail, and often e-mail clients include a calendar.

Users can subscribe to published calendars when using certain calendars. With a feature like that, the user can send out an invitation for a meeting to several people, e.g. a group of people occupied with a certain task. When provided with profiles of all co-workers, containing the necessary information and divided into groups, the selection of a certain group is all it needs to send out an appointment, which would be more time efficient than looking up every phone number and calling colleagues one by one. The answers would be carried out by the same system, providing an automatically created list of actual participants.

Means of Collaboration

In the following, we shortly discuss possibilities to communicate and their advantages and disadvantages in general and for our specific case.

An old-fashioned way of communication is to talk face-to-face. This is not always possible, especially in distributed software development, and brings other disadvantages. These conversations are not recorded and therefore impossible to reconstruct later on. Conversations over the phone allow people in different locations to converse, but offer no traceability. Using chat

for communication is widely used, for example in multiplayer games and on websites. Several applications offer chat as main feature, like ICQ, Skype, or the Windows Live Messenger (see Section 4.1). Other tools integrate chat as communication feature. Examples are Camel (Section 4.4) or Quilt (Section 4.2).

Most popular are e-mails for communication. E-mails are stored and therefore allow later reconstruction of conversations. Also, they can be sent to any number of recipients. There are two disadvantages. First, e-mail conversations are impersonal. Second, e-mail is an asynchronous way to communicate, hence an immediate answer is by no means guaranteed. Video conferences on the other hand provide synchronous communication, can be formal but are not impersonal. Also, video conferencing can properly replace regular meetings if a face-to-face meeting is not possible.

Regularly held meetings may be a necessity, but—for example in a conflict situation—a meeting for every conflict resolution would definitely take too much time and delay the conflict resolution process. Thus it is important to reduce the number of meetings to an essential number and solve other conflicts in a different and more efficient way. Especially when only certain persons are needed to solve a certain conflict, a called collaborative activity (see Robillard et al. [90]) beats a mandatory one by a mile, since it is more time efficient.

It may be concluded, that different means of communication prove to be useful under different circumstances, whereas a diversity of offered choices is necessary to ensure the coverage of these circumstances.

3.3 Methods of Collaboration

Selecting and using offered tools is only the first step to improve a teams collaboration. The team has to agree on methods on how to collaborate. That is, policies and rules have to be set in advance.

In the following we present some ways to coordinate collaboration. When a team tries to solve a problem, different opinions clash with each other, and therefore the process is a form of negotiation. “Negotiation is fundamental to all collaboration because agents need to agree in order to act together” [98]. Negotiation means two or more parties bring together their goals, share information, and search for mutually beneficial alternatives. Without supporting strategies, users might be in client to persuade others of their opinion instead of looking for an agreement and a mutual commitment [91].

The team can either set a code of practice for the whole conflict solving process, or just establish some policies to improve it. The authors of some of the following practices assure us of the successive benefits of their methods, but maybe in some cases these tight regulations hinder the process, for it is dependent on the situation after all.

Sidner [98] breaks down the negotiation process into a cycle of proposal and rejection/ acceptance. Chu-Carroll et al. [29, 30] capture the negotiation process as a cycle of recursive actions of propose, evaluate, and modify. These negotiation cycles are simple and may occur naturally. More detailed is Adelsons negotiation system [1].

Adelson [1] offers a negotiation system, which repeatedly asks the participating parties to partake in five stages. This cycle shall help to jointly build a solution by offering psychological support, and is composed by the following stages:

Problem statement. The parties describe the problem from their point of view.

Initial solution. Then solutions from the conflicting parties are proposed, with which the party offering the solution agrees.

Underlying needs and resources. Lists are constructed jointly, containing what is needed and which resources are available. This is figured out based on the problem statements. Each party rates the importance of the needs for their problem and how well the need is satisfied by the solution under consideration. Thus the negotiating parties can clarify their needs, rework or accept the solution.

Collaborative solution. The penultimate step is to jointly create a new solution, considering how the resources match the needs and thereby gain a collaborative solution.

Iterating. The parties examine how their identified needs exactly fit together with the arranged solution.

Each party can always retreat from their proposed solution to rebuild it. Needs and solutions are considered separately, which facilitates the discovery of successful solutions. The systematical matching and the explicitly made resources can clarify matches of needs and resources, that may not have been obvious at first. Also, with the detailed description of the needs one can quickly see if there is a possible joint solution at all, and if otherwise, avoid the effort.

Negotiation systems range from simple two-step cycles to specialised iterating activities. These can be applied in several different fields, on different occasions. Depending on the circumstances, they are more or less useful. Another possibility is that the used tool—or used tools—provide the users with sufficient information and means to communicate, and the willingness to find a mutual commitment exists, maybe through rules in the organisation concerning social roles, which relieves the team members of extended negotiation cycles, when it is not necessarily the case.

3.4 Time of Collaboration

The selection of means of communication and even of methods is also very much dependent on the time it is used within the development process. Collaboration in software engineering has multiple goals spanning the entire life cycle of development [112], and must therefore be facilitated every step of the way. The later the changes, adjustments, and corrections are made, the more work and trouble it brings.

For our research, we divide the life cycle in only three categories—before and when a conflict arises, and at any other point of time. In the event of a conflict, first there needs to be the notification of that conflict, and the means to support the resolution. But as already established, collaboration carried out well can accomplish more. If communication and coordination is working, it can anticipate conflicts to a high degree.

The first task of collaboration acted out well is the reduction of conflicts, and thereby the saving of a lot of resources.

3.5 Collaboration in Model-Driven Development

Models themselves provide support for collaboration, for they can be used as a common language between different kinds of experts. For example analyst and user [76], or system designer and system developers, may use models as drafts or blue prints [47]. These experts have different skills. A *non-technical* user may be presented with the relevant information in form of a model, therefore an approximate but immediate visual interpretation is facilitated [76].

Software engineering projects usually are a team effort [25]. The minimal tool support required for a model-driven development team is version control. By providing access to and management of versions from different persons, awareness and coordination of tasks is facilitated. Therefore, version control can be seen as support to collaboration. In model-driven engineering, like in other software engineering branches, version control support is essential.

CHAPTER 4

Analysing and Comparing Existing Systems and Ideas

There are a lot of different systems, we can deduce ideas from, which—modified and combined—are useful for the planned collaborative extension of the model versioning system AMOR. Additionally, we can learn from problems arising in comparable collaborative approaches, and studies concerning collaboration support in SE. With the focus still on collaboration, and especially communication, in the following pages we survey systems of interest.

This section does not constitute a complete list of every available idea that can be linked to our topic, which indeed would be excessive, but rather a selection from various fields. We intend to collect ideas, on how to integrate collaboration into model versioning systems. First, we consider successful communication tools, because communication is a large part of communication. Second, collaborative drawing and writing systems. Collaborative systems can be comprised of several tools. Third, we consider examples of collaborative use of frameworks for several tools. Then we survey tools intended to specifically draw models and approaches for collaborative software engineering. Finally, multi-player games provide an example for willingly used integrated collaboration.

4.1 Tools for Communication only

Numerous tools for communication are available. These can be divided through the way the communication is used, whether it is *typed* communication, *audio*, *visual*, or a *combination* of these possibilities. Since these tools are used for conversing only, when looking at their particular features and their popularity, and the thereby indicated usability, we have to keep in mind that the usability for our case is dependent on more, namely that more tasks have to be accomplished than communication only. Most of the tools picked in this section either stick out for their popularity or their high circulation.

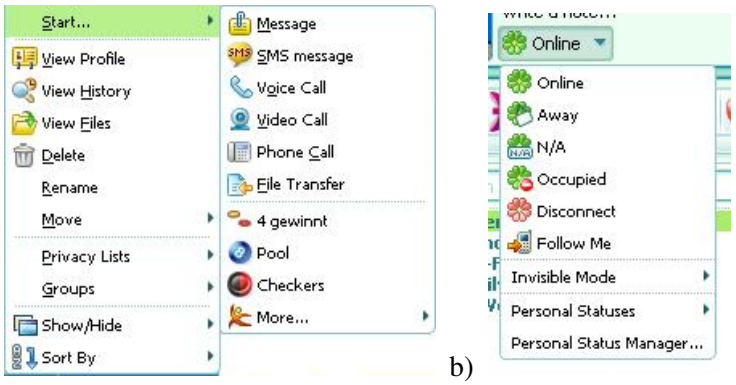
ICQ and Skype

Very popular tools freely available on the net are ICQ[1] and Skype[2], both offering textual, audio, and video communication. Also there are additional features like sending files or play games. ICQ and Skype offer very similar possibilities. ICQ often is preferred for chat, while Skype is very popular for Voice over IP (VoIP). Therefore we will especially look into these features—typed conversation via ICQ and audio (and visual) communication via Skype. ICQ has been around since 1996, and Skype since 2003. Both are constantly under ongoing development, with the newer version improving the former version.

A chat feature basically includes a list of participants, a window subdivided in a field to watch the conversation and one to type the text in plus a Send-button. The usability depends on the exact realisation of these items, ICQ and Skype provide experience on how to do that successfully. We are looking to learn from that experience.

ICQ

The chat function in ICQ is very easy to use. Once the software is downloaded and a profile is set up, the main part of the ICQ window contains the list of contacts. The size of the window is changeable, but by default setting relatively small, and justifiably so, allowing the user to keep it open while attending to other tasks. The list of contacts, which can be divided in groups, is accumulated by specifically looking for friends or by searching—or browsing through—the whole community for people of interest. The profiles of users mainly consist of optional entries and can be looked at by choosing a member of a contact list and a following right-click of the mouse on the name. With this right-click a short menu opens, offering possibilities not only to inspect the profile of this certain user, but also the communication history or a new chat window. The possibilities are shown in Figure 4.1a). A chat window can also be opened by a double-click on the name. Accessing features by certain mouse-clicks is easy to remember and fast, thus a very good way to do so.



a) b)

Figure 4.1: a) Within the contact list; b) The ICQ status menu

[1] http://www.icq.com
[2] http://skype.com

Chatting itself consists only of typing in the message and sending it by the use of a *Send* button, or additionally like in ICQ, with the return key. This simplicity makes chat to a valuable feature.

A drop-down menu in ICQ allows to set a status, the selectable statuses are shown in Figure 4.1b). With a status the user may announce whether he/she is busy at the moment or available at all. While we like the possibility to integrate a status into user profiles very much, the option invisibility should not be given in a work environment but reserved for private use.

ICQ offers different audio signals for nearly every event, whether it is the receipt of a message or a file, or the change of status of a contact list member.

Skype

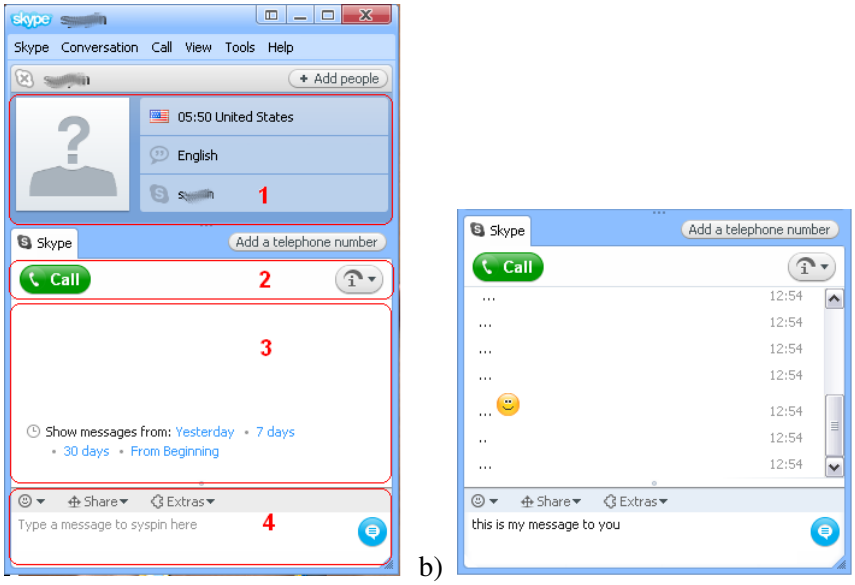
Skype offers quite congenerous possibilities as ICQ, ranging from audio signals to the possibility to set a status. This section contains a list of advantageous details found within Skype, that is accessing features through short-cuts, design of the user interface, and notifications and follow-ups.

Short-cuts. Instead of perambulating through an extensive main menu, there are short-cut methods available, providing a fast way to the most used features. In older versions of Skype, a double-click on a name in the contact list led immediately to a phone call, while in the current version—we looked into the newest available one in December 2009, which is Version 4.1 [99]—this action leads to the communication window shown in Figure 4.2a). In both cases this setting is configurable, but set in the mentioned manner by default. The priority of chat is equalised with the VoIP priority. The most used features are easily accessible in the communication window. Skype provides easy access to important features. At first glance the most important options are provided, which are next to chat and VoIP, contact information, and chat history.

The user interface. We abstain from describing the main window of the user interface, which essentially consists of menu options and contact list, and concentrate on windows used for the communication. Figure 4.2a), a screen shot of the communication window, is divided into four areas, and labeled with number 1 to 4.

The part labeled with the number one contains a summarisation of details on the chosen person, in our figure name and language. Photo, birthday, homepage address, and gender among others, would, if filled in, be shown here too. This is a short version of the user profile. For the screen shot we chose a member in our contact list residing in the United States, to point out that Skype shows the time zone the communication partner is in. This is useful to rise awareness of the time the communication partner is in, showing the availability for a synchronous communication.

In area 2 audio and video communication are offered. Area 3 is the part of the screen reserved for an ongoing chat session. If there is no ongoing chat conversation, access to the chat history is provided. The part for typing the message and the little blue *Send* button are labeled with the number 4.



a) b)

Figure 4.2: a) Skype window for communication; b) Skype's chat window

ICQ offers far more gimmicks that can be included into the chat session, indicating a slightly different user target group. Another detail worth mentioning for additional options in Skype is the possibility to send contacts. This is part of the "options" menu in the area labeled with 4 in Figure 4.2a). The recipient does not have to look for the contact himself, but is provided by a third person with a kind of business card linked with the profile of the person in question.

Next to the main menu—as usual a drop-down menu—there is a possibility to access options associated with the names in the contact list. This menu is opened with a right-click on the contact. These options are shown in Figure 4.3a). They include calling the contact, sending an instant message, a contact, or a file. Other options allow the user to view the contacts profile or rename him/her in ones' contact list. The last part of the menu offers the options to block the users messages, view the history or remove the user from ones' contact list.

The option "Share your Screen" allows the user to send a print of either his/her whole screen or a part of it to the user in question. When sending a part a movable rectangular window shows up indicating which part of the screen is selected. The options within are shown in figure 4.3b), and consist of either sending this detail of the desktop or cancel the action.

Notifications and follow-ups. Like in ICQ several audio signals are preset in Skype. When Skype is not the active window at a time and the user either gets a new message or a comment in an ongoing conversation, an audio signal makes him/her aware of it. Additionally in Skype, when you move your attention to the new message by activating the message window, the addressors name in front of the new messages is coloured orange for a moment instead of blue as usual, pointing out the new messages at first glance. Figure 4.2b) shows Skypes' chat window during a chat, with a time stamp for every message right of the text.

Though its place within the chat window of Skype changed over the different versions, a

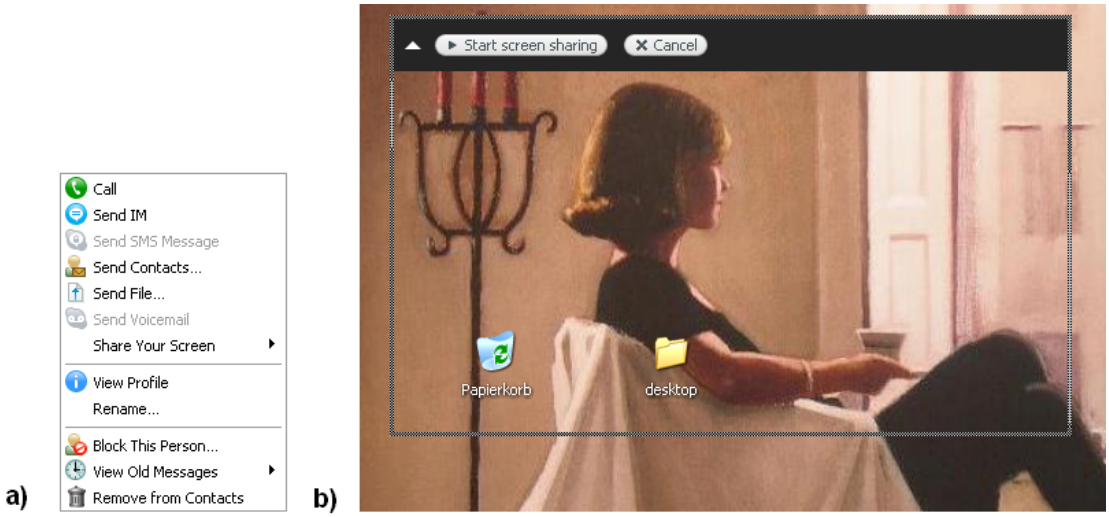


Figure 4.3: a) Skype menu on contact list; b) Sending a desktop print

small moving pencil indicates what the other person is doing: nothing at all, writing, or erasing something written. That definitely belongs to the category gimmicks, it is not necessary but likeable.

ICQ and Skype incorporate the basic necessities for successful online communication. Textual, audio and video means of communication are easy to use and paired with supportive features. Contacts are provided in lists, dividable into groups. The user can set a status concerning his/her availability for communication.

Windows Live Messenger

The Windows Live Messenger (WLM) is Microsofts instant messaging client. It was formerly known as MSN Messenger, then in the Windows XP environment as Windows Messenger. The features are similar to those offered by ICQ and Skype, which are a contact list divided in groups, instant messaging, emoticons, audio and video communication, audio notifications and so on. Even the graphical user interface is very similar.

There are two main differences between the WLM and the other previously introduced instant messaging systems. First, the WLM includes a whiteboard feature, the same as the one NetMeeting offers (see Section 4.3). Second, there is another connection to an additional application. The used e-mail client can be opened through the Messenger. Here it does not matter whether it is Windows' Outlook or a client from another vendor, like the Mozilla Thunderbird. This connection is automatically made. The e-mail address is provided because it is the name for logging into the Messenger, the default e-mail client is known.

Very easily can, if desired, additional features be incorporated into communication applications. Skype for example offers to send a screen shot to a contact. The WLM expands features like textual, audio, and video communication, as included in ICQ and Skype, by synchronous

use of a drawing canvas, thereby adding another way to communicate. Depending on a single persons or a groups needs, the tool to communicate with is chosen based on the included features.

Video Viewer

In 1992, Ropa et al. [92] conducted a case study concerning the users' attitudes utilising video for computer-aided communication only. They attempted to use the results of this case study for the further development of the communication tool called *Video Viewer*.

Experiment. Three teams participated in the study, using three computers and several monitors as private and shared desktops. The users had to perform several tasks for a week, like writing a document collaboratively while sitting in different rooms using for communication video only. Video communication was provided by a camera for face-to-face communication and one for broadcasting images and documents as well as a window for recorded video input. Users had to manually operate all video sources in an unstructured environment, for the conductors to design an easy to use supporting structure by analysing how users handle raw materials in the collaborative situation. Since the study was conducted at usual work time, the participants had possibilities to really talk face to face on different occasions. Communication outside the video environment was allowed to avoid a "laboratory effect". The results where accumulated by interviews, observations, and a logging system tracking mouse activities.

Two people looking at the same video at the same time can additionally have a conversation using an audio channel, while listening to the audio of the video channel, i.e. two audio sources. But both parties have no other visual input, nor related to the ongoing conversation itself.

Results. The results of the study give a good insight into users behaviour and their preferences while working with video as their main communication source. These, next to the discovered concentration issues, are listed consequently.

- The users mostly relied on audio communication to accomplish the different tasks. This may have been due to the fact that the participants knew each other and could talk face-to-face in breaks, while in real distributed software development users may prefer to see each other rather on video than not at all. Also the cameras position may have led to a difficulty in establishing eye-contact. We mentioned a possible laboratory effect before, which could—more or less—occur in any experiment. Sometimes the reasons for certain behaviours may not be explained or may be influenced by details of the way the experiment is conducted. Nevertheless those results show a strong tendency and may therefore be regarded as significant anyway.

- The importance of seeing each other changed with the specific reasons for communication. It was favoured when the focus was on discussion or for checking the communication status.

- The real world social protocols were transferred into the video environment. For example giving an author precedence in editing or asking permission was the same as without the communication support.
- The cameras for viewing objects and documents proved valuable for incorperating elements not in the video environment.
- The system was used for social interactions like sharing music too.

When working with the Video Viewer, the originators of the tool intended the users to make use of one video channel at a time—still being able to change the channel easily—and the possibility of a voice channel existing across all other channels. This is because the case study indicated that people usually can only visually center on one item, while they can concentrate on different audio input simultaneously.

The study of Ropa et al. [92] shows the diminished capacity of humans to take in too much input at a time. It was found, that users can concentrate visually on one item respectively, while attention to audio input can be divided. Therefore concurrent input of communication and other collaborative activities has to be limited. Also, depending on the relationships of the co-workers different ways to communicate are preferred. Communication additionally takes place outside the planned environment. In the preliminary case study for the creation of Video Viewer, projects were discussed face-to-face in breaks. But then the planned environment is used for privat interaction too.

4.2 Cooperative Drawing and Writing

Ignat et al. [64] assert that "drawing is a primary activity in many design domains, such as architectural and product design, ...". We also "draw" models, be it for discussion, documentation, or model-driven development. In this section we introduce collaborative tools for writing and for drawing.

Quilt

When writing collaboratively, additional information turns up. This information may include who is doing what and when, which must be communicated to achieve a conjointly outcome. Furthermore, the activities must be coordinated. To support the distribution of implicit information and help coordinate activities among collaborative writers, Robert Fish et al. [45] built Quilt, a collaborative writing tool. This was in the 1980's. Even then, 65 percent of scientific articles were jointly written [85].

For communication, Quilt provides annotation, notification, and messaging features. Quilt was designed for collaborative writing in research projects. Only asynchronous communication is possible. The collaborative writing process is supported, especially revision of other people's work. The coordination of activities is important to avoid redundancies, and make sure, no part is omitted. This is accomplished by users sending notifications of ones state of work, and records

are kept on current activities. The kept history provides easy access to other users' activities and allows side-by-side comparison of different versions of a document. Consistency is maintained by warnings in case of concurrent changes to a certain part, and potential conflicts have to be solved manually.

A permission hierarchy and extensible sets of social roles regulate usage privileges on documents and responsibilities. In addition to the base document, voice and text documents and suggestions for revisions can be made, those are incorporated as hypertext nodes. The comments can be private, i.e. visible to the creator only, or public for anyone to see. Also directed messages can be sent, to an individual or a group, depending on the permission hierarchy. Comments and messages are saved in an activity log. Entries in this log can be generated by the user or by the tool.

Quilt is a tool providing asynchronous communication only, and is very much tailored to writing in research projects. Social aspects are regarded by introducing *social roles*, the separation of users into groups according to their authorisations. Coordination of activities is supported. Nevertheless, discussion of the content not so much.

Draw-Together

Draw-Together is a collaborative, object-based graphical editing tool. The main goal of the authors [64] is to maintain consistency when complex operations are involved. Those could be grouping of objects or working on layers. The consistency maintenance is accomplished by an operation serialisation algorithm based on the reordering of nodes in a graph. The nodes represent operations, the edges ordering constraints between operations. Two concurrent operations result in a conflict, when the same property of a common target object is modified. Conflict handling is customised, for the users can specify types of conflicts and policies for their resolution, which can be modified and are stored in a separate document. These specifications include whether the operations are considered to be conflicting.

Figure 4.4 displays the GUI. The various buttons included in the toolbars allow the user to draw rectangles, lines, ellipses, write text, and others. The right side of the GUI offers the options when working with multi-pages and layers, and lists the participating users. Both can be set visible or invisible.

Draw-Together combines options for drawings in a layered fashion as other sophisticated drawing tools like Adobe Photoshop[3] or Gimp[4] do, with the possibility to do so collaboratively. The focus is on defining and handling conflicts occurring in the collaborative process. However, there is no other way of collaboration than drawing. That is, no communication possibility, neither textual nor audio is given. Coordination of work is not facilitated in order to avoid potential conflicts.

Collaborative Environments on the Web

In the World Wide Web we can find numerous examples of collaboration. Here, users work together for various reasons. Since the intentions are different, the way of collaboration is different

[3] http://www.adobe.com/products/photoshop/photoshop/

[4] http://www.gimp.org/

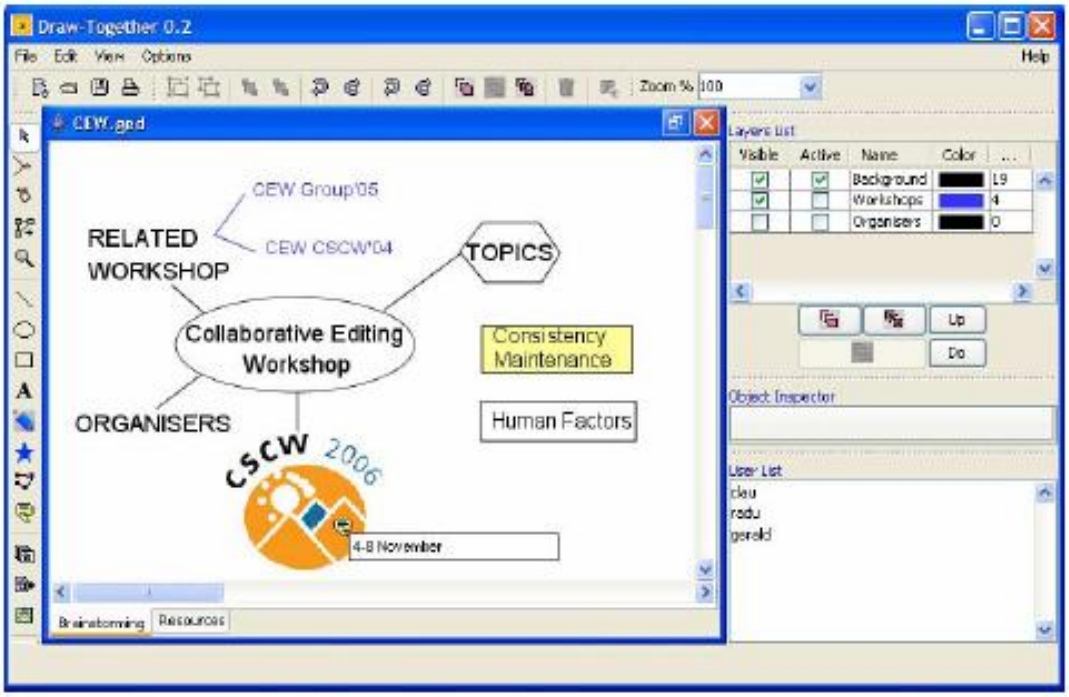


Figure 4.4: Draw-Together GUI

too. In this section, some of these environments are listed and discussed.

An example of groupware is Sourceforge [100], which is an open source software development web site additionally offering a blog and a forum. Registered users manage source code and share it, or enhance shared software, for instance by fixing a bug. Sourceforge uses implicit branching of projects. Explicit branching is used to organize the shared information in Picbreeder [96]. Picbreeder allows users to collaboratively evolve images.

An early collaborative system for music and a precedent for the Picbreeder-approach is the Faust system [67, 68]. Musicians could collectively produce compositions. The users could branch from a previously saved song, edit it, and save it again for other users to change. Unlike in Picbreeder, where users require no certain knowledge, here a basically musical knowledge to directly manipulate the notes was necessary.

Another collaborative environment on the web where some sort of knowledge is needed is Wikipedia[5]. In Nupedia, also a free collaborative encyclopedia and Wikipedias predecessor, certified expert knowledge was needed. It had a rather moderate success, mostly because of the rigorous review processes. Nupedia was shut down in 2003, 3 years after it was created.

[5] http://www.wikipedia.org

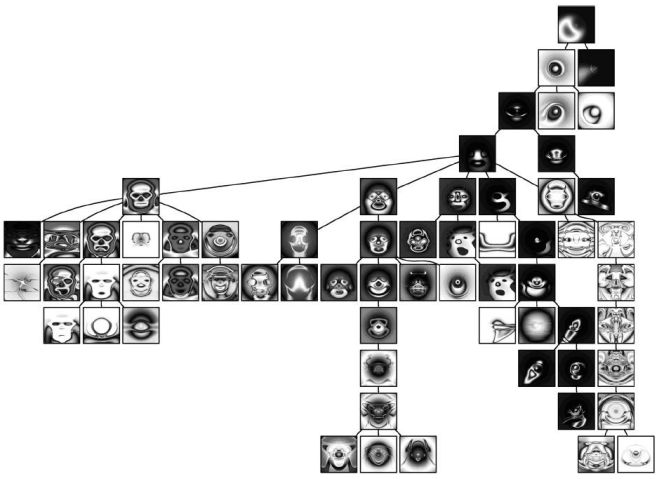


Figure 4.5: A tree of life in Picbreeder [96]

Picbreeder

Picbreeder is an online service for evolving pictures collaboratively [96]. Users can change images from scratch or continue to evolve already published images. Picbreeder is an approach to *Collaborative Interactive Evolution* (CIE). The *Interactive Evolutionary Computation* (IEC) client program provides a copy of the selected image, which can be evolved through the IEC process and then published again. The user does not have to be familiar with the process, nor have any other special knowledge.

Thereby a large catalog of user-created content is generated. There are no conflicts between users, since everyone can publish and change images, and all of them will be saved and are available for other users. The evolved and published pictures and their connection to each other result in a tree varying in breadth and depth. Figure 4.5 shows such a *tree of life*. One root image has been evolved by 13 different users. Any image can be branched multiple times, and all the images are kept indefinitely. Figure 4.6 shows how the branching works. Each set of generations is saved in a series. The last individual selected of the series is called its representative, which is what a user browsing the site would see.

Picbreeder provides a simple interface for browsing the images. Users can browse a tree by clicking on the parent or children representatives, and thus trace back the evolution up to the initial series. Picbreeder is all about having as many individuals possible participate in the project and trying to avoid user fatigue. Still, there are some—for us—useful thoughts included we want to survey. Users can rate images of other users, this encourages participation. Rating the work of others will in our case be counterproductive, since we want to support cooperation and not competition. Images in Picbreeder can be tagged. Necessary in this case, it could be helpful for us, because it enhances the search for usable information. The tags in Picbreeder are associated during publishing, the system will suggest frequently used ones in the process.

In versioning of SE artefacts the goal is to create a single agreed outcome by merging dif-

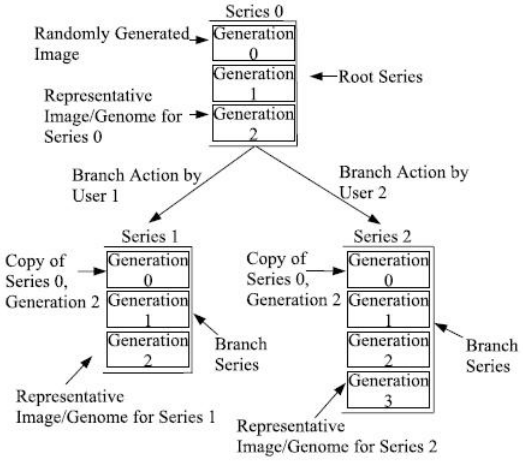


Figure 4.6: Branching in Picbreeder [96]

ferent suggestions of different users. In Picbreeder it is quite the opposite. The contribution of every single user is saved and specified, and therefore recognisable, as this users contribution. This recognition of individual work is important to motivate the participants. In our case, the traceability of the author is even more important, for another reason of course. If there is a conflict, or any other reason which makes it necessary to trace back a fragments origin, it is vital to know whom you have to talk to to gain the needed information.

Wikipedia

Wikipedia[6], the online encyclopedia, certainly needs no introduction. Since Wikipedia is well-known and popular, we will save our breath and instead of a long explanation begin with interesting details concerning the collaborative processes within Wikipedia.

In Wikipedia, articles can be labeled with multiple tags. One possibility is the “controversial tag”. Pages marked in this manner are added to the “List of controversial topics”. Every page (main page, every article) potentially has a talk page. On top of the page is a tab labeled “discussion” where one can find or create such a talk page to discuss the content. So for each article there is a connected but not visually integrated discussion. Additionally there is the Wikipedia “Village Pump” [113] for discussions not related to an article.

Wikipedia is about asynchronous collaboration. With its immense growth, the potential for conflicts and the need for coordination increases just as fast.

Reverting and Edit Wars. In Wikipedia, there is a possibility to restore an entry to its former version, to “revert” the article. This is very useful to fight vandalism, and it can be used to back up one side of a conflict. If there are conflicts between two users, where each one

[6]http://www.wikipedia.org/

expresses his/her entry to be justified by removing the new content and restoring their own entry (fittingly called "edit war" on [113]), this would keenly increase the amount of article reverts, without being useful or correct. The solution was the "three revert rule" policy[7], which states that no user should revert the same page more than three times in 24 hours. The percentage of entries being reverted more than once immediately was decreased, which concedes a point to the wikipedia team for installing the new rule.

The advice we try to give in this thesis is not intended for a system where the users are even in client to fight edit wars. However, in every office it can happen that someone just assumes to be right without considering the other persons motivation or the possibility to simply be wrong. Spinellis [101] calls these edit wars in SE *commit wars*. This individual behaviour is one of the social aspects usually disregarded, and is attended to only if it causes problems. These problems partly occur because different motivations are not taken into account and discussion is not facilitated sufficiently.

In Wikipedia the projects—writing articles—is outsourced to volunteers, still collaboration is indispensable and valuable.

4.3 Frameworks for Collaboration

In case several different existing applications or tools are needed for a task, collaboration can be accomplished or enhanced by building a framework around these tools, thereby avoiding building up a new complete system from scratch. This section describes three examples of frameworks built to collaboratively use several applications. COVE allows specialists from various fields to collaborate more efficiently in oceanographic research. IMPROMPTU combines word processing and other applications like browsers with desktop sharing and user profiles. Also Microsoft offers services to use Windows applications collaboratively.

Collaborative Ocean Visualization Environment

Oceanography, admittedly is a completely different field than what we are working on. Here too exists the need for an interactive tool supporting collaboration in larger multidisciplinary groups. There are several tools supporting the different disciplines needed in oceanography—for instance geology, biology, or mechanical engineering. Therefore experts working together interdisciplinary need to move between different systems to collaborate. To make this work more efficient, Grochow [54] is working on a tool for multidisciplinary science collaboration. Collaborative Ocean Visualization Environment (COVE) provides an interactive workplace for individuals and groups, a common visual environment. We leave out the field evaluations, geothermal and other aspects of the project.

The common data space and extensions for communication enables the scientists to focus on scientific questions rather than on laying out networks and coordinating teams. The major advantage of this system is that several alternative plans can be considered, which would have been a time-consuming task before, thereby raising productivity.

[7] http://en.wikipedia.org/wiki/Wikipedia:3RR#The_three-revert_rule

Experts in different fields using different tools have to work together not only in oceanography but in other fields as well. In software engineering these experts may be designer, programmer, tester, and user, who all use different tools but need to communicate on a common level. The requirements given from the user to the designer are different from those the designer forwards to the programmer. A single tool may not be sufficient to provide support for each of these groups and still provide a common ground for collaboration. A framework built around these tools, linking them together, may ensure well working collaboration as well as proper support for the tasks of the different experts.

IMPROMPTU

The interaction framework IMPROMPTU (IMPROving MDE's Potential to support Tasks that are genUine), realised by Biehl et al. [13], offers the possibility to show and share off-the-shelf applications for focused problem solving, and to place information on shared displays for discussion. IMPROMPTU offers a *lightweight interface*, which takes up little room compared to the applications used within.

IMPROMPTU is a way to organise collaboration in multiple display environments (MDEs). A MDE consists of co-located stations, like users with laptops, and shared devices, like additional shared displays, linked together to form a virtual workspace. As shared device an interactive whiteboard may be a good choice. With the introduced framework not only information supported by custom built applications can be shared, but also existing applications can be used. Some of these existing applications within the IMPROMPTU interface are shown in Figure 4.7.

The interface includes three elements: the Collaboration Control, the Collaborator Bar, and a shared Screen Dock.

Collaboration Control. The Collaboration Control allows the user to regulate whether an application is shared or shown to the other users, the three sharing states are shown in Figure 4.8. "Do not show or share" is the default setting. It is displayed on the top-level application window, marked with (C) in Figure 4.7. This feature provides quick access to the functionality, and with the window frames coloured according to their sharing state, one is always aware of it.

Collaborator Bar. The Collaborator Bar is on the left side of the screen, showing a representation of participating users. Item (A) in Figure 4.7 marks the Bar, with the "drawer" of the user at the top opened. The drawer includes thumbnails of the applications shown or shared by a certain user, which can be dragged on to one's desktop to replicate the window—not every shared window pops up at your desktop. Also, the movement of other the users cursor is shown within replicated windows.

Shared Screen Dock. The shared Screen Dock, marked with (B) in Figure 4.7, is the minimised window containing thumbnails of the applications on the shared screen. When opened, any user can adjust the position of the windows in it, or redirect his/her mouse/keyboard actions to the shared screen. For any shared device included, there is a Screen Dock. The shared screen

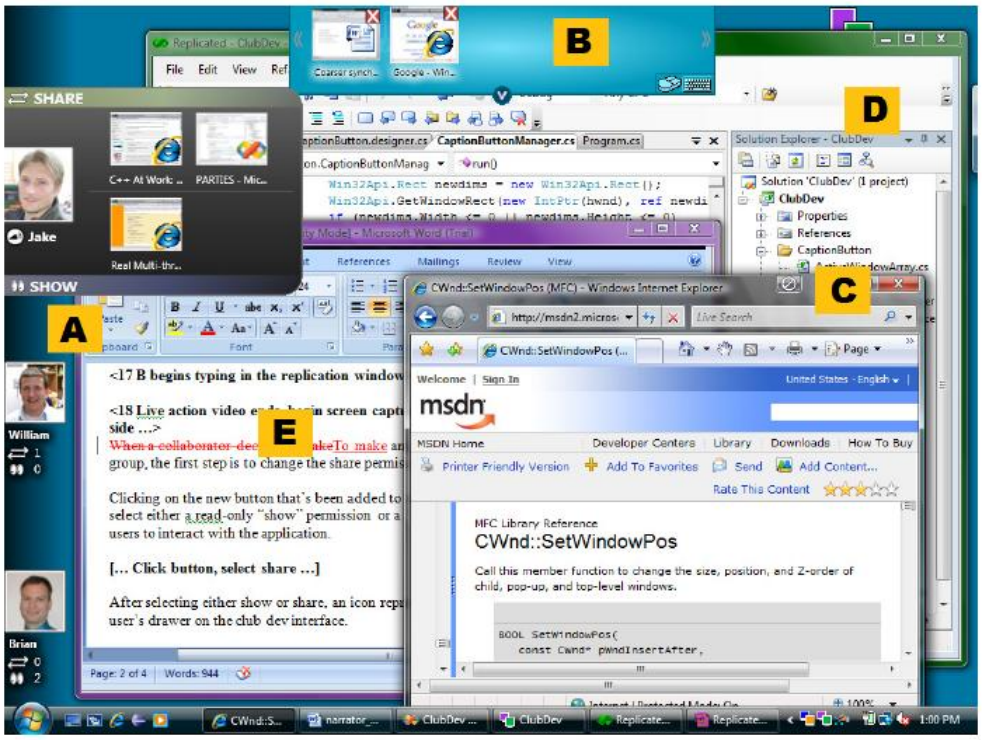


Figure 4.7: IMPROMTU [13]

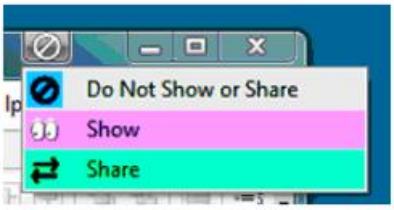


Figure 4.8: Collaboration Control in IMPROMPTU [13]

can include replicated windows from different users, those can be, as on the personal screen, be dragged on to it.

The creators of IMPROMPTU [13] conducted studies concerning the usability, and found that it was highly accepted and that almost every feature of the framework was utilised.

Collaboration in the Microsoft Windows Office Environment

With Windows, Microsoft created a virtual office workplace, by providing applications for any purpose. With the OS as basis, Microsoft Word is the application for word processing, Internet Explorer is the internet browser, Microsoft Outlook the e-mail client, and so on. Online Communication and Collaboration facilities are provided by the Windows Live Messenger and the Microsoft Office Live Meeting. The former one is described in Section 4.1. The Windows Office environment is the example we use, of course other OS offer similar frameworks.

NetMeeting. NetMeeting is a collaborative tool from Microsoft, included in Windows XP and former versions. It is the predecessor of Microsoft Office Live Meeting. A conference is started and ended by one person called the facilitator. This facilitator selects the features used within a conference. Selectable are the so called conference applications, which are voice over IP (VoIP), video conferencing, chat, transfer of data, and remote desktop sharing, respectively, which applications and files are released.

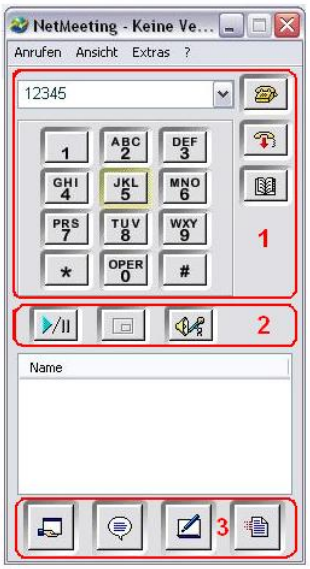
Figure 4.9a) displays the graphical user interface. The marked area labeled with the number one enables the main feature, VoIP. The buttons on the right side are shortcuts to place and end a call, and provide access to the address book. Area 2 are the video options, including start a video or picture-in-picture display. The third marked area includes four buttons. With the first one, running applications can be chosen to be released, the second one starts a chat, the third one the whiteboard option, and the last starts a file transfer. There is a do-not-disturb option. When activated, incoming calls are not announced until deactivated again.

Other Windows applications may be used with NetMeeting, for instance a version of Microsoft Paint is the shared whiteboard, see Figure 4.9b).

NetMeeting and the MSN, a collection of websites and services including the MSN Messenger, is now replaced by Windows Live, essentially a set of web services and client-side applications.

Microsoft Office Live Meeting. NetMeeting has rather restricted features compared to its successor. Microsoft acquired the company PlaceWare in 2003, whose core product was refined and renamed to the Microsoft Office Live Meeting. Subsequently the further development of NetMeeting was stopped and NetMeeting replaced. While the former one only utilised Microsoft Paint and Outlook next to the in-built features like chat, Live Meeting utilises several Windows applications. It is strongly connected to Outlook, meetings are joined directly from the e-mail invitation, or through the Outlook calendar, in which they were scheduled before, with a “Join the meeting” link [79].

There is a web-based client and a windows-based client, i.e. the client-side application. Content is shared by importing it to the Live Meeting client, for example PowerPoint files.

a) 

b) 

Figure 4.9: a) The NetMeeting GUI; b) NetMeeting's whiteboard

Another new feature is the Microsoft Roundtable, which provides a 360 degree camera view.

4.4 Drawing Models

In model-driven engineering models replace source code as the primary development artefacts, and these models have to be drawn. Whether models are created for MDE, or used as sketches or blue prints, the drawing of the models requires tool support.

There are numerous tools supporting drawing of models in UML and other modelling languages. Microsofts Visio[8] incorporates UML model elements among others. Visio seems to be treated as the poor cousin in the Microsoft Office environment, for its further development does not follow up with the advances in the main Office applications. Well-thought-out tools for cooperatively drawing models are Ideogramic UML and Camel. Ideogramic UML focuses on gesture-based drawing, especially to support learning modelling languages. Camel is an Eclipse plug-in. And both, Ideogramic UML and Camel, support free-hand drawing. Other tools for drawing models, especially UML models, include SmartDraw[9], several Eclipse plug-ins next to Camel, and the online-drawing-tool Gliffy[10], to name just very few. Other tools, supporting code-generation from models are discussed in the next section.

[8] http://office.microsoft.com/en-us/visio
[9] http://www.smartdraw.com/specials/umldesign.asp
[10] http://www.gliffy.com

Ideogramic UML

Ideogramic UML [55, 61] is a tool for gesture-based collaborative modelling with UML especially created to collaboratively teach and learn modelling. A variety of input devices like graphics tablets, ordinary desktop PCs or electronic whiteboards may be used to create models in free-hand mode. It is especially effective on large-scale devices like electronic whiteboards, which are naturally used for collaborative work.

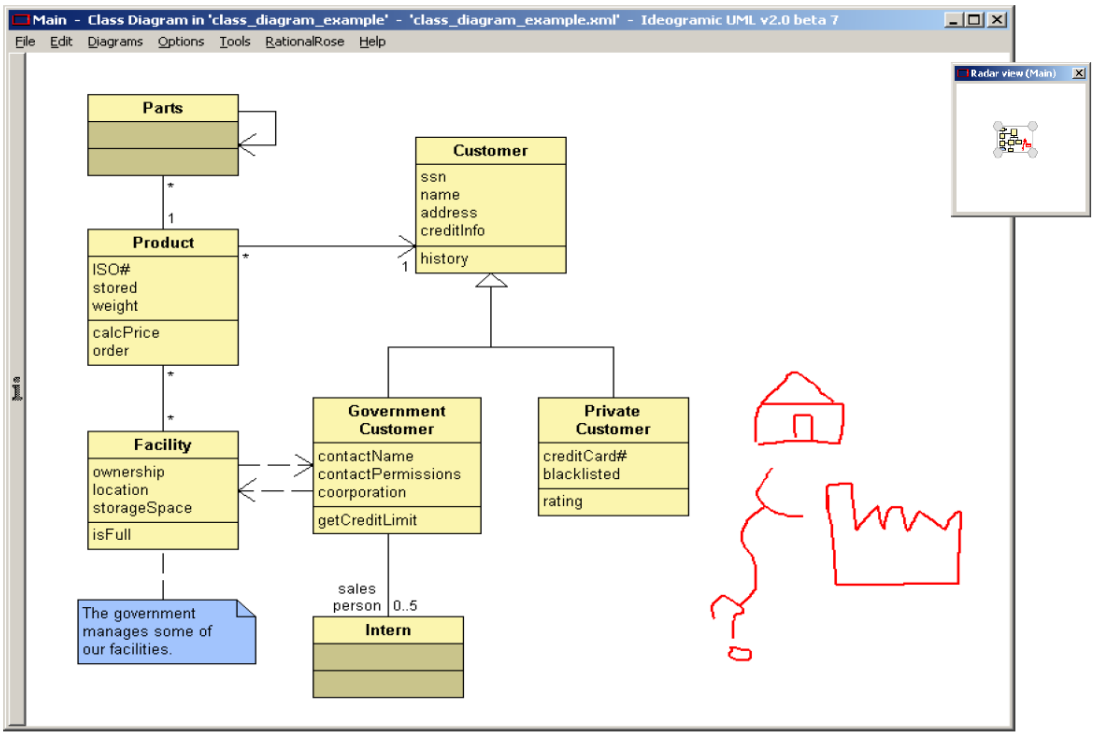


Figure 4.10: The Ideogramic UML user interface

The Ideogramic UML user interface (see Figure 4.10) is a simple and plain white surface, on which users can draw formal and informal model elements. Formal elements are gestures drawn and recognised, which are then translated into UML objects. Very simple gestures are sufficient, as shown in Figure 4.11. Informal information is displayed as it is drawn, as sketches and notes. Figure 4.10 contains sketches coloured in red, and in the upper right corner is the radar view, which allows the user to zoom in and out.

Ideogramic UML completely forgoes toolbars or pull-down menus. Pressing and holding the pen opens context-sensitive, hierarchical pie menus, displaying relevant options only. A pie menu is shown in Figure 4.12. Drawing a short gesture in the direction of a command invokes the command.

In studies concerning the usability, the authors found that the basic gestures are easy to draw and learning UML is facilitated strongly. Users found the free-hand mode very important for

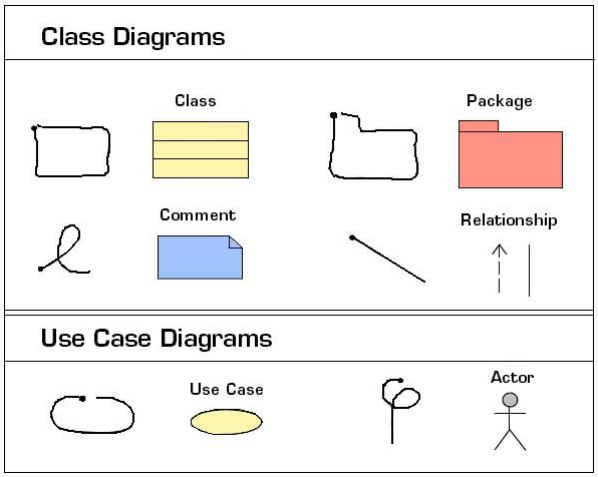


Figure 4.11: Examples of the incremental recognition and transformation of gestures [55]

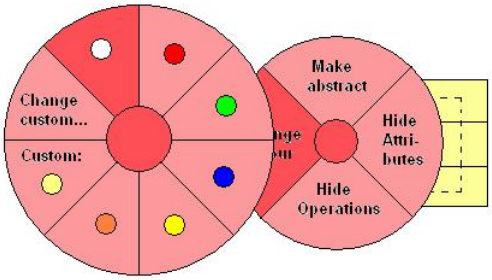


Figure 4.12: Context-sensitive, hierarchical pie menus [55]

UML modelling, especially for novice users and because incomplete elements are allowed.

Camel

Camel is an Eclipse plug-in, a tool for supporting virtual collaborative software design meetings. It consists of a well-arranged client interface and a server backend. The main part of the client interface, as shown in Figure 4.13, is the posterboard area, where users can simultaneously sketch and discuss multiple diagrams on a shared screen, either UML diagrams with the supporting drawing tools or free-hand drawings on the so called whiteboards. There is no intent to replace video conferencing or VoIP phones, so there are no audio or video features.

For a collaborative design meeting a session is created, which may be recorded and can be reviewed after the session is closed. Users are invited to sessions, when they join they get a colour assigned. Every change made on a whiteboard is drawn in the colour of the creator, other users are aware of who did what constantly. Camel provides the participants of a session to collaboratively make changes on a large canvas, while other screen-sharing tools only allow to

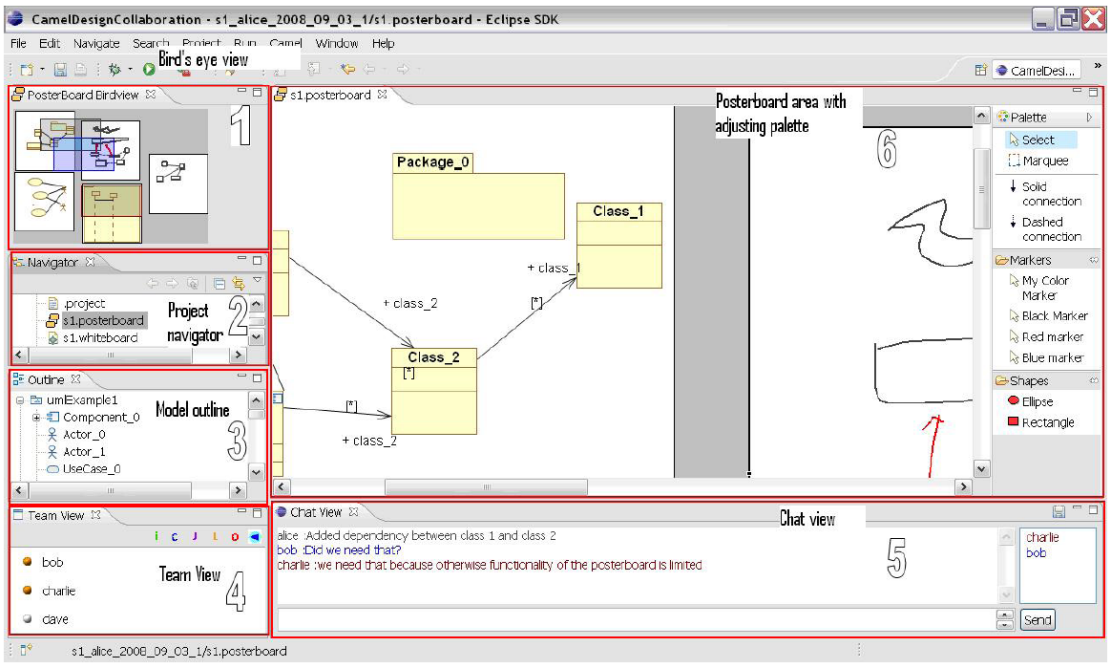


Figure 4.13: The Camel client interface

watch. Of course there are restrictions. Every session has a facilitator, who is the only person allowed to change the UML diagrams while everybody can—simultanously—draw free-style on the whiteboards or use the chat tool. The privileges of the facilitator can be moved to another person at any time of the session.

Figure 4.13 shows the client interface with each key part in an numbered box, therefore dividing the picture into six boxes. Box one is the *Poster Board Overview*, where users can see the whole poster board, the part they currently look at as well as the part the other participants in the session look at. This is presented in couloured frames, where the colours match the users colours. One can easily move in the *Overview* window or in the *Poster Board* itself. An additional feature concerning the birdview window is that the facilitator can send a signal to specific users to get their attention for a certain detail on the posterboard.

Next to the *Navigation box* and the *Model outline* (box 2 and 3) there is a window *Team Overview* (box 4) displaying invited and participating members of the session. The chat tool (box 5) works as follows: simply type in the text and click the send button. A nice accessory is a little window in the right corner of the chat tool, consisting of an enumeration of the participants in their assigned colours. Bob in blue and Charlie in red are aware of who else is participating, and which colour is assigned to them at first glance.

The main part of the GUI is the Posterboard (box 6) where numerous boards can be added to draw different kinds of models or free-hand. The supported UML diagrams are Class, Sequence, State Machine, Activity, and Use Case Diagrams, and the posterboard provides the necessary drawing elements for the different types of diagrams on its right side. Next to the offered geo-

metric shapes, a "pencil" options allows to draw freehand. Users can also zoom in and out in the posterboard. If required, the whiteboard can be exported as an image file in JPEG format. After any session the users can review it, step by step with the Playback feature by simply clicking a specific step or browse through the session with accommodating little arrow keys.

Since the possibility to automatically convert sketches into UML models is a productivity enhancement feature and not critical for the primary goal—design meetings—the authors decided not to include this feature, but may include it in the continuing development of Camel.

SLiM

SLiM (Synchronous Lightweight Modeling)[11] [106] is a lightweight environment for synchronous collaborative modelling. Independent of software or hardware equipment, a modern web browser only is needed to operate SLiM, and no software or plug-ins have to be installed. SLiM is being developed at the University of Mannheim. The main goal of SLiM is to support informal collaboration—especially ad-hoc collaboration—for distributed teams with heterogeneous system environments.

Figure 4.14 shows the SLiM user interface. A collaborative session at the SLiM website is started by naming and entering a virtual room specifying the session. The number of the virtual room is passed on to team members, which then can join the session.

On the left side of the drawing canvas is a list of shapes, UML model elements. Additionally, there is a highlighter tool, allowing the user to highlight specific areas within the canvas. In the part on the right side of the canvas the user can name the drawn model elements, and according to their type, for example add attributes and operations (cf. Figure 4.14). UML diagrams can be exported using XMI and then imported by other XMI 2.1 compliant modelling tools to ensure interoperability.

The communication feature takes a permanent place at the bottom of the application. Next to a list of participants, there is a chat window of the usual sort. It is composed of a dialog window, a line to type in text, and a *Send* button. The difference of this chat window to similar approaches like Camel (see Section 4.4) is that the chat window not only contains the users messages written to each other, but a description of the steps taken, for example requested and received information. There is a notification service in place to enable implicit information exchange between the attendees, thereby allowing synchronous collaboration. Locally executed user actions are published to all other users to synchronise local states.

4.5 Collaborative Software Engineering

As discussed in Section 3.1, collaboration is a vital part of any current software development project. Studies conducted in SE education provide us with valuable insight into social aspects in SE. We also introduce version control tools supporting models, for optimistic version control is required for collaborative software development, and SE tools with built-in collaborative features. Most advanced by far in the latter section is JAZZ.

[11] http://slim.uni-mannheim.de

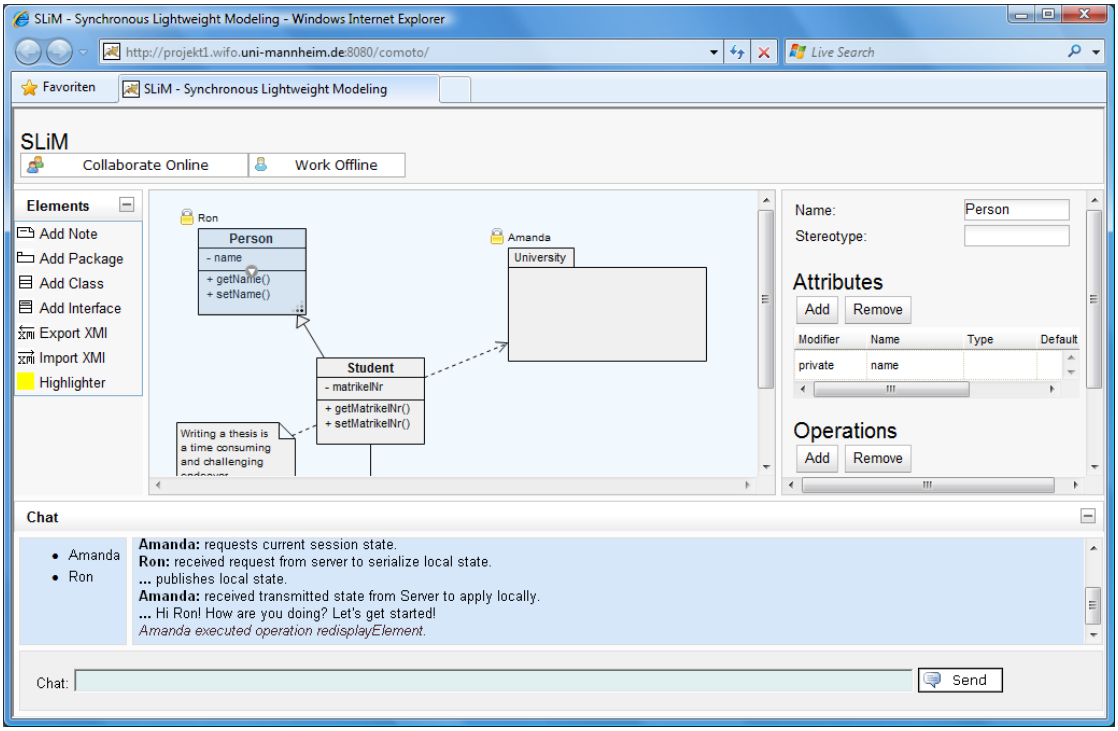


Figure 4.14: SLiM

Teaching and Learning of Software Engineering

Education provides a rich setting in which collaboration and communication co-exist in a number of forms [55]. Courses including testing of techniques or tools by students, potentially after the students designed the techniques or tools by themselves, provide pedagogical value and value for research, either for researchers in the academic environment and for industrial practice. This section contains a brief discussion of differences of studies conducted with students as subjects compared to studies in industrial setting. Next we look into studies concerning software engineering in education.

Student Environments vs. the Real World

Empirical studies concerning practicability of new processes, methods, techniques, and tools in industrial settings require plenty of time, effort, and resources. Therefore preliminary studies—called pilot studies—often are conducted using students as subjects [23]. Other studies using students as subjects, for example [43], are not conducted to later be continued in industrial practice, but still offer value for similar projects outside educational research. In any case, the results of these studies can not be transferred from educational environments to industrial practice one-to-one, for circumstances and primary goals differ in these environments. Nevertheless, these studies are unquestionably beneficial, for academia and industry. Carver et al. [23] identify four

primary actors for these pilot studies: researchers, instructors, students, and industry. Depending on the different goals, different costs and benefits arise for these actors.

Since the primary goal of students—learn and/or complete the course—and the environmental preconditions differ from those of studies conducted in an industrial setting, the results differ too. Through pilot studies using students as test subjects, preliminary insight for further development can be gained for industrial use of the processes, methods, techniques, and tools in question. These studies can not replace real-world tests.

Experiences in Collaborative SE Education

Global software development is more and more common, specialists in different areas have to communicate and coordinate their activities over distance. Therefore new challenges arise, which there is little training offered for in education. Students participating in the following study later received positive feedback in job interviews about their participation in the course. Favela et al. [43] conducted their studies concerning the matter of collaborative software engineering at an educational level, analysing aspects of distributed software engineering.

Experiment. Several project-oriented courses were held, using self-made and existing tools to collaborate on software projects, familiarizing the students with distributed collaborative software development. In all projects, software for computer-mediated collaboration was developed. The details of the projects will not be mentioned here, we focus on the outcome of the study concerning collaboration between the groups only. The results were gained by interviews with and reports from the students, surveys about team harmony and tool use, plus an analysis of all electronic messages. These 32 week projects were realized by students of the Massachusetts Institute of Technology (MIT) in the US and the Centro de Investigacientca y de Educaciperior de Ensenada (CICESE) in Mexico.

Each student had to adopt a role within the team, and additional tasks had to be performed, requiring the students to develop questions for clients and gather requirements by interviewing experts and potential users. Next to the analysis of commercial products, experience reports using similar tools had to be made, and Use Case Diagrams created collaboratively.

In the beginning of the course, instructors scheduled meetings and offered other kinds of assistance helping the teams to coordinate the work. The meeting agenda was solely left to the students after some weeks. During the whole course, weekly lectures were held—via video, reinforced by text-based chat to anticipate potentially confusing content occurring through insufficient audio and video quality. The development effort was higher than anticipated, and although the instructors warned the students about not underestimating the communication and coordination effort, they did. The mandatory prearranged communication was made up of weekly held formal meetings, using desktop video conferencing equipment. For everything else, video conferencing, e-mails and messaging systems were mostly used. For the everyday communication, instant messaging systems like ICQ [60] and e-mail were preferred by far. Telephone and the postal system were seldom called on.

The conductors of the study wanted to create a flexible environment including different loosely integrated tools, using shared resources. Selected tools which did not fulfill the requirements were exchanged or modified during the process. The suitability of a tool depends on

various criteria, such as group size and location. The most suitable tools where determined by modifying the combination of tools. Moving easily between the tools and the documentation was facilitated by providing appropriate links.

Results. In the following, we shortly discuss the results concerning social aspects like different languages and cultures, and how awareness was created.

Different languages. Two countries and with it two native languages were involved in the projects. It was agreed that all conversations take place in English. Although the language skills of the participating students from the Mexican university were perfectly sufficient, there was a language barrier. This was due to psychology, not to skills. The students with Spanish as their native language were hesitant to speak in English as a result of a lack of confidence. This problem could not be overcome for a long time. With the relationships established, this sensitivity to language lessened.

Cultural differences. The cultural differences occurring in the project do not have to be seen only dependent on the fact that the participants live in different countries, but refer, in our opinion, to the different manner of education at the two institutes, which would be possible at two universities—or companies—in the same country too. At MIT the students were encouraged to be outspoken, which might be offensive when accustomed to another policy. At CICESE the work was more group oriented, while MIT encouraged individual responsibility. On account of these differences, several problems occurred, which were, much like those of language, lessened or solved over time—not without making efforts on both sides. It is important not to underestimate and always be aware of the potential difficulties arising when members of a team have different mother languages, cultures, or long-established policies.

Awareness. When a participant requested project information from the repository, the system informed the other teammates of his/her presence by sending User ID and current URL to an awareness server, where information on all current user's locations are stored. Every move updates the information on the server, and the information is displayed for the other users in a separate window. Within the awareness window, a person can be selected, and a textual communication with this specific person is established.

Awareness was created by publishing as much information as possible, as small as the information may have seemed. Even the authors of this study called this policy seemingly tedious, but with it the entire team was on one wavelength. Everyone was provided with any possibly needed information. To ensure, that a meeting was fully understood, the main points were posted on the web.

MDD Tools

There are several tools available supporting model-driven development (MDD), athough MDD is a relatively young software development discipline. Just a few examples of MDD tools are

the Eclipse plug-in Omondo[12], MagicDraw[13], or Enterprise Architect[14]. These tools support different modelling and programming languages, but they usually do not offer collaborative support.

ArgoUML

ArgoUML[15] [88] is an open source UML modeling tool, it includes support for all standard UML 1.4 diagrams. There are no collaborative features within ArgoUML. We mention it because of the way diagrams are depicted. The drawn shapes are coloured, as shown in Figure 4.15, presenting themselves very clearly laid out. A clear depiction of the model elements is important to properly support MDD. Since users are reluctant to comment source code or models in the first place, the good visibility and clear distinctness from other model elements of the comment—in this case of the pink note—may additionally lead to more frequent use of comments.

The Odyssey VCS

The Odyssey VCS is a version control system tailored to fine-grained UML model elements [83]. Using heterogeneous UML-based tools, Odyssey aims to support concurrent modelling. In 2008 some enhancements for Odyssey 2 were planed, including an expansion from an optimistic concurrency policy only to the possibility of pessimistic model versioning added [80]. This is achieved through a lock command, without changing the versioning model. If a lock command is applied on a model element, check-in commands by the lock owner only are accepted. This person can also unlock the model element, otherwise it will automatically be unlocked after the previously mentioned check-in. Other improvements concern branching, and the support for UML 2 and for hooks. The first version of Odyssey did not contain extension mechanisms. Mechanisms to trigger external tools are enabled through hooks. The focus of the Odyssey VCS and its followers is improving the handling of models in versioning and doing that concurrently. Supporting communication and coordination is left to external tools, enabled through hooks in Odyssey VCS 2.

MetaEdit+

MetaEdit, a tool for model-based development, first was designed to be a single-user tool supporting one modelling language at a time. Its successor, MetaEdit+, allows several modelling languages and offers single-user as well as multi-user sessions. MetaEdit+ enables creation, maintenance, manipulation, retrieval and representation of design information among multiple developers [78]. A multi-user session is managed by the *Repository Administrator*. Figure 4.16 shows the user page, where the adminstrator manages the user sessions. The administrator can, for example, create new users or delete user accounts.

[12] http://www.uml2.org/
[13] http://www.magicdraw.com/
[14] http://www.sparxsystems.de/uml/ea-function/?L=0%3Ftx_indexedsearch[sword]%3Dcooperate%20licence
[15] http://argouml.tigris.org

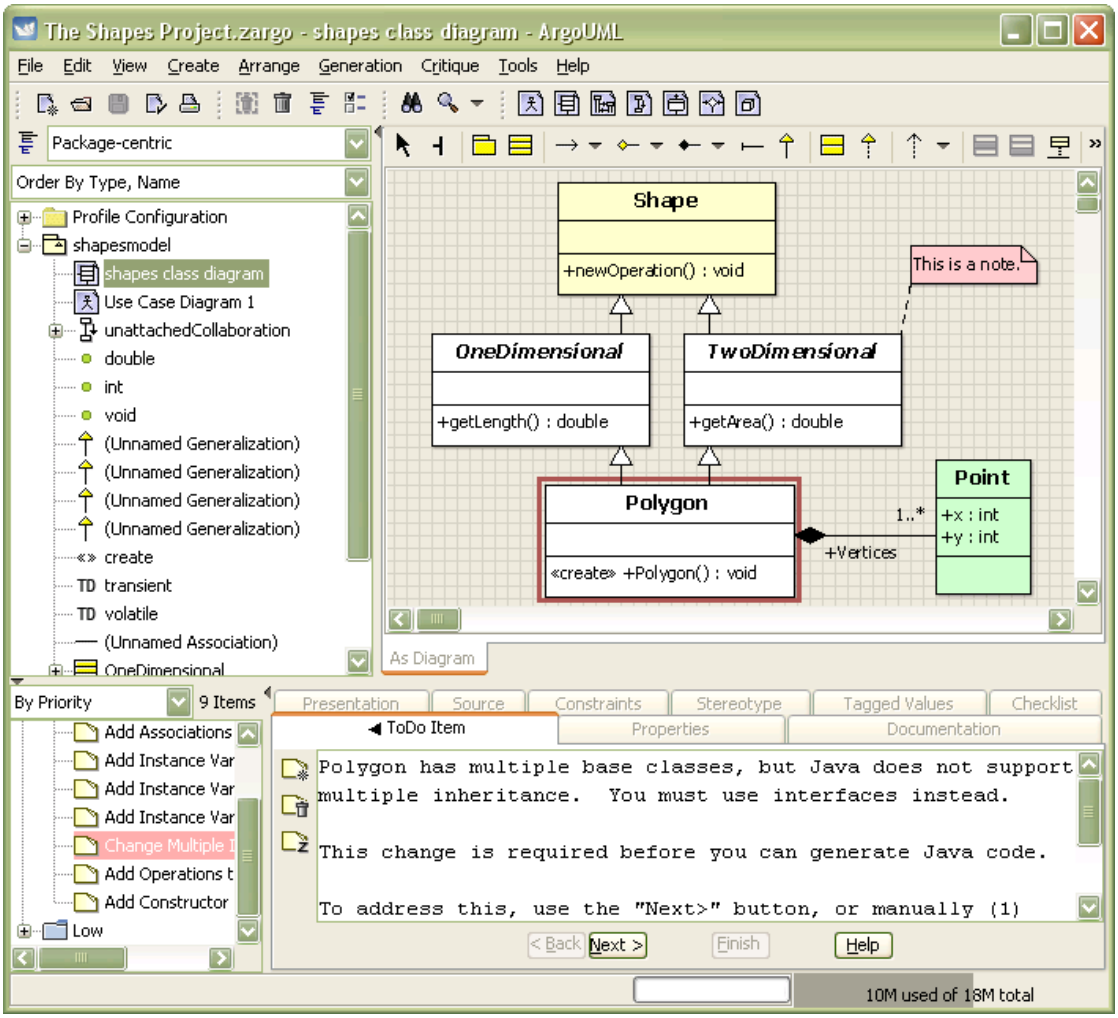


Figure 4.15: ArgoUML

MetaEdit+ is an example of a MDD tool, where collaboration is supported by furthering concurrent work, in this case with multi-user sessions. A user list provides information, who else is—at the same time—working on a certain fragment. There are no features for communication built in.

Enhanced Collaborative Support within SE Tools

There are few tools supporting collaboration by more than version control and other basic features, which allow concurrent work and help to coordinate the team, as the Odyssey VCS and MetaEdit+ do. Advanced collaborative support is given in the JAZZ project. Wong et al. provide a study on how assisting the collaborative process results can change, using the tool CONFER [114].

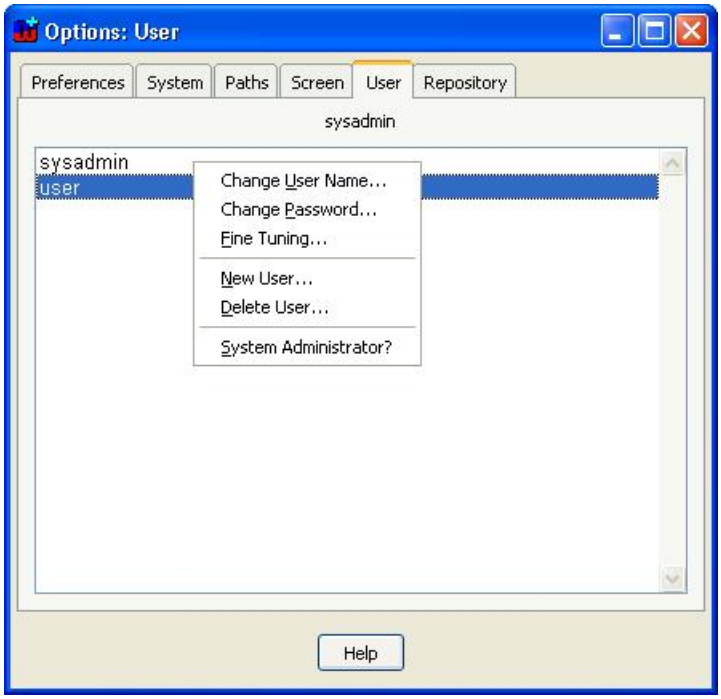


Figure 4.16: User list in MetaEdit+ [78]

Supporting Collaboration in CONFER

CONFER (CONflict Free Editing in a Replicated architecture) [114] is a tool detecting conflicting operations in software engineering, focusing on software engineering diagrams. Conflicts are not only detected, they are recorded and highlighted and the users are notified. The detected conflicts are stored to be resolved later by user actions. So the actual conflict resolution solely takes place in the social domain, while CONFER maintains divergent copies until differences are resolved.

There is an existing technical environment, a prototype of this collaborative editing system was partially implemented in 2006 to proof the concept. In [114], Wong et al. want to add collaboration to support the conflict resolution process in the social domain. They therefore conducted a study concerning a certain method of manual conflict resolution. The experiment conducted shows the improvements to conflict resolution by keeping to a spoecific chain of activities, while communication was reduced.

Experiment. The experiment aimed to test the subsequently introduced CONFER method for conflict resolution. 60 persons were divided into two groups, one using this method, and the second group would be the control group for comparison, carrying out their work without any specific conflict resolution aid.

The assignment for both groups was to develop a class diagram concentrating on the basic structure of classes and their associations according to a problem description open to different

interpretations.The social support mechanism was simulated by a facilitator and all participants using paper and pen to transform the problem description into classes, class hierarchy and relationships between the classes. Hence no one was able to see the others work before finishing ones own work, and not able to see conflicts earlier.

The proposed CONFER conflict resolving method for the experiment:

The persons first had to study the given problem definition individually and try to solve them. Then the facilitator collected those solitary made resolutions and identified the conflicts within the group. The subjects were not allowed to talk to each other, communication was possible through the facilitator only. Each member was informed about the conflicts and asked to look at their own work again, and the participants were allowed to withdraw their design. If the conflict was not resolved by then, alternatives had to be proposed, those were available for all group members. Another review, and then the participants could add a level of confidence to their designs. Everyone had to review their proposals again and then the group would together, face-to-face, produce an agreed diagram by discussing remaining conflicts.

The control group was working as follows: Everyone would individually read the problem description and try to solve it. After conflicts were highlighted by the facilitator—until now the same procedure as the experimental group—the participants could discuss their work and try to find a resolution collaboratively but without assistance.

Results. Wong et al. aimed to find out the subject's satisfaction with the model and how well the model would assist them with conflict resolution. The interviewed participants indicated support for the proposed method and found the process easy to follow.

The number of conflicts and the time it took to resolve this conflicts were recorded. The mean number of conflicts found in the experimental group was 3.5, while it was 4.3 in the control group. Even more significant is the difference between the average time to resolve conflicts—about 16 minutes for the experimental group and nearly 30 minutes in the control group.

Although the CONFER strategy may sound complicated in comparison to the control groups strategy—just get together and talk it over—conflicts were resolved quicker. The diagrams of both groups had comparable quality, so one can not make a statement about the proposed approach having an effect on quality. Although the authors of [114] caution the readers not to draw a too wide conclusion from the experiment, one can definitely assume that the proposed method can help reduce the time needed to resolve conflicts. Furthermore the CONFER approach seemed to be easy to use and may contribute to the effectiveness of the conflict resolution process and therefore be a useful extension to CONFER. Concerning supporting the collaborative process in general and conflict situations in particular, there are, of course, different other potential focuses for future work.

JAZZ

The Jazz project seeks to integrate collaborative capabilities into the Eclipse IDE for small teams of developers to work together more effectively [26, 65]. The focus is not on software development, for the development platform Eclipse takes care of that, but enabling collaboration, as well as defining, administrating and integrating development processes [58]. Productivity shall

be raised by facilitating awareness, communication, and coordination among the members of a team [27]. Jazz is based on the metaphor of an open office approach, as Cheng et al. [26, 27] describe, where in a team of developers everyone has his/her own workstation, and elsewhere there is room for team meetings, shared whiteboards, schedule information, and others. Communication is facilitated in a way, that, for example, questions can be addressed to everyone, or particular persons can be consulted.

The foundation of Jazz was laid among others by Li-Te Cheng of the IBM Watson Research Center [50], and the idea described in 2003 by Cheng et al. in their paper "Building Collaboration into IDEs" [25]. A preliminary version attempted to integrate instant messaging into an IDE, the present version includes much more, for example user list with fotos (cf. the *Jazz Band*) [50]. The goal of the Jazz-enhanced Eclipse environment is to provide access to as much of the team information as possible, and to restrict the work of any team member as little as possible [26]. Jazz supports the development team throughout the whole software lifecycle, by offering a single consistent GUI framework for all tasks [65].

Cheng et al. [27] describe the *Jazz Band* as the most visible aspect of the Jazz enhancements to Eclipse. It is a small strip at the bottom of the screen, providing peripheral awareness of the other developers. At first glance one can find out about any team members' status (online or offline), and if he or she is active at the workstation. When hovering over icons, the user gets additional information about the other developers activities and the files they did check out of source control.

The *Team Jam*, accessed through the Jazz Band, is a team-centric discussion board, were messages and questions can be posted for the whole team or a chat in instant messaging style with a particular team member initiated. These private discussions can afterwards optionally be posted to the Team Jam.

Jazz offers synchronous chat and asynchronous messages, which can be associated with a place in a code file, linking code and discussion together. Another feature of Jazz is screen-sharing, for pair programming, code walkthroughs, or consulting. All communications and file reservations are stored in the Team Jam accessible from editor markers, personal icons on the Jazz Band, and the Package Explorer.

Figure 4.17 shows the Jazz Band in detail. There is a pop-up menu to set the user's status message and manage all teams (a). On the bottom right of each portrait is the user's instant messaging status. Offline team members have dark, gray-scale portraits. People are grouped in user-defined teams (b). Hovering over portraits reveals their online status messages (c). Another pop-up menu for a team member reveals options to start a chat or share the screen (d).

One of the main goals is to further awareness, not only of team members but also resource-centered awareness accomplished through extensions to the Eclipse Package Explorer, were recently checked out or modified packages are marked specifically. Markers are used to indicate discussions or annotiations linked to the source code, and the content is easy accessible through these markers.

As mentioned before, Jazz is designed for small teams. Those are meant to be informal, ad hoc, and invitation-based. Privileges are equally divided among the participants. Any changes

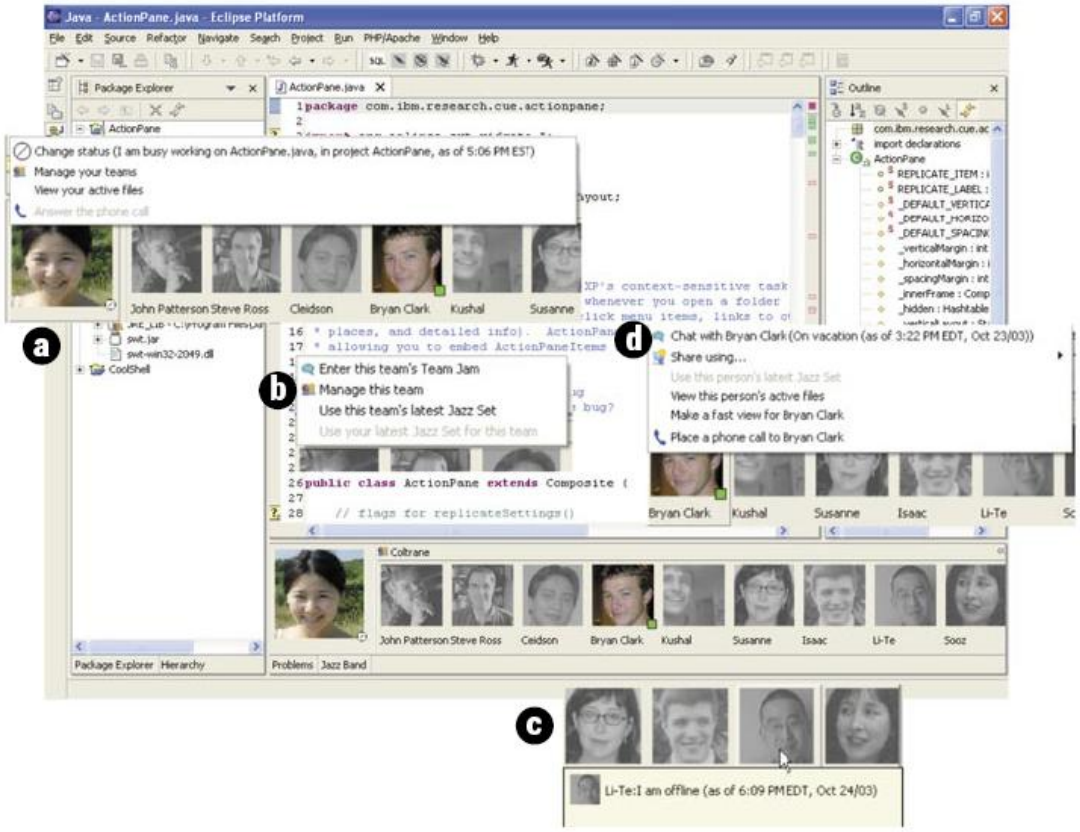


Figure 4.17: Jazz Band and options [25]

made immediately are displayed for the other users in the Jazz Band [26]. The development team does not necessarily have to be small, Jazz is intended for the "immediate team" [25], or "the core team of developers" [26].

4.6 Games

Future internet communications may be based on what is now used by virtual communities playing online games, hence it is important to study social interactions within these kinds of communities [15]. Whitehead [112] names the use of massively multiplayer online (MMO) game technology as a collaboration medium a possible future direction for collaboration in software engineering.

A subcategory of massively multiplayer online games (MMOG) are massively multiplayer online role-playing games (MMORPG). In MMORPGs a large number of players interacts within a virtual world via the internet, usually communication between them is facilitated by

different possibilities within the game.

Early multi-user role-playing games called multi-user dungeon (MUD), described entirely in text, or graphical MUDs, additionally offering graphical virtual worlds, were the predecessor of MMORPG. In the category MUD, there are systems for distance education or virtual conferences too. MOOs (MUD, object oriented), which are text-based virtual reality systems, are not only used for role-playing-games (RPGs), but since the use for games finds favour with the largest group and they are developed farthest, we take a look at the way collaboration works in systems like that.

World of Warcraft

By far the most popular interactive online game is World of Warcraft (WoW)[16]. The users choose characters and professions, which give them certain abilities. In a vast virtual realm they can communicate, fight against, or work together with other players or non-player (computer-controlled) characters (Figure 4.18 shows a screenshot within a WoW game by a forum member of the european WoW website). The part of social interactions in this sort of game we are interested in, is how the communication works, and why it works so well. The communication possibilities are extensively used by the players. There are a lot of negative—even alarming—reports concerning the impact of playing WoW, but we are not concerned with the real-world social skills of hardcore players.

The communication in WoW is designed for players to learn to be at ease with strangers [81], and since some peoples' main or only relationships are those within the game, it obviously works. We want to learn from the prosperity of World of Warcrafts' collaboration, and therefore take a look into the details of collaboration of the game.

Please note that WoW is under constant further development and some features or their details may already have changed or be obsolete.

The biggest differences to professional collaborative activities are the motivation of the participants and the pursued goals. Working together seems to occur as a goal in both fields—in profession and in spare time. By acting jointly the primary goals are promoted. In the professional area this would be a product realized through the collaborative process. In the game field on the other hand, the main goal is to win, as an individual of course. And the collaboration is the use—or even abuse—of other players. Brignall et al. [15] state so fittingly: "Make no mistake, WoW's game dynamics center around competition". Teamwork in a professional environment, for example in model versioning, is characterised by *collaboration*, while in a game like WoW, it can be called *competition* with collaboration. Yet World of Warcraft is a rich source for well working collaboration. Predominantly the communication and certain general coordination is interesting for us, since the rest of the collaboration specifically is designed and used for the game itself, not providing us with a comparison to our research field.

The communication is mainly done by chat, though there are other possibilities like the *e-mode*, where a player can make his/her virtual figure laugh or show other emotions. Also external

[16] http://www.wow-europe.com

Figure 4.18: The World of Warcraft

programs can be used to broaden the communication possibilities. For further information see the following section on Teamspeak.

The key communication feature, the chat, is shown in a closeable window, of which the size can be changed too, in the lower left corner of the screen. The messages are colored not according to the originator—the communication partners change all the time—but by the nature of the message itself. The users names are shown in squared brackets, followed by the message. There is no additional signal for an incoming messages, like an audio signal or flashing used by Skype or ICQ (see Section 4.1).

Here is a summarisation of the different kinds of communication:

- **Talk.** When something is just “said” , the message is displayed in a speech bubble above the virtual figures head, visible to anyone looking at that figure at that moment, and also appears in the chat window of anyone within a certain radius—coloured in white.
- **Whisper.** Whispering is a possibility to send a private message to one person only, displayed in purple.

- **Shout.** Important or emotional messages can be sent by "screaming" the message. Those messages appear coloured in red.

- **Group messages.** To send a message to a certain group of people, your team in that case, one sends a group message. These messages will be displayed in blue, received by members of the group wherever their figure is in the game, as long as they are online. Also there are so-called raid-messages, but a raid is simply a larger group.

How does a player specify the kind of message to be sent? There are different possibilities. Any of the former described messages can be sent by starting the message with a certain shortcut. For example \g starts a group message and \w *username message* is for a private message called whispering. Whispering can also be accomplished by either click the name of the intended addressee within the chat window or use the additional options provided by clicking on the user profile of a person.

There also is an ignore list. Messages from players in that list will—as the name indicates—just be ignored. This list is to prevent expected hostile messages or bullying.

Every player—actually every player's virtual figure—has a profile. There also are information windows for the groups and a function allowing the user to list the player, and look for someone by filtering this list by different criteria.

Battle.net

Blizzard Entertainment[17], the company that created World of Warcraft, maintains a website for their clients called battle.net[18]. This website mainly offers forums for WoW and other multiplayer Blizzard games. These forums include technical support, beginners guides and suggestions addressed to the creators.

The european WoW website[19] promotes a new extension of battle.net, a *friend system*. Similar to Facebook[20], users on this friend system can add other users to their friends list. Users in the friend system appear with their real name, additional information provided includes their characters in the Blizzard games, which level in the game they are on, or a current status. The option to chat is given regardless of the game played at the moment. The friend system on battle.net is intended to increase the social component of the game. User reactions in the forums of battle.net concerning the new WoW-Facebook are divided at least.

TeamSpeak and Similar Software

Browsing through different bulletin boards (our readers will hopefully excuse that we do not mention them one by one, since the reliability of the offered information is questionable and the rest are just opinions) we found several game players looking for either attachable or additional

[17] http://www.blizzard.com
[18] http://www.battle.net
[19] http://wow-europe.com
[20] http://www.facebook.com

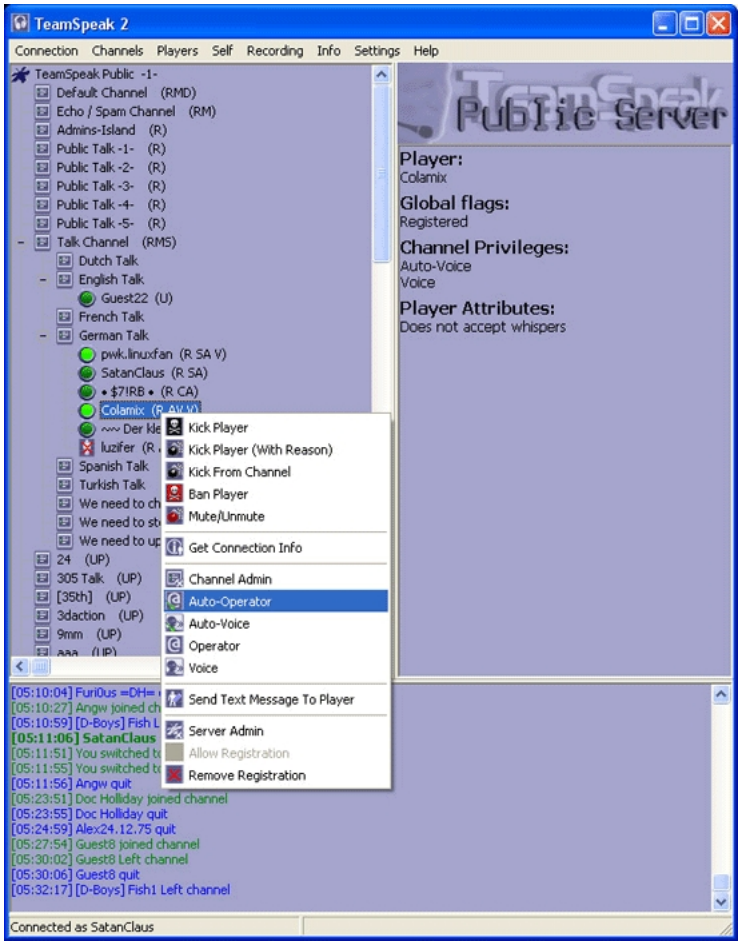


Figure 4.19: Screenshot of TeamSpeak 2 [105]

software to use communication features not included into the game, like an in-game voice chat, and yet additional software to organise the former mentioned by finding out who is speaking at the moment and display that information while playing and thereby not disturbing the game itself.

TeamSpeak (TS) is an external communication software, enabling voice integration within online games, virtual worlds and MOOs, social networks, or education [105]. Through IP addresses and passwords the communication is established.

Figure 4.19 shows a screenshot of the second version of TeamSpeak, displaying channel options and the menu options for a certain participant opened, providing options to kick, ban, or mute a player from the channel. The client user interface is very intuitive according to the developers. The main feature is VoIP, text messages are possible too, foremost to send URLs or IP addresses to avoid dictating them.

When TeamSpeak is used with the game World of Warcraft, there is no actual contact be-

tween those two software systems, for WoW does not allow it. To blend additional information into the screen while the game is played, yet another software is needed, for example the TeamSpeak-Overlay (TSO) software [108], which was designed to be used with TS and displays the name of the person speaking, overlaying the game with a text via DirectX. TSO does not support the latest operating systems or DirectX versions, for it is no longer in development.

TSO can be used with TeamSpeak and also with Ventrilo [109], another VoIP group communication software. Ventrilo and TeamSpeak are, although applicable for other systems, most popular for online multiplayer games like WoW or Vendetta Online[21].

Virtual and Real Worlds

Games like World of Warcraft enjoy great popularity. This is partly due to the fact, that people like to assume roles, to act as different characters. The game itself is one reason for the popularity, that the game is a multiplayer game is another important reason. A game providing only interaction between the player and computer-controlled players will never be as successful as multiplayer games. Interacting with real people makes the game more real. Still, the social interaction takes place within the game. The social interaction between players is different from the interaction between co-workers in a professional environment due to the different goals and kinds of relationships. Nevertheless, WoW shows how communication can successfully be incorporated with a vast number of ever changing participants.

The introduction of Teamspeak and similar software again displays how missing communication features are substituted by the users. To avoid that additional software has to be used, which complicates the process and leads for example to poor traceability, the creators of a system have to carefully think about which features will be needed.

4.7 Summary

Teamwork in general and collaborative support and communication features can be found in various fields. There are tools independent of other tasks performed, that is for communication purposes only, providing different communication features. ICQ and Skype (cf. Section 4.1) offer textual and audio/video communication, the Windows Life Messenger (cf. Section **??**wlm) textual communication and drawing, in the Video Viewer (cf. Section 4.1 audio/video communication is facilitated, whereas all three possibilities are incorporated in Microsofts NetMeeting and subsequent products (cf. Section 4.3. These features are implemented in different ways and have successfully become a part of all our daily online communication.

Examples for tool support for collaborative activities can be found in collaborative writing—for a group of users writing scientific articles in Quilt (cf.Section 4.2), or for any willing participant on the web in Wikipedia (cf.Section 4.2). Supporting cooperative drawing is implemented in different ways, more general in Draw-Together (cf. Section 4.2), and specifically for collaboratively drawing models in several tools (cf. Section 4.4). To extend several existing tools with collaborative support, frameworks may be built around them (cf. Section 4.3).

[21] http://www.vendetta-online.com

In software engineering, there are now several approaches made to further teamwork by including collaborative features. For model-driven software engineering, the Jazz project (cf. Section4.5) provides an example of successfully integrated collaborative features. Still, the most mature tools supporting collaborative work in the field of MDE are for only drawing models, without real MDE support.

Studies concerning certain tools and especially teaching and learning situations in software engineering provide valuable insight into individual behaviour, and point out the influence of certain social aspects (cf. Section 4.5). Multiplayer games (cf. Section 4.6) offer a new way to collaborate in a virtual world. As an example we give World of Warcraft, at least part of the success of which has to be explained by the integrated collaborative support.

This chapter lists examples of collaboration in various fields, implemented in different ways to achieve different goals, which are summarised, arranged, and analysed in the following chapter.

CHAPTER 5

The Big Picture on Communication

The usability and therefore the acceptance of a tools' collaborative features depends on the location of participants, the structure of the organization, the organizational culture, the time of collaboration and the individual social roles. In the following, criteria for successful collaborative features in model versioning, possibilities for these, and their combination are discussed. We then follow up on relevant social aspects, and conclude with the role of distributed software development.

5.1 Criteria

Criteria for successful version control in MDD also apply to the collaborative aspects. That is for example traceability—a communication has to be traced the same as different versions of a software artefact—or usability, which has to be given for any IDE as well as for any GUI of a collaborative tool. Additional desirable goals for good collaboration include awareness. The following section describes criteria for efficient collaboration in model-driven development, which have to be minded especially.

Traceability. Not only production of code and the versioning of it, has to be traceable, but also the collaboration. Reviewing the process behind the core tasks, the communication, allows retracing not only which changes were made on the source code, but also for example why the changes were made. When using different systems for conversing and coordinating, this process is made more difficult due to the fact, that retracing a certain collaboration might include looking through sent and received e-mails, going through meeting protocols plus finding out the date and time of those, as well as numerous other activities.

Traceability of collaboration is important, and it is made much easier by a central storage, providing the person looking for information on past events with one place—and not several—to search for the desired information. When looking for certain information, it would be best to have several possibilities, like search by timestamp, by tag, by originator, or by files.

In a working environment, where the core tasks are taken on by a specific tool, and the collaborative part by several others, like an e-mail client, an instant messaging system, an additional calendar, and a word-processing application for documentation, tracing back information is made much more difficult.

Usability. The more the offered collaborative features are liked or at least accepted, the more they are used, and therefore the more sense they make. Usability is provided by the right selection and combination of tools, they have to be easy to access and to use. The graphical realisation, the user interface, has to be defined in a manner, that makes potential users want to use it. Clarity is the keyword for the user interface. It has to be well-arranged, and functions and options need to be clear, and—again—easy to access. Ad-hoc communication for example should even be faster to start as prearranged communication. These and other criteria have to be minded to avoid the necessity of users switching to other more convenient ways to collaborate.

Awareness. Versions, processes, and communications being traceable afterwards is important, but also being aware of the current state. Awareness of the current status of a team member includes knowing whether he or she is approachable, i.e. if a person is online, offline, or busy at the moment. This information may be provided through profiles including announcement of the present status. These profiles can support awareness in two other ways. First, what is the person competent for, what are the responsibilities and authorisations. Second, either an automated regularly updated profile, or regular entries by the user, can display on what this person is working on at the moment, may it be resources like a certain file or a task.

Concerning awareness of files linked to persons, there are different options. In pessimistic versioning, one is made aware of the fact, that someone is using a certain version by the version being locked for the use by another person. We look for solutions in optimistic versioning. Notes, either in form of comments or a sort of sticky note, may be added to files to facilitate awareness. Imagine a situation, where person A is working on a certain artefact, adding a at first glance visible note, like “user A is occupied with the further development of this artefact this week, please contact before changing anything”, which could simply avoid a potential conflict.

Software developers often are reluctant to satisfactory and descriptively comment produced code, but general comments or non at all are hindering the maintenance later [52]. Even more important are descriptive comments, within the code or commit comments, in team work. Raising awareness is only one advantage. Satisfactory commenting is furthered by handy means and policies set to bring people to do so.

5.2 Integration of Features

Cheng et al. [25] ask themselves “why integrate collaboration?”. Collaborative tools for e-mail, instant messaging, or others have been used outside the development environment for a long time. But the integration of collaborative features, like screen sharing, instant messaging or e-mail, brings benefits for the development team.

According to Singer (National Research Council of Canada), collaboration in software development evolved via tools that control artefacts like code and documentation, and the Jazz

project takes that one step above by incorporating collaborative tools [50]. Systems in various other fields make a similar effort to integrate collaborative tools to provide a link between collaboration and the main purpose. In the case of software development the main purpose is source code development.

Using numerous external tools for collaboration takes up a lot of time and complicates traceability. Tools which should support users in organising their work create an information and communication overload, and thereby a disturbance of work [87].

Tang et al. [104] found that their offered desktop conferencing feature partly substituted other formerly used ways of communication like the phone or e-mail. This is due to different reasons. First, compared to e-mail, video conferencing offers synchronous communication. Second, distributed locations were involved, which influenced the choice of means for communication. Because of different locations and the resulting time difference, synchronous communication was not sufficient. The reason why the video feature substituted the phone to a certain extent is for one thing the possibility to see each other. Still the novelty of the feature for the group, and that the study was all about it, surely added to the positive results concerning the use of the tested video conferencing system.

Integrated features relieve the team from switching between different external tools, which are necessary, for in different situations different tools are beneficial. It also improves the traceability of additional information, by providing easy access. This kind of information could be a problem solving process, which already took place in a similar way, and can be reused. When integrating features for collaboration, these can be specifically chosen. This anticipates overlaps, occurring when the users choose themselves. Also, different teams would choose different means of communication, and yet would have to choose common tools when interacting. This is evaded by providing everyone with the same set of features.

There are a lot of different circumstances influencing the acceptance for integrated collaborative tools. Studies for different collaborative features usually prove the usefulness of those tested, as for example the video conferencing example mentioned above. Putting together all these studies leads to inconclusive results, for every study favores the studied collaborative feature. The only common outcome is, that integrating collaborative features supports the process, by either reducing the time needed to complete tasks, rising quality, or an overall improvement of the collaborative process. Therefore it is a good idea to deliberately integrate collaborative features into the process, and better yet, into tools used for everyday tasks.

Concerning the inconsistent outcome of comparing studies for the usefulness of different collaborative features, the conclusion is that only real world tests would prove, how many benefits lie in them, depending on the actual circumstances they are intended to be used in. Also, the integration of yet other features, for example additional chat to a video conferencing system, would change these results again.

5.3 Choosing the Way of Communication

The way of communication can be divided according to how it is done and how it is absorbed. We came across several overlaps, by dividing into writing, drawing, talking, and watching. Within this subdivision, even more means of communication may be identified as well as details

one has to mind when choosing and using them, which are described in this section. As usual, in different situations certain ways of communication or their combination are preferred. This is dependable for example if the conversation is ad-hoc or prearranged, or if it is used for updates, conflict resolution, or others. In the following, communication possibilities are divided into for categories, examples are given, and advantages and disadvantages discussed.

Writing

Written communication means communication like e-mails or instant messages, chat, different types of notifications and features like calendar files. In all these cases policies on how to write might be desirable.

E-mails

Probably the most popular way of asynchronous communication is e-mail. Obviously because it is free, easy to use, takes up little time and is delivered fast. A lot of people not owning a PC do own an e-mail address. Online and paper contact forms show the wide acceptance of electronic mail, since they usually require an e-mail address. Robillard et al. [90] observed, that people will even use electronic mail, when working in the same room or office, to ask questions and provide answers rather than talking personally. The most important advantage of e-mail is that people are accustomed to using it.

When working on a project, and using this mean of communication, it could prevent the enquirer to receive an answer to an important question at once if the messages are not read in time. This can be avoided by setting intern policies to check your mail regularly or providing notifications for the receipt. As an example for setting a rule like this, see Section 4.5.

Chat

When using a chat feature, one sends messages to one or more persons, who are instantly able to read and answer them, even simultaneously, usually being provided with a history enabling you to inspect former comments and whole communications. These features are provided by free tools like ICQ or Skype (see Section 4.1), and even websites have chat options, which usually only require registration and login to use. Especially when using a chat option, people like to use short language, examples would be “lol” (laughing out loud) or “asap” (as soon as possible). In order to guarantee for everyone to understand the communication without having to enquire, rules have to be set up to exclude or limit the use of acronyms.

Colour your chat. Colour immensely improves the clarity of a chat window. There are different ways to colour messages. In Camel [24], a colour is assigned to a person, and the name and message of this user are displayed in that colour. Next to the chat window is a smaller one containing a list of the sessions’ participants, their names and assigned colours. An additional window like this helps to always be aware of who is participating whether the person has just sent a message or not. The more participants, the more nessecary is such a feature.

In WoW (see Section 4.6) the messages are coloured according to their type but independent of the originator. Normal messages for everyone around are displayed in white. With a white window, the desired colour would be black. Group messages are blue and a private message ("whisper") is shown purple. The colouring describes the importance of a message, if you "shout" in this game, it is displayed in the signal colour red.

When using ICQ, the chat window displays one's own name in blue and the name of a second person in red, with a disengageable timestamp. The message itself is by default setting black, in font Ariel size 10. The font, its size, and the colour can be changed easily for every message, it can be left-aligned or right-aligned and displayed italic or bold. Features usually provided by any text editor. These gadgets are definitely dispensable in our case, there is no need to offer the participants to choose their favourite colour, but it still might be a nice accessory. The possibility to change ones' assigned colour, or better choose it at the beginning of a conversation, may make someone happy and thereby increase the acceptance for this particular communication feature.

The preferred method of colouring a chat communication may change with the number of participants. If there are about two to four users partaking, assigning a colour to everyone supports clarity very much. With a larger group participating, a lot of maybe similar colours can be confusing, therefore a disposition in groups or another way of colouring might be interesting.

Regarding the way of colouring, with a limited number of participants and types of messages, the first approach is preferred, assigning colours to persons. Additionally it would be a good idea to leave out the colour red, plainly *the* signal colour, to use it to point out important messages, so they can not be overlooked. Not unlike the possibility to prioritise e-mails with red exclamation marks.

Chatting with your system. Notifications or warnings, originating not from another person but the system, are usually shown separately from ongoing conversations. Within a chat option, a notification can also be built in. Similar as in WoW (see Section 4.6), where messages of non-player-characters are like any other message part of the chat. It can be beneficial, not to interrupt a collaborative process with a pop-up message, but integrate this message. These messages can be coloured in a specific colour, like black or a signal colour like red.

To Whom It May Concern

There should be a differentiation between public and private messages. When answering an e-mail, the answer can be sent to either all quoted addresses or the originator only. Regarding chat the matter of public and private messages is implemented in Skype as follows. In a phone communication with several people at once, an additional chat window—actually several ones concurrently—may be opened to send a written message not intended for anyone else. The same applies when in a group chat. An additional chat window allows messages to be sent to a certain person only.

An interesting appliance of dividing messages in public and private ones is offered by the game WoW (see Section 4.6). Just sending a message makes it a public one, for everyone around to see. There are two types of private messages. First, there is the group message, which every member of your team receives. Secondly, you can "whisper" to a certain person, and only this person will get the message. These special messages may be sent by starting the chat message

with an abbreviation, "\g" for group messages, "\w" and the players name to whisper. There are two other possibilities to whisper. To open a menu by clicking on a players' portrait with the right mouse button, or to click on the name of the person you want to send a private message to in the chat window, which will open another window to do so.

Imagine a larger group conversing via chat, maybe some programmers and some people occupied with testing. Tester A has a comment or an idea he wants to share with tester B, only the message would disturb the communication flow for the whole group. Therefore the possibility of a private message would be advantageous. The makers of WoW provide us with ideas to easily choose the type of message, either to start the message with an abbreviation or select the intended targets.

The game section offers another example of private and public messages. TeamSpeak (see Section 4.6), a tool to additionally talk while playing games like WoW—where only written communication is possible—provides written communication too. These messages are intended for sending information regarding the audio communication, not for actually chatting. This information, like IP-addresses, is intended for a certain group of people. WoW only offers private messages to one person, group messages to your group or public messages. The mentioned group of people may not coincide with these options, so TeamSpeak has to offer another one. Ideally private messages are sent to any person or any group of persons.

Timestamps

By default setting a timestamp is added next to the users' name when conversing via the ICQ or Skype chat. This has two advantages. Within a communication one may look up, when exactly a certain message was sent. Afterwards, when looking at the history, or searching for a communication at a specific time, the same is possible. Written conversation, stored for eventual later revision, should be attached with a timestamp. Either the whole conversation is marked with a timestamp, or every message itself.

When a team is geographically distributed over different time zones, a problem concerning the timestamps arises. Which local time is used? A chat conversation with user A and B in different locations offers the possibility to either use both local times, respectively according to the user, which is confusing during the conversation and at revision. Choosing one local time confuses the user on the other end. The solution left is to display the messages of both users in their own local time. In separate repositories, the information can be stored this way, lightening the revision. In a central repository, one local time has to be chosen. Being aware of the chosen time zone for the stored information should not hinder revision.

Comments

Commenting source code in software engineering is very important, still developers are often reluctant to do so [52]. Not every comment is practical to be and remain in the source code. Modification proposals communicated through comments like this make the code confusing. If the comment would be sent by other means, for example e-mail, the recipient would be forced to search through the source code to find the position of the place the comment is intended for.

This problem may be solved by a sort of “sticky note”, easier to add and remove, and more salient, than a comment within the code. If this kind of comment were in colour, for example yellow like post-its, it serves the purpose of the very ones. This sort of notes would be used for messages, which do not have to be kept.

Notifications

Notifications are unidirectional messages, usually sent from the system to a user, or from a user to one or more other users. In case of human-human interaction, a notification like this is a single message, without an answer being sent the same way. In case of computer-human interaction, a notification can be a warning, for example reporting a conflict situation. In ICQ (see Section 4.1) there are notifications about incoming messages, birthdays coming up of people in the user list, and others.

The game World of Warcraft includes, next to the players representation, computer-operated characters, so called non-player characters (NPC), whose chat messages are included in the chat window the same as any other players' messages. Within SLiM (see Section 4.4), the messages in the chat window include messages from the users to each other, and messages describing the process. In Figure 4.14 in the mentioned Section, Amanda requested the current session state, and Ron received the request. Those are two examples for including messages from the system into the users' chat. A third possibility would be to include notifications into the chat. Notifications, for example in a tool for collaboratively drawing models including a chat, could be that a finished model is invalid. Instead of a pop-up warning, this notification may be included into the users chat. While simple steps of the way, like *user draws relationship* would not be included into the chat, leaving room for more important messages, that is notifications or the users' discussion.

Especially in case of an important notification, like a warning, a pop-up window may be preferred. Depending on the nature of the message and its importance, its appearance may be set. This can be a pop-up window, a message integrated into a users' chat, or a message shoeing up in the log only.

Notifications from one user to another may be sent according to their importance too. An important message may be sent through a notification system, while all other messages with the purpose to notify only may be sent through e-mail.

Drawing

In model-driven development the source code usually is not written, but drawn. We are interested in collaborative drawing like it is done in Camel (see Section 4.4), namely concurrently. The opposite would be provided by for example Picbreeder (see Section 4.2), where the collaboration is consecutively.

In Camel, only the facilitator can draw UML-diagrams, every other participant can just draw notes and sketches. For conflict resolution and every other process, a drawing canvas, where users can all draw models and make notes and sketches, may be preferred. There all users

actually can partake. The number of participants on a drawing canvas has to be limited, to prevent it from being overloaded and confusing. NetMeeting and the Windows Messenger allow cooperative drawing on a Canvas very similar to Windows Paint. Here sketches and basic shapes are the only possible elements. MetaEdit+ (see Section 4.5) and others offer model elements as drawing shapes only. In any case, a possibility to draw sketches is a very desirable addition to a system designed for drawing UML models.

Automatically converting sketches into UML models is not part of Camel, but it might be developed in that direction in the future [24]. Ideogramic UML (see Section 4.4) does include this feature.

There are numerous possibilities concerning the drawing features. Different tools provide different predefined shapes. These are model elements and basic geometric shapes. Textfields, freehand-pencils and others may be an extension to those. Draw-Together (see Section 4.2) offers drawing in layers. In SLiM, a tool to draw models, there is a *highlighter* (cf. Figure 4.14), allowing the user to highlight certain parts in a signal colour like yellow.

Talking

Mostly used for audio communication is the common phone. Regardless of the other communication means used, whether integrated into larger systems or not, the phone has a place in every environment. Participants in a study concerning video conferencing [104] felt that they did use the phone much less, because it was substituted by video conferences. In reality, the number of conversations over the phone nearly stayed the same. Actually only longer phone conferences were substituted.

Over the internet, there are different tools for voice over IP (VoIP) communication. Studies on video and audio conferencing [104] found that with speakers and open microphones used, a considerable amount of audio echo was produced. Still, the offered headsets were found to be too bothersome to use. For most other messages, whether short or extensive, a kind of e-mail or instant message seems to be more convenient. Also, stored written messages can be reviewed much easier than audio recording. General meetings are best held in a room, with people attending and concentrating on the subject discussed. If not possible because participating members reside in different buildings, video conferencing is preferable to audio only, for the latter is rather impersonal and may lead to split concentration.

In many situations the conversation itself is a supplement and additional resources are needed on the screen, and in other cases audio only brings disadvantages and therefore written messages or conversation via video is preferred. The situations left would be to give someone a quick message, which does not have to be recorded but requires an immediate answer, in this case the common telephone may be best. Sometimes, namely in the just described situation, audio conversation can solve problems faster. When setting priorities concerning the integration of features into an existing system, voice over IP is not among the first choices.

The game WoW (see Section 4.6) was designed with chat as main communication. A lot of users desired additional audio conversation, and therefore relied on software like TeamSpeak (see Section 4.6). We have to keep in mind, that a desired feature not integrated in the system

may be substituted with external software, which would take up time and complicate processes. Thus we suggest that after integrating a feature for video conferences, which is especially justified in distributed software development, a slim audio only feature is next on the list. Preferable these two features are combined to one, where users can select whether they want to activate the visual component or not. This way it is realised in popular VoIP tools—for communication purposes only—like Skype (see Section 4.1).

Audio only is preferred in situations where on one hand audio conversation is desirable, but the screen is for example needed for collective drawing. As discussed earlier, concentration on visual input is limited (see Section 4.1), and therefore the visual input should be limited too. But additional visual information concerning the audio conversation is vital. Using the example above, in the WoW-TeamSpeak combination TeamSpeak Overlay is used to get this information. This information, which users participate in the conversation, is vital, but should not take up too much room on the screen, and therefore should be displayed in a slim window during the conversation.

A huge disadvantage of audio only is the timing. There is no way to anticipate when someone starts to talk, which leads to audio collisions or users staying silent to prohibit a collision. Tang et al. [104] found in their studies, that video can be valuable to help with this problem, for gestures can be used to take or give a turn of talk. These gestures were used much as in face-to-face communication. Another disadvantage of an audio only phone conference, can be that the participants are not aware of who is participating. When adding TeamSpeak to the game World of Warcraft, this problem existed and was solved by TeamSpeak Overlay, as mentioned before, allowing the user to see who is participating at the moment.

In ICQ (see Section 4.1) every message comes with a different audio signal. Users accustomed to these signals, can anticipate the sort of incoming message by the sound it triggers. The audio signals in ICQ can be highly customized. They can be turned on or off individually. Messages with sound may indicate the importance of a message, for example the system's important warnings may be delivered with an audio warning signal.

Watching

"Watching" does not only include video communication, where participants can attend by merely watching too, but also any kind of screen sharing.

Video Conferences

Video conferences are valuable, they offer the possibility to see a remote team member. Video can be used for two or more persons conversing, thereby replacing a normal meeting. This is necessary if there are no alternatives, namely no possibilities for a mandatory meeting to be held in one room. The comfort of interaction is decreased without the possibility to establish direct eye-contact due to the camera's position [92, 104]. Technical improvements allow users to have direct eye-contact, but most used cameras have not taken advantage of this development yet.

Using video for communication purposes next to regularly held meetings can be selectively prearranged or spontaneous. Skype [99] offers free telephoning with video over the internet.

Like in ICQ, users create profiles. ICQ and Skype both offer chat, audio and video conversation. Only the former is mainly used for typed conversation, whereas the latter is preferred for telephoning with or without video.

When calling someone with Skype, the name on the contact list has to be chosen, and a double click on the name or to hit a simple green button starts the communication. Easy enough to use, Skype especially comes in handy for calls outside the country. A disadvantage of video conferencing is, as described in the last section, that either a headset has to be used, or when using speakers, non-participating colleagues in the office may be disturbed and information is given away not meant for other ears.

Offering video conferencing as an every day communication feature can be beneficial, but is highly dependent on the circumstances. Two users in the same office may be reluctant to use video conferencing, but satisfied with audio conversation, since they could see each other during the breaks. Not knowing each other too well, being in different buildings or cities, users may on the other hand prefer to see each other while conversing, thereby improving the collaboration. Whether users decide on using video for visual communication may highly depend on their relationships and their location. If conversing over video indeed is used as everyday communication, other details need to be considered. Video usually takes up a big part of the screen. If there are enough other visual entities the user has to concentrate on, at least the number of simultaneously used video channels has to be reduced to a minimum [92].

Video conferencing can, depending on the situation, advance or hinder collaboration. Therefore, in our opinion, the best solution is to offer conversation through video but on no account make it the primary way of conversing. Or, if offered, constrain video conferencing to situations where it is necessary, like meetings which could not be held otherwise.

Studies on Video Conferencing

Tang et al. [104] conducted several studies concerning video conferences. Two of these will be discussed here. The first one was a survey on users' opinions on the use of an existing video conference room system, the second was a comparison of video, audio and face-to-face communication in a distributed group.

Study A: Being provided with several video conferencing rooms, the filled in questionnaires clearly indicated three shortcomings. Firstly, the availability of the rooms was too limited, hence the difficulty in scheduling a room, which was no technical but a problem of use. Second, video and audio lacked quality, with a noticeable audio delay—more than half a second. Third, users requested a shared drawing space. Users on one hand seemed to mind the audio delay more than the poor video quality, but with the latter, users could for example not see materials used in presentations. The positive response included that the users liked having regular visual contact with remote team members, which saved time and travel in the users' perceptions.

These studies were conducted in 1992. Now that VoIP is standard for private and business use, the quality is not an issue any more. With Skype [99] one often has a better audio quality to the other side of the globe than with common mobile phones within one country. Still, this study shows two important things. Firstly, a lack of quality has a huge impact. Secondly, video conferencing may be beneficial in certain situations, but for everyday communication it is insufficient. Most desired was an additional shared drawing surface, or the incorporation

of other software such as word processing. A third of the participants also indicated that they would like a larger video screen.

Study B: A distributed group, with all participants residing in the US, using video and phone conferencing with additional weekly actual face-to-face meetings, again found the lack of audio and video quality strongly affected the conversations. Mainly resulting from quality issues, the users had difficulties to make comments, due to the delays they could not see verbal openings coming or coordinate eye gazes. The additional face-to-face meetings of the same group provided the conductors of the study with a direct comparison. In video and phone conferences, audio collisions occurred more frequently and were more problematic, and the conversations where rather shaped by monologues. Face-to-face communication is highly aided by gestures. In video conferences the attention is divided between the screen and notes or other things. "Just missed glances" is how the study called the resulting impairment. The worst noticed effect was that due to these shortcomings participants avoided working through conflicts and disagreements.

Next to the already established enhancement of audio and video quality, we identify two ways to improve the situation. Firstly, having your personal notes or a shared space on the screen concurrently leaves the concentration on the screen. Secondly, the limitation of using video conferences rather for presentations—one person speaks while the others watch and listen, or situations where they are a necessity.

Tang et al. [104] analysed the advantages and disadvantages of face-to-face meetings, video conference rooms, and desktop conferences in comparison. The rules of politeness seemed to be the same in any case. The biggest differences between the former two and the latter one is spontaneous/prearranged interaction, and the users' access to personal resources. These include resources of their own workstation and office like books, as well as telephone, e-mail and others. On one hand, resources can be useful for the discussion, on the other hand, users can attend to personal work in between a conversation, like checking e-mails. This can be an advantage, but users were often annoyed by the distraction of others. Attending to additional work was facilitated by the fact that the lack of establishing direct eye-contact made the users feel sort of detached. In a telephone conversation that would not be possible, for the team member on the other end of the line would have no clue of what you were doing, while with video support it is visible.

Shared Whiteboards and Screens

On a whiteboard users can draw sketches and notes free-hand and thereby additionally share not yet completely developed ideas. A whiteboard can combine the possibility to draw sketches and make notes with the drawing of models like UML-models—nicely done by the developers of Camel [24]. In Camel, only the facilitator can draw UML models, while all other participants can only draw sketches and make notes, and partake in the chat communication. This limitation has the advantage, that not several persons simultaneously change a model, causing confusion. Regarding the space needed for drawing, Camel offers a zoom-option and a birdview (overview) window. NetMeeting (see Section 4.3) also provides a zoom-option, allowing several pages.

At any time a page can be added and one can easily thumb through these. The realisation of a whiteboard in model-driven development—Camel concludes at the picture itself—can be done by taking on the models and storing the additional information, notes and sketches separately.

A whiteboard like this is a form of screen sharing. Screen sharing is by definition the collective use of the screen, may it be actively or passively. Active use of screen sharing would be more than one person using the screen at the same time for writing or drawing. When passively sharing the screen, a user may only watch someone else write or draw. This would be useful for example to provide team members with an update. Also, if the number of participants exceeds a certain number, and therefore not everyone can be included in the process of designing but is still required to participate, the participation can be limited to watching only, or a combination like watching the drawing canvas, but be able to participate in an additional chat function.

IMPROMPTUs (see Section 4.3) main features are to show and share off-the-shelf applications. There is a passive, view-only mode, and an active mode, allowing the group to interact with the shared content. While conducting studies concerning IMPROMPTUs usability, the authors found that both modes were necessary, for the sense of ownership for the information provided by a person leads to this necessity.

Another way to passively share a screen are screen shots. While there is no interaction with the other person's applications, an image of the screen as it is at a certain time is shared. Skype allows the user to share the print of the users screen or a part of it with someone else (cf. Figure 4.3).

5.4 Combining the Features

The right combination of different features is essential, for one feature can make another one obsolete. Features take up a certain part of communication, synchronous or asynchronous, or for example asynchronous communication to schedule synchronous communication. These types must be balanced, so there is no missing part on one hand, and no redundancy on the other hand.

The possibilities of combination are infinite, as well as the realisation. One could integrate any possibly desired feature and trust on the end users to combine them regarding the situation. A better way might be to provide a set of key features set by default for certain situations, and the possibility to customise the system by adding others.

Synchronous and asynchronous communication. When combining different features, it is important to offer synchronous as well as asynchronous communication. Quilt [45] offers asynchronous collaboration only, but problem solving is more efficient when done synchronously. The back and forth of asynchronous communication like e-mail or comments within the core product takes up a lot of time, which may frustrate the participants. In other fields this kind of collaboration is favourable. For example in Wikipedia, this way of communicating is the only one possible, for the users are not reachable all day long. Synchronous communication like desktop conferencing may significantly lower the number of e-mails by substituting whole e-mail cycles by one video conference [104].

Spontanous and prearranged communication. Being only provided with concurrent communication must lead to substitution. If video conferencing is used, the conferences have to be scheduled. The designated participants can not be reachable at any time, so the scheduling process might be done on the phone, one by one, or per e-mail. Various e-mail clients are combined with a calendar like Microsoft Outlook, seperate calendars are Mozilla Sunbird or Microsofts calendar included in Windows Vista. The .ics-files can be used platform-independently.

Missing features. Desired, but missing features may either be substituted by other tools or will impact the working process negatively. An example for the first one is the addition of TeamSpeak or others to the game WoW, to additionally provide audio communication (see Section 4.6). An example for inhibition is provided by a study from Tang et al. [104], where users were inhibited due to the lack of a shared whiteboard and shared word processing when offered a video desktop conferencing system.

Studies from Tang et al. [104] indicate that users were reluctant to use video conferencing, for they could not tell in advance, whether a co-worker was available or interruptible. Since the scheduling system by phone did not include answering machines, it could not been done asynchronouslly.

The substitution of means of communication may also occur, when the included feature does not fulfill the requirements. Poor audio quality in a video conferencing system can lead to users turning off the sound and using phones as their audio channel instead [104].

Number of participants. Any communication feature is limited to a certain number of participants. Jazz (see Section 4.5) provides collaborative features for the core developers only. In a situation where a larger group of co-workers have to communicate, certain features are insufficient for the number of participants is limited.

Recapitulatory, the best combination of communication features is to provide the users with synchronous and asynchronous, ad-hoc and prearranged possibilities for communication. These possibilities may include writing and/or drawing as well as audio and/or video.

5.5 Designing the GUI

The design of the user interface is of vital importance. Depending on the combination of features and the circumstances these combinations are intended for, the arrangement in a user interface has to be planned. This is highly dependent on the core task. In applications designed for communication only, like Skype [99] or ICQ [60], the size of the user interface of the application is changeable without limit, still allowing the user to set it to a smaller size, to attend to other tasks during the communication. In WoW, the core task is gaming, and the additional communication feature—chat—can only take up a certain space. Here the changeability is limited.

The user interface has to be flexible to a certain extent. Firstly, the size of windows of certain features have to be changeable according to their immediate importance. More users in a chat, which means more lines per time unit, require a larger chat window to follow the conversation

properly. Secondly, certain features should be possible to turn off, if not required, to provide more space for others.

Different features require different additions. A group chat requires a list of participants. ICQ and Skype (see Section 4.1), Camel (see Section 4.4) or Draw-Together (see Section 4.2) include lists of users, providing awareness of the other participants. A drawing canvas requires preset shapes. Concerning its size, either several pages (whiteboard in Netmeeting) or a possibility to zoom in and out are required, supported by an overview, like the Radar View in Ideogramic UML or the Bird View in Camel.

5.6 Accessing Features

To access certain features, pull-down menus or toolbars may be provided, as well as shortcuts either with the keyboard or the mouse. A combination of these is preferable, to not overload any of it. When designing a tool, available choices are often stuffed into the main menu. There are different reasons to think about additional options. Firstly, fast addressing of more often used operations is beneficial. Secondly, especially when getting to know the tool, shortcuts are learned easily.

There are different situations where the distinction of options is handy. When a user has to browse through an extensive menu, to access the offered integrated feature, although more practicable but also more time consuming, the user would probably just pick up the phone instead. A system not optimised for quick performance, like one for desktop conferencing, leads to users changing to known external means, as mentioned, the telephone [104].

At different times in the project, the user might want to look up team members, to find out who is responsible or to invite someone to a conversation or process. In the game World of Warcraft, the key “o” opens the friend list.

When drawing models, the geometric shapes needed may be provided through a list next to the drawing canvas at any time of the drawing process, or be provided with the suitable options in form of short menus like pie menus. An example for the first one would be Microsofts Paint, used in NetMeeting (see Section 4.3), arranged at the left side of the screen, or Camel, with the palette on the right side. Ideogramic’s (see Section 4.4) predefined shapes for drawing can be accessed through pie menus.

Double-clicks are highly favoured to access a feature quickly. A double-click on a name starts a chat communication in ICQ [60], opens a conversation window in Skype [99], and starts a private message (whisper) in WoW (see Section 4.6). Available applications often offer short menus with a right-click, adjusted to whatever the right-click is on. At least Microsoft Windows users are very much accustomed to be provided with options by a right mouse click. Hot keys, key combinations, or also keyboard shortcuts, usually combinations with the control-key with Windows and the apple-key (also command-key) with MAC OS, often provide quick access to available features.

5.7 The Team

With few exceptions not individuals but teams develop software, and a supporting tool has to make sure, that the team members do not constrain each other at their work [75]. Such teams necessarily include a range of members, with different areas of expertise, experience and knowledge of the domain [114]. Also, next to the different skills and levels of expertise, the members of a team have different beliefs, attitudes, and motivations. These varieties should not be underestimated, for they influence the productivity of a team.

The originators of Quilt [45] (see Section4.2), a collaborative writing tool, found that coordinating activities, jointly supervising each other and sharing information depends on the kind of interpersonal relationship, the social roles, and the stage of the project. In education, students or assistants reviewing a document, would rather comment on it than change it. Peers would revise and change each others texts, without permission, but with an unequal social status proprietary rights over text are maintained.

Knowing Each Other

Communication improves when the members of a team know each other. In distributed software engineering, when a lot of people located at distant places are involved, this can not be the case. Especially in distributed SE it is important, to offer ways to improve acquaintance, for example by offering profiles. The key word here is the inhibition threshold. Team members, who do not know each other personally or at all, may be reluctant to communicate. At least the preferred way of communication changes, which means the users acceptance of certain ways of communication is reduced.

The communication features in the game World of Warcraft are designed to learn to feel at ease with strangers [81]. This is accomplished by a chat feature. In this case impersonal contact. The situation is different from a situation in a professional environment, there are no responsibilities and the contact can be ended any time in the game. Added audio contact (TeamSpeak) would be used later, when a group already knows each other.

Profiles

The originator of a message specifies the addressees by name and therefore by number of addressees. This specification requires a list of possible targets, which could be a list of e-mail addresses, names, or nick names. Profiles of colleagues, together forming a list of them, offer the possibility to find out who else should be informed of certain matters.

Contact lists, consisting of names, offered by tools like ICQ and Skype, can be divided in groups and sorted by different criteria. A right-click on a name gives the option to view a profile. These tools also offer a status attached to the profile. The status is visible at first glance by different icons next to the names and the appearance of the names. Next to the status offline, different choices are offered when online, like busy, not available, or invisible.

In Jazz (see Section 4.5) the status is provided on one hand by the colouring of the picture, indicating if a person is online. On the other hand, when hovering over a picture, details typed in by the user are visible.

ICQ offers to write a note concerning details to your status, which is displayed within the contact list next to the name, visible to see for everyone. In the user interface of the user announcing his/her status, this message is typed in and shown on top, next to name and picture, see Figure 5.1.

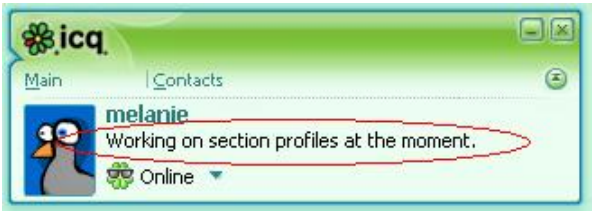


Figure 5.1: Write a note about your status in ICQ

In ICQ [60], mainly used for chatting but comprising other features as well, the users fill out a profile, including mandatory and optional information. In a professional environment, when a company requires their employees to set up profiles, private information like hobbies would not be required, but optional information may include skills. The company may also exclude optional information to be filled in, and comprise the user profile of mandatory lines only. Imagine a larger project team, where the members do not know each other personally or well, with a list of profiles it is easy to find out who or who else is responsible for certain activities. With an additional option to search for specifics, a programmer could for example send a message to all team members occupied with testing without knowing their names by heart. Knowing the team members improves collaboration, and a nice accessory feature would be to be able to browse through the profiles.

In World of Warcraft, every group has a group leader, the one who opens a group and invites others to join. This person has certain rights, like inviting or excluding team members. The position may be handed over and thereby all rights transferred to another person. This is not temporarily, like in the previously described tool Camel (see Section 4.4). Anyone starting the game can either be a group leader and assemble ones own team, or join an already existing team. In our reality, the “group leader” has to be selected by other criteria. Portraits can either be supervised by an administrator, who is responsible for creating and updating them, or by the single user. In our opinion an approach including both is best. On the one hand this administrator is responsible for the creation or deletion as well as mandatory entries of the profiles, while additional entries can be made by the user itself.

Skype [99] offers the possibility to send contact information of a third person, much like a business card. The other person does not have to look for the third persons profile, but is provided with the contact information.

Authorization and Responsibility

Next to the degree of acquaintance, the individual's position affects the collaboration. In a project at a University, including students, assistants and professors (three clear positions), a student would limit collaborative activities to questions and remarks, while the professor has the authority to implement changes any time—without asking.

This example provides us with a clearly settled chain of commands. In the private sector, however, different ways of hierarchy are established and thereby the rules are defined. Teamwork can be based on authority or on discussion and consensus. The former is accomplished by a steep hierarchy and the latter by a shallow one. The type of command structure depends on the structure of the company or is just a matter of taste. Nevertheless, it influences the collaborative process.

Policies and rules for communication have to be set. How these rules are chosen is affected by the teams hierarchy and the means used to communicate among others. These policies state authority and responsibilities. A responsibility in communication is for example that the addressee of a message is responsible to answer in time. An exemplary authority would be the authority to send out invitations to meetings—not just anyone may be permitted to arrange meetings for other people.

In their studies on collaborative software engineering, Favela et al. [43] found, without establishing these rules, the process is impeded. While holding their class, it became apparent, that no standard expectation existed for many of the collaborative processes. As a result, meetings scheduled through e-mail were missed, and e-mail messages requiring answers were not answered for days. To resolve these issues, which led to delays in work and especially discontentment, a "team contract" was developed and established. The contract included certain rules on communication, response time, and authority. An important rule figured out, was that e-mail should be answered and expected absences had to be reported within 24 hours. Although this team contract did not avert the problems discussed above, the frequency was reduced distinctly.

There are two ways to get a team member to address oneself to a task in time. Either set policies, which—when followed—assure the timely attention, or force the attention by choosing the means of communication. That is, instead of sending an e-mail, to send an event notification popping up at the addressee's screen, informing him about the importance of his/her immediate attention. Also synchronous, instead of asynchronous communication, facilitates a quick reply.

It is often not possible to share the same authorisations. For example in Camel (see Section 4.4), only the facilitator of a session can draw UML elements, while the other participating users can only draw in free-hand mode. In this case the restriction is to prevent chaos on the canvas, true to the motto "too many cooks spoil the broth".

The Number of Participants

For certain means of collaboration a certain number of participants is desirable. For most features applies, that the smaller the group, the more convenient is the use. If a certain number of

participants is exceeded, features like chat, video, and audio conferencing become confusing. To many lines in a chat make it harder to follow the conversation. The more people join in an audio conversation, the more likely it is, that people talk at the same time. Therefore the number of simultaneously participating users has to be limited in the first place. For example a drawing canvas is only suitable for a smaller group, otherwise it would be chaos. A solution would be to allow a certain number of users to take part in the drawing, with the additional possibility to invite more users to just watch. Depending on the kind of invitation, users are provided with authorisations—to draw, to take part in an enclosed chat or to just watch.

For larger groups, fitting best are either asynchronous means of communication, like e-mail and notifications, or a limitation to certain possibilities, like watching only in video conferencing. The project Jazz (see Section 4.5) solves the problem by providing means of collaboration only for the core team of developers. No matter how distributed or large the team is, the collaborating parties are always limited to a certain number.

In a project team consisting of several people, the collaboration is done in smaller groups, depending on the situation. The features formerly mentioned therefore are of course suitable for a large development team too, and any message concerning the whole team might be sent asynchronous or in form of a one-way presentation. Anyway, it is necessary to provide collaborative options for smaller and bigger groups.

General Help Features and Private Use

Private use here describes the use in a personal manner, unlike the private—not visible to everyone—messages described before. In their studies, Favela et al. [43], found their system was used privately too, mainly for sharing music. When conducting studies concerning video conferencing, Tang et al. [104] were asked by the participants to erase one of the videos, which contained the discussion of a sensitive personnel issue, which would usually have been discussed face-to-face, but here by using desktop conferencing, because it was available.

There is private communication in a company, and a system which strives to form the whole workplace, by providing means for the main chores and collaboration, could as well include the possibility for anything else. Person-to-person notifications could be marked as private. These would, if at all, only be saved in a special folder. For example, a user wants to arrange a lunch date, he or she would not have to use an external communication device. Also, if common e-mails are replaced by an integrated feature, the invitation for the Christmas party could be sent with a private notification.

Next to private messages, questions not related to the project at all, may occur. For example a question concerning the use of the tool, or discussion about arranged policies. Bulletin boards can take on this function. Those divided into categories, allow the users to review already asked questions, or discuss issues with no direct relation to the actual work. In Wikipedia (see Section 4.2) every page has a potential talk page, and the “village pump” a bulletin board for general discussions.

5.8 Global Software Development

The increase in inter-organizational cooperation leads to teams and communities communicating across organizational boundaries, and therefore to the need of standardization and integration of collaboration tools in a collaboration middleware [87]. The same applies to a geographically distributed organisation, where teams are formed according to location. Current technologies like phones, e-mails, and video conference rooms do not satisfy the collaborative support needed across geographical distances [104]. The groups are not only divided geographically, but also by language barriers and cultural aspects [51], as well as professional and educational aspects [38].

Depending on the location, different means of communication are used. Most geographically distributed communication is currently accomplished by e-mail [87]. Tang et al. [104] found in their studies, that team members in one location rarely used e-mail, except when they tried to avoid personal contact, while between the different locations e-mail was used heavily. This was especially because of the time difference, which made phone calls more difficult. E-mails were not the preferred way of communication, except for sending files, but the easiest one. Teams in one location would often use desktop conferences instead of meeting face-to-face. This indicates, that even a short distance is still a distance. Different choices have to be offered, according to the geographical distribution. The first decision in distributed software engineering is the selection of the supporting collaboration environment. The different teams—which together build the whole organisation—are usually already used to certain ways of communication [87].

Distributed SE opens the possibility to work around the clock. Regarding synchronous and asynchronous communication, in global software engineering the latter may be the only available choice through different time zones. Assuming a distributed SE project, which includes teams in different time zones. There may be some teams in one location and others at a distance, in a different time zone. Therefore within one location collaboration may include concurrent communication, supported for example with an option to chat. If there is a time difference between teams, communicating at the same time is restricted to mandatory activities, which are scheduled up front, this could include video conferences. The other part of the communication would be asynchronous, accomplished by e-mail or other messages, stored and answered later. In distributed software development the collaboration has to be adapted, and more effort concerning team building has to be made.

CHAPTER 6

Building Collaboration into a Work Environment

The idea for this thesis was born out of the paper "We can work it out: Collaborative Conflict Resolution in Model Versioning" [19]. The authors propose to expand the project AMOR[1] by a component called the Collaborative Conflict Resolver (CCR), which allows—as the name implies—collaborative conflict resolution. Whereas this paper proposes enhanced collaborative features in model versioning from a conceptional point of view only, concrete technologies have not been proposed. This thesis provides detailed suggestions for such a component in the adjacent chapter, tailored to a model versioning system like AMOR. This chapter discusses appliance and usability of the general results presented in the previous chapter and introduces AMOR.

6.1 How to Support Collaboration

Admittedly, our results from the previous chapter are rather general, but this is due to the repeatedly mentioned fact, that efficient collaborative support has to be specifically tailored to the team and their working environment, including for example how well the team knows each other and if the environment is a distributed environment.

We explain the importance of actively shaping collaboration and present basics of teamwork in the adjacent chapters. The basics include for example the fact that a great deal of the communication is rather ad-hoc than prearranged, and that synchronous and asynchronous communication is part of any collaboration. To efficiently further collaboration these basic facts have to be considered. Social aspects are, if at all, considered only in experiments regarding specific systems or in educational environments, although they massively influence the usability of the collaborative support.

[1] http://www.modelversioning.org

This thesis provides a list of means and methods to collaborate, we deduced from a selection of systems and ideas described in detail. This *list* contains the following.

- **The possibilities.** Chapter 5 contains a summarisation of possibilities. These possibilities include a list of means for communication.
- **What to use.** Different means and methods of collaboration have advantages or disadvantages for different teams in different situations. The important question is, for *whom exactly* do we want to provide support? Before designing collaborative support, the current and the desired state has to be determined. To further the choice of support, we provide information about relevant social aspects, for example which role the relationships between team members play. We describe aspects to be minded for choosing and combining the features.
- **How to use.** When the way to support collaboration has been chosen, there are numerous ways to apply this support. Again, the specific appliance depends on the target group. If for example a chat feature is the tool support of choice, the chat has to be adjusted to the number of participants per chat session and on the situations the chat is intended for. If the chat is for common messages only, these should be displayed. A chat feature which is otherwise used in specific situations where additional visual input is required and users want to be informed of activities carried out by team members, the chat may display these activities too. Which information is displayed in a chat window is just one of the aspects to be considered when integrating this feature.

We describe the use of the given possibilities with the help of some specific examples in the following.

Applying the Results

To illustrate the applicability of the results of the former chapter, we illustrate them with different scenarios. We introduce several software engineering teams working in model-driven development. Every team has specific goals and habits. We describe the starting position—what is and what is desired—and propose a solution to further collaboration for every team.

Scenario A

The communication in team A is not mainly restricted to certain parts of the team. Nearly all of the team members communicate with each other on a daily basis. For the mainly ad-hoc communication they primarily use e-mail, the phone, and talk to each other face-to-face, which often requires them to leave their working station and walk to the next room. Some of the team members started using Skype to call and chat, but the rest of the team does not want to join this public tool. Team A is altogether fed up with the simultaneous use of several tools and finds fault within the traceability of discussions in these tools. With their modeling tool however, the team is highly satisfied.

Solution. A customised instant messaging (IM) system provides the team with asynchronous and synchronous instant messaging at any time. The IM system includes a user list with user profiles, and is linked to other used tools. Therefore, the users can for example reference a part of a model created in the modeling tool, or send files. Messages can be sent to one person or a specific group defined within the user list, they can be prioritised, public or private, and colours are assigned to users in every session. The tool is customised and accessible only for the team. A history log provides easy access to past messages.

Scenario B

The work of team B is very much customer-oriented. Several commissioned works are done simultaneously by parts of the team. Meetings with customers are scheduled for every day. The team finds it difficult to arrange the meetings for the rooms are limited. The main problem in communication is between the customers—i.e. the future users—and the different sets of experts.

Solution. To solve the problem with room reservation, a joint calendar is set up. Schedules can be looked up by user name and by room. Viewing a rooms schedule allows the user to see when the room is free, and who otherwise reserved it for a meeting. Again a user list has to be set up, providing among others contact information for the users. One modeling (drawing) tool for the whole group allows different levels of completeness in models. A sketch of the requirements created by customer and designer together in a meeting may be, all with the same tool, further developed by the designer to reach the level of completeness required for the programmer to implement. The drawing tool therefore has to handle freehand drawing, drawing of different shapes, or complete models, which then are used for model-driven development. A model created within a joint meeting includes a link to the calendar, showing date and time of creation.

Scenario C

Team C is distributed to three locations. Since face-to-face meetings are often difficult to arrange, parts of the group gather in meeting rooms in the different locations to hold phone conferences. The team criticizes that they have to bring a lot of documents to these meetings and are not able to show them to the team members on the phone. Using and being linked together by one common system for all persons is not possible, for the co-workers in the different locations use different applications, and the Microsoft Office Word documents are of no use to the Mac Pages users in the other location. Still, non of the teams want to change the used applications. The team is fed up with creating and sending pdf-documents back and forth per e-mail.

Solution. A common tool, including a common file repository, creates a framework around existing applications and allows desktop sharing and video conferencing. Through the shared desktops, files created in certain applications can be viewed by users not using that application. If files usually have to be worked on by users using different applications at the moment, the team simply has to agree on a common tool, or otherwise make friends with copy and paste. As in the previous examples, a list of users provides a list of potential adressees. Video conferencing—being able to see the person opposite—is more personal than conversations on the phone, and

allows the users to stay at their working station, hence providing access to materials during conversing.

Creating Efficient Collaboration

The following section summarizes our results into a plan of action. To achieve proper and full collaborative support, these are the steps to be taken.

Step 1: Primary Questions

At first, some questions are subject to clarification. The first questions to be asked are about the potential users.

- For whom is the collaborative support intended for?
- Is the support intended for a specific field of work, company, team, etc.?
- For how many users is the collaborative support intended for?
- Is the collaborative support part of a newly created system, built into an existing system, or a stand-alone application?
- Is collaborative support intended for the whole SE life-cycle, or specific situations only?

The more specifically the collaborative support has to be tailored to a certain field of work or even a certain team, the clearer the requirements can be described, and the better the support fits. A too general approach will not be convenient for everyone, and therefore maybe not for anyone. The number of simultaneous collaborating users does not have to equal the number of team members. It may be sufficient to provide collaborative support to a core team only.

Step 2: General Considerations

The next step is to consider general technical and social aspects, which influence the collaborative process. Again, the more specific the target group was selected, the better the following questions can be answered, and the more accurate the requirements can be deduced from those questions. Step 2 provides a list of general aspects to be considered, which may, if known, be included into the planning.

- *Existing tool support.* Existing tool support has to be considered, if the collaborative support is planned to be integrated in an existing system or used additionally to other tools. A specifically tailored external collaborative support tool may provide links to the other tools used, or create a framework around them.

- *Version control is required.* In software engineering, a version control tool allows concurrent work, and therefore facilitates teamwork. Since pessimistic versioning restricts teamwork, the optimistic approach is preferable. There also is the possibility to provide an optional locking mechanism in optimistic versioning, which may help to avoid conflict situations.

- *Comments in source code and models.* Sufficiently commenting source code or models creates clarity and awareness among the other users, and therefore is important not only for reasons of traceability.

- *Cultural aspects.* The culture of the potential users defines how they prefer to collaborate and communicate. Cultural aspects are not to be underestimated. For example, in different cultures it may be more or less important to see each other face-to-face, so that one party may be offended by written communication when video conferencing would be polite. In distributed software development, different cultures come together.

- *Organisational structure and culture.* The organisational structure, for example the hierarchy, sets roles and therewith authorisations and responsibilities. The organisational culture, much defined by cultural aspects, defines rules and methods on how collaboration is accomplished in a team.

- *Roles.* Depending on the target group, which different experts are involved? A software development team may consist of designers, programmers, analysts, and others. Hence, inter-expert communication is required to provide possibilities to display information in a way, that is understood on both sides.

Social awareness. To work together efficiently, it is important to know whom to talk to and their availability for interruption. Co-workers can be found in a user list, profiles show responsibilities, and status describes availability. User lists and profiles often are a necessary supplement for certain communication features. Especially in a large group or when the users are not very much acquainted, theses features help them to get to know each other.

Step 3: Spoilt for Choice

When the primary questions and general considerations are crossed off the list, the actual planning begins. The possibilities constitute a long list, including the different means of communication like textual, audio, and video communication, or collaborative drawing. More specifically these range from different kinds of instant messaging to desktop sharing and so on. Then there are the different ways to implement these possibilities. We are clearly spoilt for choice.

Although the first two steps provide us with information on which features may be suitable, still the choice and combination of those is dependent on other criteria too. In the following, we describe aspects which have to be considered for choice and combination of efficient collaborative support.

- Collaborative activities, which have to be integrated into planning, include, next to the collaborative execution of tasks, the coordination of these tasks, communication, and coordination of communication.
- Synchronous and asynchronous ways to collaborate and communicate are required.
- Spontaneous and prearranged collaborative activities have to be supported.
- Prearranged collaborative activities have to be scheduled.
- The suitable collaborative features have to be accessible at any time, more frequently used features may be accessed in an easier manner than others, through short-cuts for example.
- When combining different features, there should be no redundancies and no missing features.

As mentioned in step 1, the collaborative support has to be tailored to the field of work. Is this field of work model-driven engineering, providing common drawing may be beneficial. Consecutive (asynchronous) work on a drawing is easily accomplished by the use of a common tool, also means and rules to comment the drawing are required (cf. step 2). Concurrent (synchronous) drawing requires a shared drawing canvas, and a way to additionally communicate, either textual or audio/video. This applies to drawing in general and to drawing models for model-driven development.

To regard only parts of the given guideline may still lead to an overall improvement of the process, for supporting collaboration is beneficial in any case. A tool or feature which would hinder collaboration will most likely not be used. Missing collaborative support, for example providing asynchronous communication only and no synchronous communication possibilities, leads to less efficient collaborative support, for the missing features will definitely be substituted by external tools. Redundant features, like several features for synchronous or asynchronous instant messaging, may either be used not at all, or the different features are preferred by different users, and there is no *common* tool at all.

If all mentioned items in the guideline are considered, full and efficient collaborative support will be achieved.

In the Real World

The studies concerning collaboration discussed in this thesis were conducted in different fields and have very different, partly overlapping, partly contradicting results. This is because the studies concentrate on different elements under limited circumstances. Adelsons [1] negotiation system is beneficial under the right circumstances, and the acceptance of the VideoViewer project [92] was influenced by the novelty of the approach for the test group, to name two examples. The only consistent result is that enhancing collaboration is worth the effort.

When designing a system, interviewing potential users helps to define goals. The questionnaire[2] concerning versioning of models in practice was created by the AMOR project team to

[2] http://www.modelversioning.org/fragebogen

gain insight into different opinions of potential users. This is the first step to adjust the project to real-world conditions. Later on, pilot studies, for instance in student environments, are beneficial, but offer preliminary insight only (cf. Section 4.5). Only after the realisation can real world tests show the real usability and acceptance. Then, there is no "laboratory effect", and needed adjustments and extensions can be implemented.

6.2 Adaptable Model Versioning

The project AMOR (Adaptable Model Versioning), a cooperative research project, aims to improve versioning mechanisms for modelling languages [7]. AMOR is realised by the Business Informatics Group[3] (Vienna University of Technology) and the Department of Telecooperation[4] (Johannes Kepler University Linz), in cooperation with SparxSystems[5].

A first vision of AMOR is presented in "AMOR—Towards Adaptable Model Versioning" [4]. Like other VCSs, AMOR is divided into a front-end and a back-end part. The front-end part is referred to as *Versioning Assistant*, which supports the user in versioning, but leaves the resolution of potential conflicts as the most challenging task. The project team identifies three problems in model versioning, and these shortcomings in current practices and tools are addressed by the project. The focus of the research lies in precise detection and intelligent resolution support of conflicts (back-end), and proper and generic versioning support.

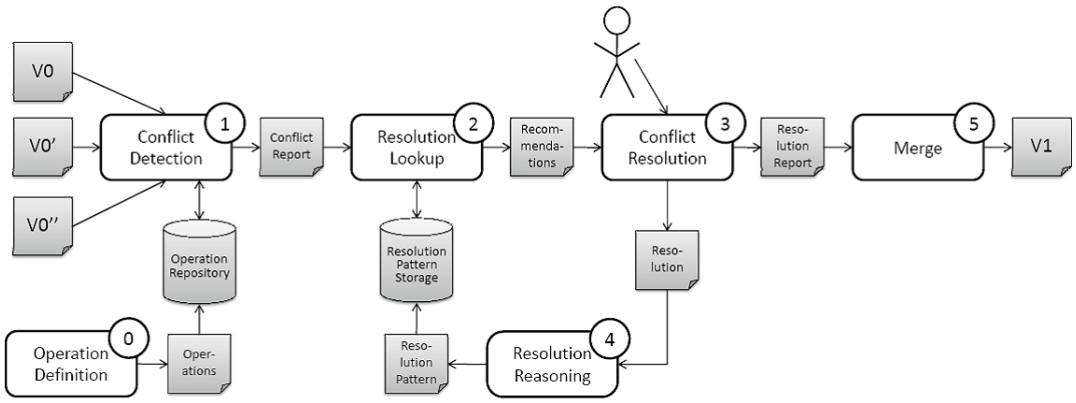


Figure 6.1: Conflict detection and resolution in AMOR [4]

Conflict detection. The graph-based structure of models makes conflict detection in model versioning more difficult than it is in line-oriented text comparison of arbitrary artifacts. In AMOR, the structure and semantics of models is considered, also generic and language-specific aspects are supported, leading to more precise conflict detection.

[3] http://www.big.tuwien.ac.at
[4] http://www.tk.uni-linz.ac.at
[5] http://www.sparxsystems.at

Conflict resolution. Research and advances in model versioning systems often focus on correctly detected conflicts, while the resolution of conflicts is left to the user. First, there is hardly any dedicated support from recent VCSs concerning resolution support for detected conflicts. Second, any tool-supported resolution process is effectively a single-user task, and benefits of collaborative conflict resolution may only be gained by resolving a conflict manually, with pen and paper. Third, the same or a similar conflict may occur again, but has to be solved repeatedly from scratch.

In AMOR, the *Resolution Recommender* offers resolution recommendations, patterns containing a conflict description and an executable operation pattern, which are stored in the *Resolution Pattern Storage*, to the modeller in charge. Manually, the conflict may be resolved collaboratively. Collaborative support is provided by the *Collaborative Conflict Resolver* (cf. Section 6.2).

Language-independent versioning support. Currently available model versioning tools either support generic model versioning, hence language-specific aspects are not considered, or are specifically tailored to one modelling language, and therefore not applicable for other languages. AMOR provides a generic framework with extension points for including language-specific features, thereby combining the advantages of generic and language-specific VCSs, and improving conflict detection and conflict resolution.

The workflow of a conflict resolution process with AMOR, as described in [17], is shown in Figure 6.1 and described in the following.

0. Operation Definition. Language-specific knowledge incorporated in the merge process may considerably improve the detection and resolution of conflicts. With the *Operation Recorder* composite, user-defined operations for specific languages, for example refactorings, may easily be specified. These *Operation Specifications* are the basis for a precise conflict detection.

1. Conflict Detection. Input for the merge process are the potentially conflicting working copies intended for the merge (v0' and v0") and their base version (v0). Next to generic atomic changes, composite operations stored in the *Operation Repository*, may be detected as conflicts.

2. Resolution Lookup. The *Resolution Pattern Storage* is searched for solutions for the reported conflict by the *Resolution Recommender*, which identifies matching and similar resolution recommendations and provides those available in form of patterns to the user.

3. Conflict Resolution. AMOR supports semi-automatic and manual conflict resolution. Also, a collaborative resolution process may be initiated from the user in charge of resolution.

Semi-automatic and manual conflict resolution. Semi-automatic conflict resolution describes the process, where suggested recommendations are selected and then automatically incorporated, in some cases additional user input is required. If no patterns are available, or the offered resolution recommendations are not chosen to be used by the modeler, a manual process resolves the conflict. This can be accomplished by one person solely or in a team.

Collaborative vs. Single User Conflict Resolution. Without resolution recommendations found, the conflict has to be solved manually. Since the manual resolution by a single user may not incorporate the other modellers design intentions, AMOR offers, next to a validation of the

merged version, an extension called *Collaborative Conflict Resolver* (CCR) [19]. The CCR, a component allowing conjoint modelling and providing a communication platform, is discussed in Section 6.2.

4. Resolution Reasoning. The solution of manual conflict resolution processes is analysed by the *Resolution Reasoner* in the background, wherefrom a new pattern is created and stored in the Resolution Pattern Storage, to aid in future resolution processes.

5. Merge. With the *Conflict Report* the consolidated merged version is constructed and stored in the repository.

Several papers related to the project have been published by the AMOR project team [7]. For example, [5,18,20] deal with model versioning, [16,19] with conflict resolution in particular. At the Business Informatics Group, concerning conflict detection and resolution, two other diploma theses have been presented [73, 107].

Enhancing AMOR with Collaborative Features

With optimistic versioning, the developers are not restricted in parallel development due to locks on artefacts, but conflicts may occur through concurrent changes. When working with models, development parts usually can not be strictly separated and assigned to different team members as it is practise in line-oriented software development, thus the probability for conflicts rises. These conflicts usually have to be solved manually by the modellers.

AMORs Resolution Recommender provides the user with resolution strategies for this process [4], thereby facilitating the resolution process. A workflow for such a process is displayed in Figure 6.2. The modellers Harry and Sally concurrently modify a component (time t0). After Sally checked in her Version 1 (time t1), Harrys check-in of his Version 2 (time t2) triggers the comparison of the different versions. If a conflict is detected, the conflict resolution process is started, leading to a new merged–and consistent–version committed to the repository.

Collaborative Conflict Resolution

In optimistic version control, users are usually forced to resolve conflict situations alone. Thereby different, maybe partly overlapping motivations, are not taken into account. The CCR [19] allows an approach to include all conflicting parties. The conflict resolution process is expected to be enhanced by the consolidation of all parties involved. With a consentient found solution the quality of the merged model is improved. This is advantageous on a social level too. The developers acceptance is raised by not disregarding their intentions, thereby avoiding social conflicts and frustration, which alone would constitute a reason to make conflict resolution a collaborative task. A preliminary sketch for the Collaborative Conflict Resolver User Interface (CCR UI) was presented in [19]. Here, Harry and Sally collaboratively try to solve a conflict. They communicate via chat, and possibilities to either chose Harrys, Sallys, or the merged version are given.

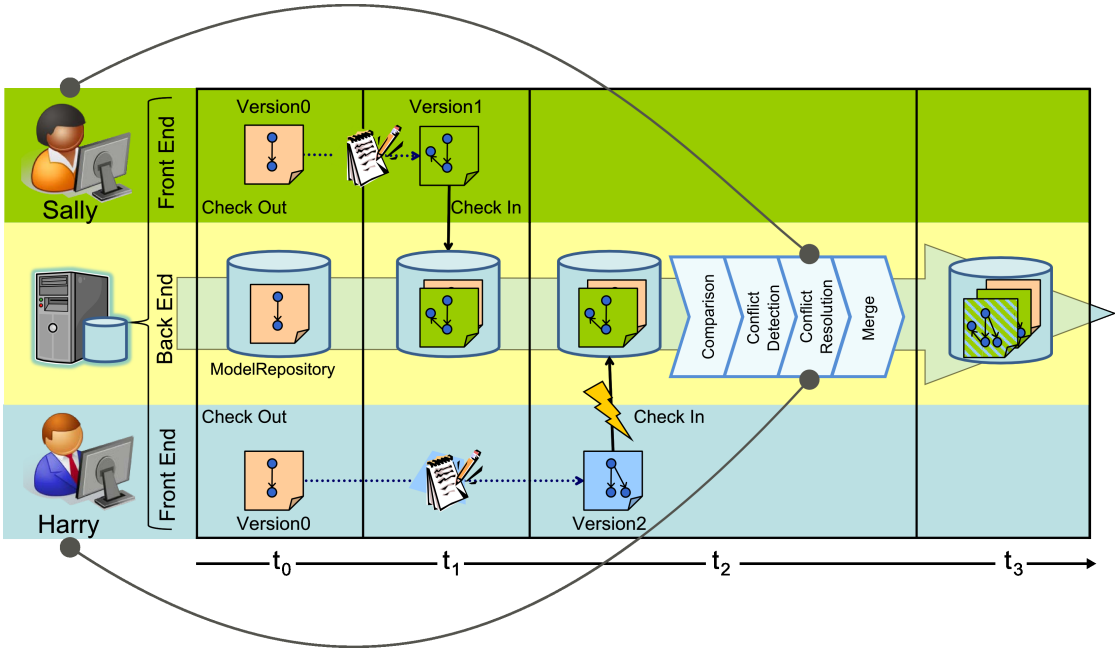


Figure 6.2: Versioning workflow with AMOR [7]

The Collaborative Conflict Resolver (CCR) is started when the collaborative resolution process is induced, notifying the conflicting parties and offering resolution patterns from the Resolution Pattern Storage. Thus, the conflicting parties may start the manual resolution process [19]. The proposed CCR user interface provides the users with the problem description, a drawing canvas, which includes the conflicting versions and the graphically marked problem, as well as adequate shapes for the kind of model in conflict to ease the drawing.

We propose to broaden this approach. Our proposition, the subsequent Chapter describes the approach in detail, includes:

- **Customise communication.** We recommend to change the chat window corresponding to the analysis and comparison of similar chat features presented in Chapters 4 and 5. Supplementary features like a user list and associated profiles are required. To display a users' status is recommended, for the co-workers to be aware of each others availabilities for interruption. An additional drawing canvas—a whiteboard—takes the communication up a notch by communicating ideas in drawing and illustrate the written communication.

- **Provide additional information.** The graphical representation of the base version the conflicting fragments originated from is required to on one hand provide this important information, and on the other hand to be presented with an additional resolution possibility—to go back to the base version. The use of the pattern has to be a choice, and the pattern has to be presented graphically. There may also be more than one suggested

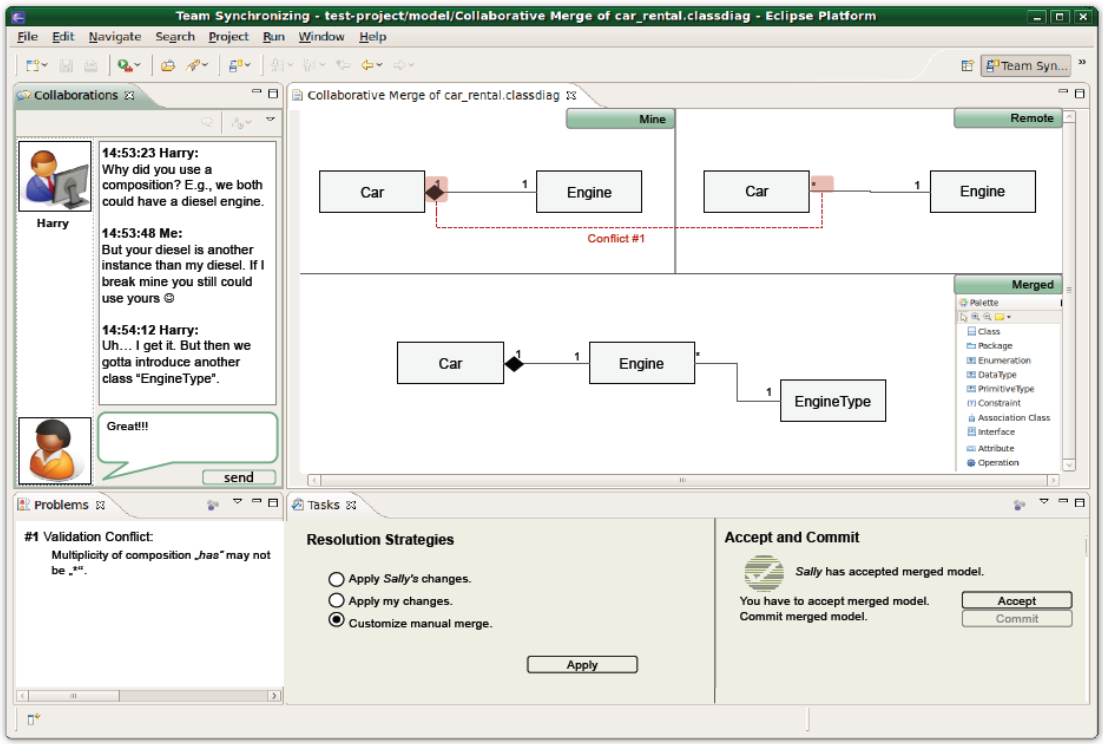


Figure 6.3: Preliminary sketch for the Collaborative Conflict Resolver User Interface (CCR UI) [19]

pattern. Automatically presenting the pattern in the space reserved for finding the solution therefore may be unwise.

- **Offer more solution possibilities.** The possibilities for conflict resolution are to chose one of the conflicting versions over the other, create a new version, or go back to the base version where no conflict existed, to start over from there. We recommend to add the base version to the list of solutions. Collaborative conflict resolution is important, still it may be the case, that one of the conflicting parties is out of reach. For the conflict to be solved anyway, there has to be a possibility for one of the conflicting parties to solve the conflict solely. In another situations a third parties opinion may further the resolution process. With contact lists in place, a third person may be invited to join the process.

- **Extend the possibilities for use.** A component such as the CCR may be used for discussion and without a conflict present. This modified version of the CCR is presented in Section 7.5.

These are the minimum requirements to successfully integrate collaboration into the CCR, therewith the process for the conflict resolution process is enhanced. To further collaboration

even more, the model versioning system and the development environment may be enhanced to a fully collaborative system. There are, of course, numerous ways to integrate more collaborative features, even into the CCR.

A Proposition for Collaborative Support within AMOR

According to the guideline presented in Section 6.1, we assume primary questions are answered and general considerations thought through, and the thereby gained information is available, we discuss some collaborative features recommended to be tailored to and integrated into a model versioning system like AMOR and a component like the CCR in the following.

Supplementary features. A list of users is required for most collaborative features or tools. The already discussed user list set for the collaborative conflict resolution process provides a basis for other collaborative features. Profiles including photos of users are more personal. Those including the time of the users location (cf. Section 4.1) facilitate awareness of availabilities in distributed development.

Communication. The preferred communication feature for the CCR is a chat feature, in [19] and in our approach (see Chapter 7). This textual communication is enhanced by the integration of system messages into the user chat (cf. Section 5.3). We recommend to extend the chat feature in the CCR to a system-wide instant messaging service for AMOR, were different kinds of messages can be sent asynchronously to certain persons or specific groups (cf. Section 4.6). Also, a steady accessible history log is required, to review conversations.

Model-specific support. Models are not textual descriptions but a combination of drawings and text and therefore have to be represented graphically. Creating or changing a model requires a canvas, were the artefacts can be presented, and predefined shapes to draw. Hence, it makes equal sense to use graphic representation to additionally discuss the artefacts in a team, and not only textual—or in face-to-face communication verbal—communication.

Here, a free-hand tool may be used to quickly communicate by drawing, for example to explain an idea, for the restricting required completeness and correctness of models is not given. A manual collaborative conflict resolution process without tool support includes pen and paper. We propose to add the possibilities of the pen-and-paper-method to the tool support.

Developers are often reluctant to sufficiently comment source code. In model-driven development, were comments may be assumed to be less easily integrated into the artefacts in comparison to line-oriented source code, it is even more important to further the commenting of the produced artefacts. This may be accomplished by sticky notes, like they are integrated into the Adobe Acrobat Pro[6]. In traditional source code editors comments stand out since they are coloured in a different way than the rest of the code is. In editors for MDD comments appear as comment nodes, which fit in with the other model elements and are more likely to be overlooked. For that reason and in the case of expected removal of comment, these sticky notes are preferred. Sticky notes are easy to create and remove, are more emphasised graphically, and may be used as comments or messages.

[6] http://www.Adobe.com/Acrobat9

Provide extension mechanisms. Teams have different habits and preferences. Not all potential wishes can be fulfilled, but the possibility may be given by providing hooks and links for external tools.

6.3 Summary

In this chapter, we explain that collaborative support has to be tailored to specific fields of work, the users goals, the size of a team, and other aspects. Afterwards we provide a general guideline on how to integrate suitable collaborative features in a proper way, to successfully facilitate teamwork and therewith improve the SE process.

We then apply our results to the introduced model versioning system AMOR in general, and the conflict resolution process within AMOR in particular. The former includes supplementary features like a user list, and means of communication, both textual and—to specifically support the work with models—graphical. The latter is presented by example and in detail in the following chapter.

CHAPTER 7

The Collaborative Conflict Resolver

While Chapter 5 provided a general overview of general possibilities for collaborative support, this chapter introduces our specific proposal to support collaboration in a conflict situation in model versioning. This proposal is the detailed description of the Collaborative Conflict Resolver (CCR) (see Section 6.2).

A component like the CCR can extend a standard model versioning system like AMOR [4] to support the collaborative conflict resolution process. The features and the user interface are described before the resolution process with the CCR is explained and an example of use is given. Also, we propose to extend the CCR for common—conflict-unrelated—discussion between the modellers.

7.1 The Features

The features of the CCR described in this section are divided into the following groups. The *supportive features* are not exclusive parts of it, but necessary to use other functions within the CCR. The process is strongly facilitated by the *graphical visualisation* of the problem, i.e. the conflict, and the options for resolution. The *Communication window* offers chat and user information, and the *Whiteboard* gives room for additional discussion. Last, but not least, general functions and the *actual resolution of a conflict* with the CCR are described.

Supportive Features

The CCR is a component for a versioning framework and is supplied with resolution patterns—automatically executable suggestions for conflict resolution (cf. Section 6.2). To work with the Collaborative Conflict Resolver, two additional features are necessary. These are not directly visible during a conflict resolution process. First, potential communication partners have to be listed in a form of *contact list*. The names in the list represent the users, which are described

in the profile attached. Second, a *notification system* is required. This notification system informs the users about the conflict and invites them to participate in the resolution session. This notification system can additionally be used for human-human interaction (cf. Section 7.5).

User list and profiles. Every user within the system extended by the CCR has to have a user profile and be in the contact list. This profile includes a current status of the user. The status may be chosen out of a specific range of states, like *available* or *not available*, and can be extended by a personal note describing details. Being part of the user list is required to be invited and to partake in conflict resolution processes. Other users can find out responsibilities, authorisations, and availability through the profiles. For example, in the modified version (see Section 7.5), users can first look up someones current availability before sending out an invitation to join the session.

Notification system. After the conflict is found, and before the CCR can be started, a notification, and with it an invitation, has to be sent to the conflicting parties. This notification system also has to deal with currently absent users. Piling up notifications of a conflict in a users in-box while he or she is on vacation without the possibility for the conflict being resolved is of little use. Here, the formerly described user list containing a current status of each user is needed again, linked to the notification system of course.

We propose to extend the notification system, which informs users of conflicts, to be used by team members to send asynchronous messages additional to the synchronous way of instant messaging provided by the CCR. For example, user A thinks the name of a class given by user B lacks signification. First, there is no conflict, second, there is no need for a longer discussion. User A is only pointing out his/her opinion, leaving potential changes to user B. The advantage of this kind of message compared to e-mail is that the message is within the system, it may include a link to the class in question, relieving user A of describing its location in the e-mail.

Graphical Visualisation for Conflict Resolution

Conflicts, as well as associated information, is graphically visualised to support the resolution process. This graphic representations include the conflicting versions of the model and explicitly the conflict, the base version, and possible resolution patterns. Next to the textual description of the problem, the conflict is marked within the visualisation of the contradicting versions. By colouring the conflicting parts of the model the conflict can be localised at first glance. The graphical representation of both, the base version and the resolution pattern, provides easy comparability of one's options for resolution. The textual discussion in the chat window can be extended with the optional use of the *Whiteboard*, where details can be explained graphically, ideas communicated, and sketches drawn.

Communication

The outcome of the resolution process is a model, which is either drawn or adopted. The models representing the conflicting versions, the proper shapes to draw the solution, and different other options are required on the screen to find a resolution. Therefore the additional communication should be kept as condensed as possible. We find chat to be the best solution in this case. The

chat window takes up comparatively little space, and is accessible during the entire process. Textual information like this can be stored and reviewed easily. Users are not distracted by audio comments and do not have to handle additional visual input like video.

The *Communication window* is as concise as possible. Only necessary elements are incorporated. There is a line to type in messages and a *Send* button, a field where the dialogue is displayed, and of course a list of users. Font size and type may not be changed, and there are no pictures of users taking up room needed for something else. Clarity is established by colours assigned to users, and the users messages are displayed in that colour. The user menu can be accessed through a simple toolbar button, offering possibilities to invite users or view profiles. The only non-essential feature is the option to change the users assigned colours.

Textual communication may, in certain situations, be insufficient. An additional drawing canvas gives the possibility to describe a notion in detail and communicate ideas. Contrary to the drawing canvas used for drawing the actual resolution, this drawing canvas—the *Whiteboard* (cf. Section 7.2)—has to have a freehand mode.

General Features

General features include options in the toolbar (see Figure 7.1). The toolbar provides a zoom-level and a full-screen view. Basic functions are accessed here, like copy and paste, or to open a new *Whiteboard*. There also has to be an undo/redo-function, allowing the user to go a step back or forth.

At any time in the process the model in progress can be validated. The user can check if the model is valid at a certain time. This anticipates surprises at the end of the process.

To end a session in the CCR, the *Main window* includes the button “Solve”. Here, the user can either delay the resolution process by rescheduling it, or solve the conflict by choosing a specific version, which has to be accepted by both conflicting parties. Thus the chosen version is committed and stored in the model repository (cf. Section 7.3).

7.2 The User Interface

The CCR’s user interface is divided into two parts. At the bottom is the *Communication window*, while the larger part of the screen gives information and room for the graphical resolution. This part consists of one to four pages, arranged through tabs. The following section describes the arrangement of the features within the graphical user interface of the CCR, displayed in Figure 7.1.

The Communication Window

Any chosen feature has to be clearly arranged and represented. Convertibility of size is important, for in different situations different sizes of communication windows are convenient. A chat tool with the sole purpose of communication may be arranged differently than the chat in our planned extension for a model versioning tool, where communication is merely a supplement. Still, it is important, and certain factors have to be considered to provide usability. With chat, there is the position and the size of the chat window, which must be chosen in a way so

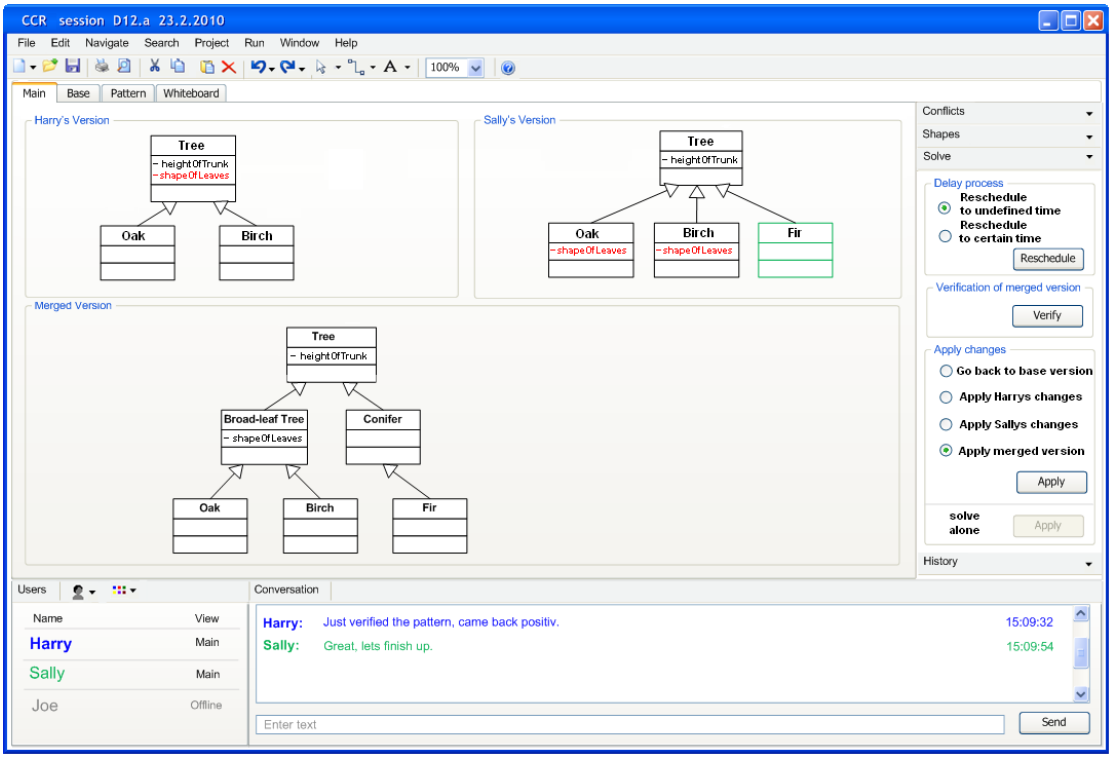


Figure 7.1: The Collaborative Conflict Resolver user interface

it does not constrain the concurrent execution of tasks. The lower left or right corner is usually favoured. An example for the choice of the left side is the game World of Warcraft (WOW, see Section 4.6), whereas the developers of Camel (see Section 4.4) place the chat window in the lower right corner. A sketch of the AMOR project team (see Figure 6.3) provides another option—placing the chat window on the left (or otherwise the right) side of the screen.

The importance of position and size of any feature integrated as part of the screen was discussed before, the point left is to determine position and size exactly. Where to put items, in this case the chat window, depends on different other criteria. The most important of these would be what else is on the screen.

The *Main window* provides the user with the contradictory versions, room for the merged version, i.e. the resolution, and options concerning the completion of the process. Optionally, additional room for discussion is given by the *Whiteboard*. If available, the process is supported by the base version the contradicting versions originated from and resolution patterns, offered by the *Resolution Pattern Storage*. During the process, these parts can be accessed consecutively, while the written conversation and the information of participating users forms the lower part of the CCR, visible and accessible the whole time (see Figure 7.1). The height of this part of the screen can be changed, to provide the user with a larger or smaller room for the typed conversation.

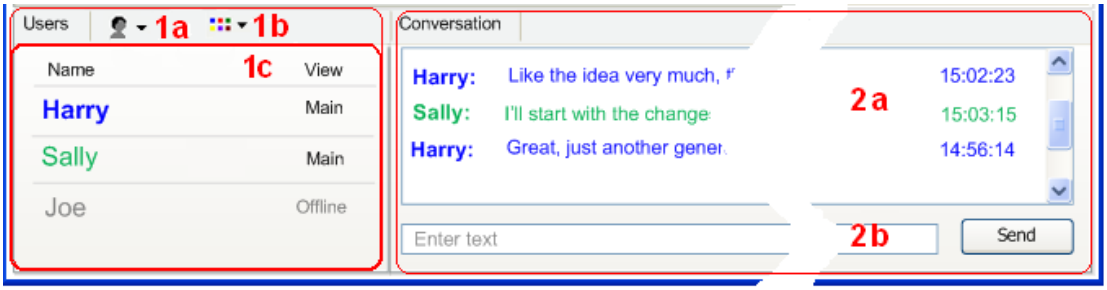


Figure 7.2: User list and conversation in the CCR

Figure 7.2 shows the *Communication window* of the CCR. Marked with number one is the section containing user information. At any time the user is provided with a list of the other participating users. Each user has a colour assigned, in which the user name is presented. The figure presents Harry's view, with his name on top of the user list (Figure 7.2 marked with 1 c)), appearing in bold. Also participating in the resolution process is Sally, while Joe was invited and is either not participating or has already left the process, hence his name being presented in grey. Next to the users names is the "View", indicating the window the users are looking at a specific time. In Figure 7.2, Harry for example knows that Sally is looking at the *Main window* as he does at that moment. The options within the *Communication window* are accessible through buttons 1 a) and 1 b). The first button extends to the user menu, offering to invite additional users and view user profiles. The profile includes a status, indicating whether the user is available at that moment or not. The second button allows the user to change his/her assigned colour. The typed conversation itself takes place in the right part of the *Communication window* (marked with number 2). This is a common chat window, displaying the messages (2 a)), a line to type in messages, and a button to send them (2 b)). The written messages appear in the assigned colour, the only colour not available is black, for it is reserved for messages from the system. Some of these messages appear in the chat window and others can only be reviewed in the history (see Figure 7.3c)). For example, if a user accepts a merged model which is incomplete and therefore can not be committed, the CCR sends a message about it including the reason why it can not be committed within the chat. The history includes actions made during the drawing of the merged model, who drew what and when. This information placed within the *Communication window* would soon fill it up. Non-critical messages will therefore appear only in the history, making each change traceable without disturbing the conversation.

The Main Window

The *Main window* is arranged in a similar way as the *Base window*, *Pattern window*, and *Whiteboard window*. The larger part of the user interface is taken over by a drawing canvas and/or graphical information. The right side provides the user with options. The *Main window* is obligatory in place when the process is started. The mentioned main part of the window presents a graphical visualisation of the conflict (see Figure 7.1) and a at first blank space for the res-

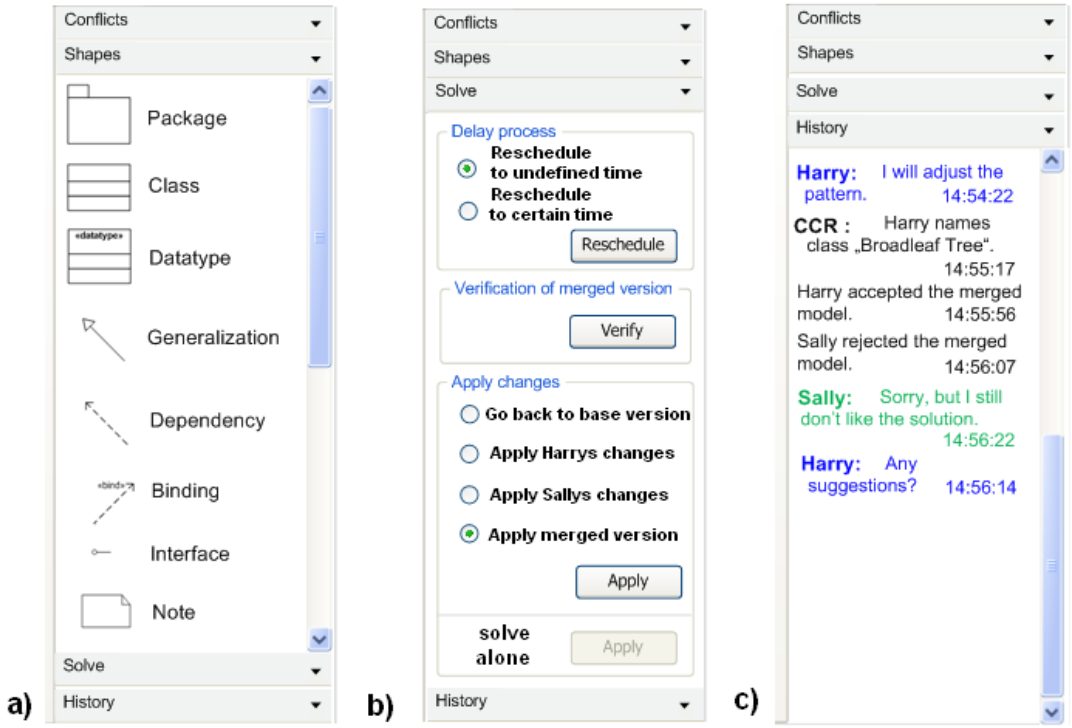


Figure 7.3: Options in the Main window

olution. The options are divided into four parts, named *Conflicts*, *Shapes*, *Solve*, and *History*. These parts can be opened consecutively. The *Conflicts window* is opened by default. If another window is opened, the former one is closed simultaneously, much like the shapes menu provided by Microsoft's Visio[1], a drawing tool providing shapes for any occasion. The four parts contain the options as shown in Figure 7.3 and described in the following.

Conflicts. This section displays the textual description of the conflict—or the conflicts, if several exist—and is opened by default at the beginning of the process. For an example, which will be explained in detail in Section 7.4 (see Figure 7.1).

Shapes. The shapes required to draw are accessed here. Depending on the kind of model, the adequate shapes are offered (see Figure 7.3a)).

Solve. To end the resolution process, there are two options. Either delay the process by rescheduling to an indefinite or a fixed date and time, or commit a solution (see Figure 7.3b)).

If, for example, one person has to leave the collaborative resolution process to attend a meeting, or is not available in the first place, the process may be rescheduled. A conjointly

[1] http://office.microsoft.com/en-us/visio

Figure 7.4: Commit the solution

rescheduled session to a certain date may be restarted at that time by notification like the first time. Without a specified date and time, the CCR reminds the conflicting parties of the still existing conflict from time to time. A shared calendar is needed in any case, whether it is built-in or external. An external calendar could be one of the calendars integrated in widely-used e-mail clients.

Options to commit a model are to either go back to the base version, accept one of the conflicting versions, or use the merged version, created during the process. The version of choice is selected by one of the users and committed by the *Accept* button. During the conflict resolution, the model in progress may be verified to check if it is valid. If other conflicts are triggered by the new version, the users are informed, but can still accept the version. The acceptance of a version by one user leads to the other user being asked if he or she agrees. If, for example, Harry accepts the merged model, Sally is provided with two options in a pop-up window (see Figure 7.4). She can either accept the merged model too, which leads to a commit—the selected version is saved within the model repository—and ends the session, or reject it. In case of rejection, the resolution process continues.

In the case that only one of the conflicting parties is available, this person can solely reschedule the process. He or she can also commit a solution. If only one person is participating, and no one else is invited into the process, the first *Apply* button is disabled, and the second one enabled (cf. Figure 7.1), allowing the user to commit one of the versions by him/herself. This is recommended only if the other conflicting party is not available and the conflict has to be solved immediately.

Regarding the invitation of additional users to the conflict resolution process, their acceptance of a version to be committed to the repository is not needed, except if this person substitutes one of the other conflicting parties. Taking the example given in Figure 7.1, if a third person invited to aid the resolution process would still be in the session, his/her acceptance would neither be required nor needed. If the second conflicting party is unavailable and the first person therefore asks someone else (third person) to substitute this second person in the process, the resolution has to be accepted by the first and third person to be committed.

History. The *History* provides a log of the conversations and the actions performed. In Figure 7.3c), some entries are formed by the conversation of Harry and Sally, the rest documents Harry making changes and accepting the merged model, which is then rejected by Sally.

The Base Window

The *Base window* contains the graphical visualisation of the version, the conflicting parts originated from, i.e. the base version. Also, relations to other fragments are displayed, which have to be kept in mind, for a change in the conflicting fragment could affect other parts and hence create a new conflict.

The Pattern Window

If one or more suggestions in form of patterns exist for the conflict, the CCR is extended by the *Pattern window*. Formerly accumulated similar situations are displayed in form of resolution patterns in the *Pattern window*. A displayed pattern can be chosen, and with the *Use pattern* button directly transferred to the *Main window*, where the pattern can be adjusted to the current situation.

The Whiteboard

A whiteboard is opened by default in the CCR, but of course it does not have to be used. Within the *Whiteboard window*, the user is provided with a list of shapes, not only for the type of model in conflict. Sets of shapes for different other types of models are offered as well as general shapes, like a simple rectangle or circle, and a free-hand tool. This whiteboard offers additional room and drawing possibilities to support the discussion. A drawing consisting of compatible elements may be adopted into the *Main window*. This way, a solution found within the *Whiteboard* does not have to be drawn in the *Main window* a second time.

7.3 The Conflict Resolution Process

In case of a conflict, the Collaborative Conflict Resolver notifies and invites the users responsible for the conflict. There is the possibility for one person to solve the conflict alone, but the collaborative approach is more beneficial. We assume, the conflicting parties joined in a session to seek a conjoint outcome. By default, the users are first presented with the graphical and textual representation of the conflict (view conflict). The process can either be rescheduled, which ends the session, or a solution can be found and committed. During the whole process, the users may communicate via chat or the *Whiteboard*. If a pattern is available, it can be applied and customised. At any time additional information, like the base version and the models relationships to surrounding models can be looked up, while a new version may be created. These activities—creating a solution and looking up information—can repeatedly be done consecutively in any order. A solution accepted by one person can be accepted by the other, therewith the model is committed to the repository, or rejected, which leads to a continuation of the session.

7.4 Example of Use

Instead of forcing the users to participate in a cycle of mandatory actions forming the resolution process, the CCR GUI is arranged to make this process self-explanatory. In the event of a con-

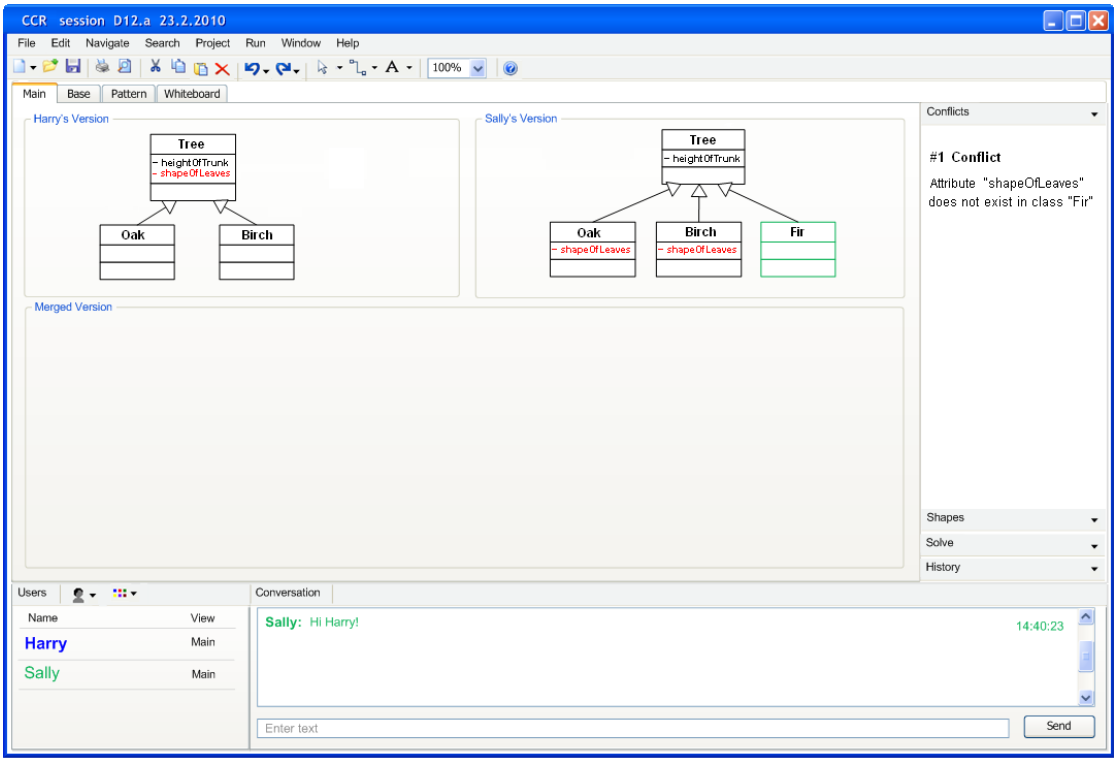


Figure 7.5: A collaborative conflict resolution process (1)

flict, the conflicting parties are notified about this conflict and—we assume that these parties are available at that moment—invited to partake in the resolution process. The relevant information and room for discussion and creation of a solution or sketches can be accessed at any time and does not have to be in a specific order. In this example of use, the conflicting parties are called Harry and Sally. The screenshots display Harrys point of view.

The first step of each participating party is to look at the given information. The problem is presented graphically and textually in the *Main window*. Figure 7.5 displays the beginning of the conflict resolution process. By default the *Main window* is opened, and Harry is presented with the textual description of the conflict, and with his and Sallys version—the graphical representation of the problem. The elements which generated the conflict are highlighted in terms of colour, allowing Harry to comprehend the conflict situation at once.

The conflict occurred through the following conflicting modifications in the class *Tree* and its subclasses. Harry took the attribute *shape of leaves* from the subclasses *Oak* and *Birch* to the class *Tree*, for the attribute is contained in all subclasses. Sally, unaware of Harrys changes, introduced a new subclass *Fir*, which does not contain the mentioned attribute.

The *Communication window*, consisting of the chat window and the user list, is visible during the whole process, while the other elements can be switched. The user list on the left side of the *Communication window* (see Figure 7.5) shows Harry—his name appears bold—that next

to him, Sally is participating in the process. The "view" next to the user names, allows Harry to know that Sally is now looking at the same window as he does. In the chat window Sally's first message appears with a time stamp, greeting Harry.

Harry and Sally can individually look up additional information. This includes in the *Base window* the base version from which the conflicting fragments were generated, and relations, for example to surrounding classes, which might be *park* and *garden*.

A similar conflict was already solved earlier, creating a resolution pattern, suggested in the *Pattern window*. Harry opens the pattern to view it in more detail, where he is presented not only with the suggested pattern, but also with the conflicting versions and the base version (see Figure 7.6).

After discussing the proposed resolution pattern with Sally in the chat window, they agree that the pattern is useful and to continue with the pattern. Still within the *Pattern window*, Harry clicks the button *Apply pattern*, and the pattern is transferred to the resolution space in the *Main window*, where Harry and Sally now adjust the pattern. In Figure 7.7, Harry already has filled in the missing class name. Within the window *Main*, he opens the *History* option, allowing him not only to review the conversation, but also events. In this case, Harry is satisfied with the solution—namely the modified resolution pattern—and accepts the merged model to be reintegrated.

Therefore Harry opens the *Solve* option within the *Main window* (see Figure 7.10), chooses the version he wants to solve the conflict with, and clicks the *Accept* button. Now Sally is presented with a pop-up window, telling her that Harry accepted a certain version, and asking if she wants to do so too (cf. Figure 7.4). Sally chooses not to accept but to reject the solution, for she is not yet satisfied with it. And as soon as she rejects the solution with the *Reject button*, she is back in the CCR *Main window*, telling Harry so in the chat window. Now Harry asks for other suggestions from Sally. Harry is now reviewing the dialog and the changes made in the History option (see Figure 7.7). Since Sally is not satisfied with the solution yet, but does not know how to improve it, Harry and Sally decide to ask Joe for suggestions.

A button in the user list—part of the *Communication window*—allows Sally to see that Joe has set his status in his profile to *available*, and she invites him to join the resolution process. Joe accepts the invitation, and luckily has an idea to help out Harry and Sally. To explain his idea, Joe uses the offered shapes and freehand drawing in the *Whiteboard* (see Figure 7.8).

In the *Communication window* of Figure 7.8, Joe now appears within the user list, while the chat window displays his conversation with Sally, and Harry is just about to send his remark about the ongoing events.

In Figure 7.9 Joe has already left the session, therefore his name now appears in grey instead of purple like before. Since Harry and Sally both liked Joes suggestion, they go back to the *Main window* to adapt it. Harry and Sally have already created the class *Conifer* with the offered shapes. Only another relation is mission, as Harry states in the chat in Figure 7.9.

In the *Main window*, the *Solve* option offers three choices (see Figure 7.10). First, the process can be rescheduled, and when restarted, continues where the participating users left off. Second, at any time during the process the new common version the conflicting parties are working on may be verified. Through changes, for example deleting a class which has relations outside the displayed fragment, new conflicts may occur. The reintegration of the fragment worked on is

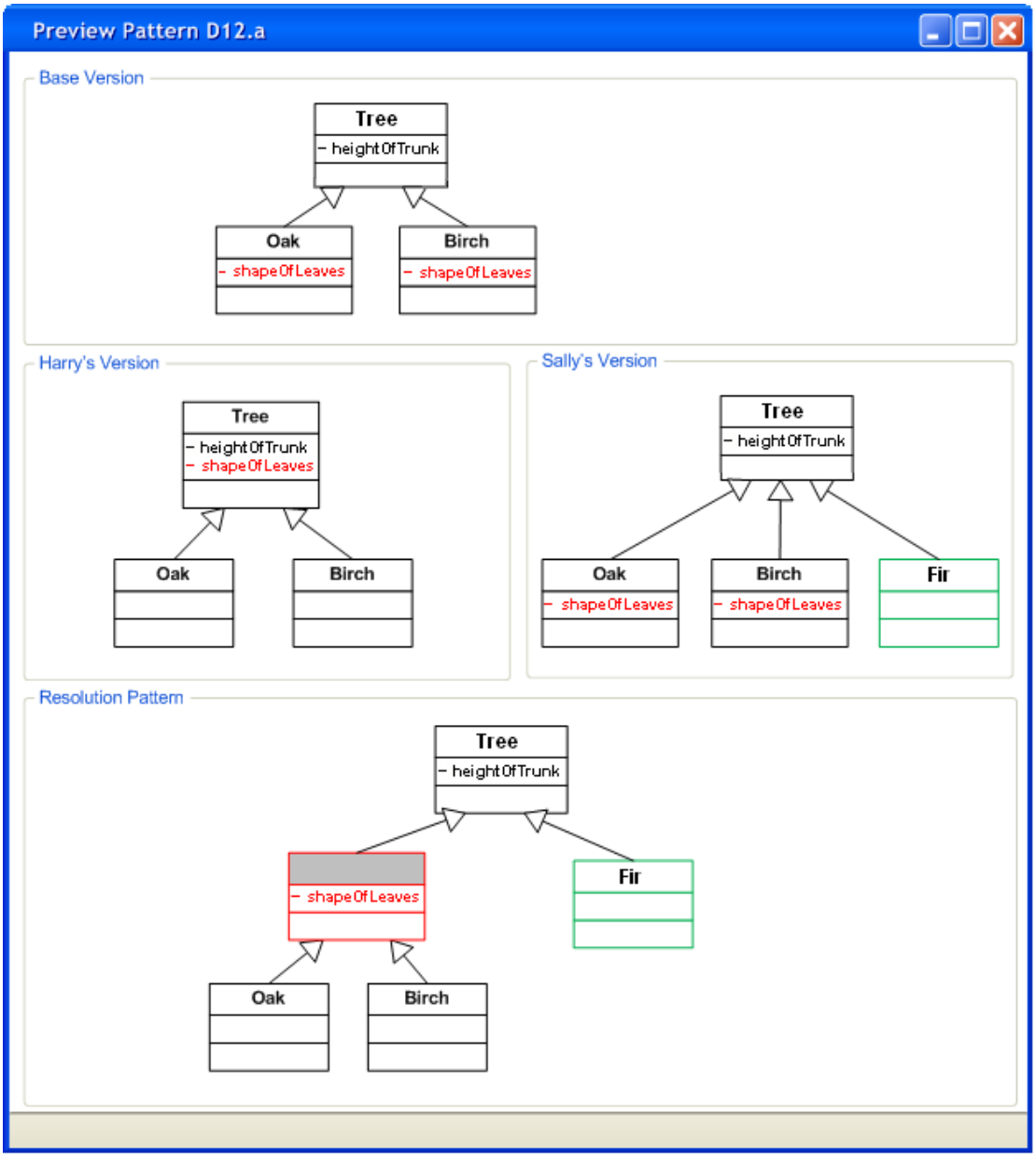


Figure 7.6: A collaborative conflict resolution process (2)

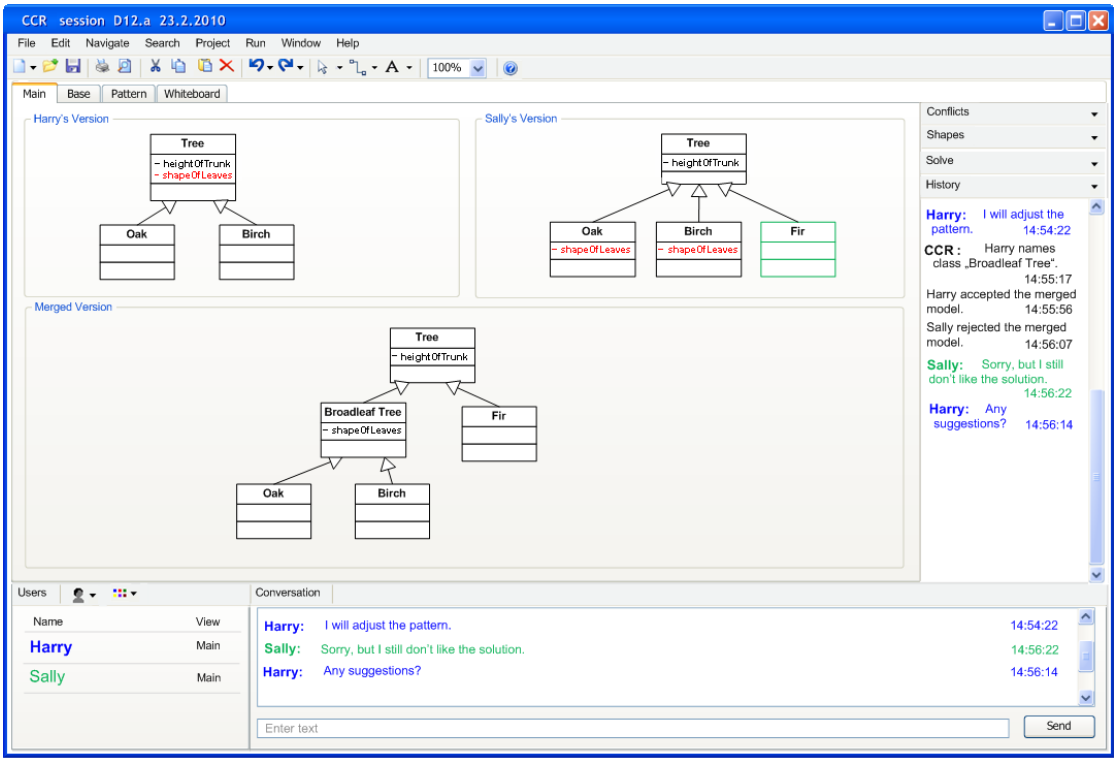


Figure 7.7: A collaborative conflict resolution process (3)

not prevented by a potential conflict created by solving another conflict. The users can verify the correctness of their work during the process for assurance and to avoid unpleasant surprises in the end. In Figure 7.10 Harry verified the found solution and told Sally so in the chat.

Third, the session can be ended in the most desired way, by applying a conjointly found solution. In our example, Harry and Sally finished adjusting the pattern and integrating Joes ideas. Sally chooses the option *Apply merged version* and clicks the *Apply* button.

This time, Harry is presented with the options *Accept* and *Reject* after being asked if he wants to apply the chosen version too (cf. Figure 7.4). Naturally, Harry accepts. This ends the conflict resolution process by integrating the new conjointly found solution for the conflict into the model repository.

7.5 The Modified CCR for Discussion

Without a conflict, the CCR can be used for discussion. Imagine a user Dave, who reviewed the resolution process of our example above. Before making contradictory changes, he may want to discuss the resolution found and its motivation. Within his contacts, he sees that Sally's status is set to *available*, so he sends her an invitation to partake in a discussion concerning the found solution. Sally is not only invited, but also informed about the addressor and the topic (see

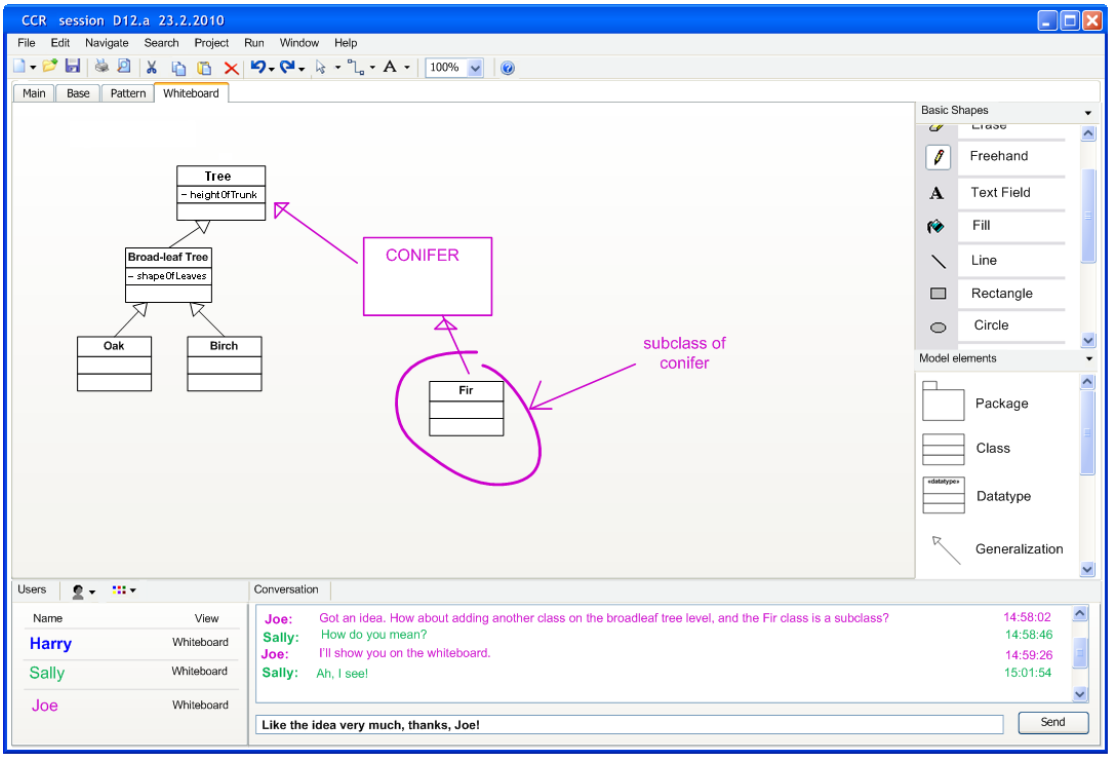


Figure 7.8: A collaborative conflict resolution process (4)

Figure 7.11). The invitation contains a link to the addressor's profile, and one to the topic he or she wants to discuss. Declining the invitation provides Sally at once with a possibility to send a message to Dave, explaining her reasons. This is a matter of common courtesy, and hence the opportunity should be given.

Accepting the invitation on the other hand, opens a modified version of the CCR. This version includes, since there is no conflict, the *Whiteboard* and the *Communication window* only. If a certain topic is specified, in this example the formerly conjointly found version of the conflict between Harry and Sally, the information concerning that session is displayed too. A session in the modified CCR ends, when the last user leaves the session. Only specified information is saved. The modified CCR supports the users with the advantages of the CCR without having to wait for a conflict. Collaborative features—like a chat, a user list, a shared drawing canvas—are advantageous not only to deal with finicial situations like a conflict, but also in general.

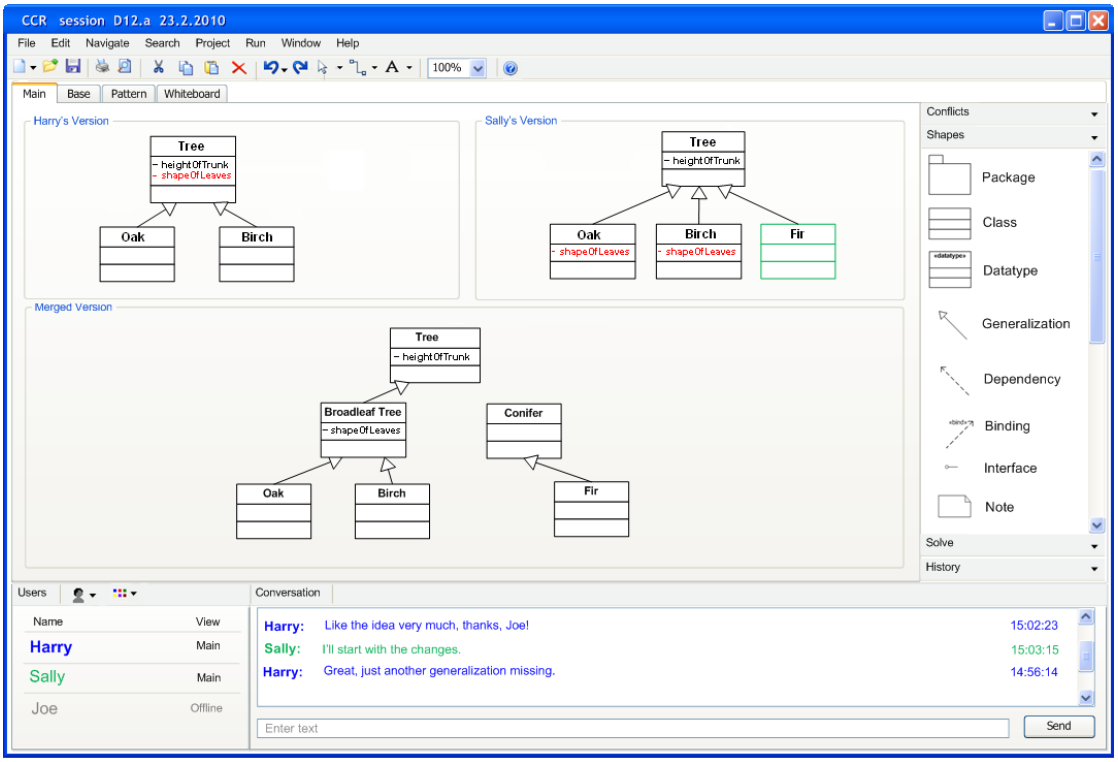


Figure 7.9: A collaborative conflict resolution process (5)

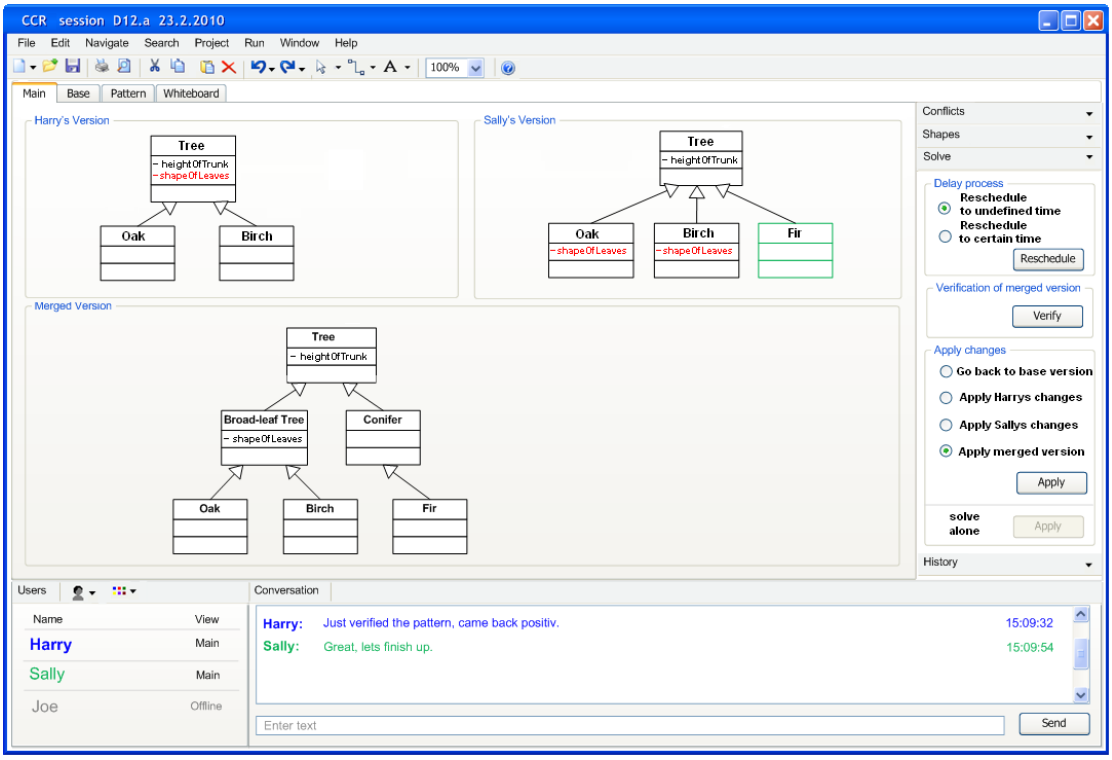


Figure 7.10: A collaborative conflict resolution process (6)

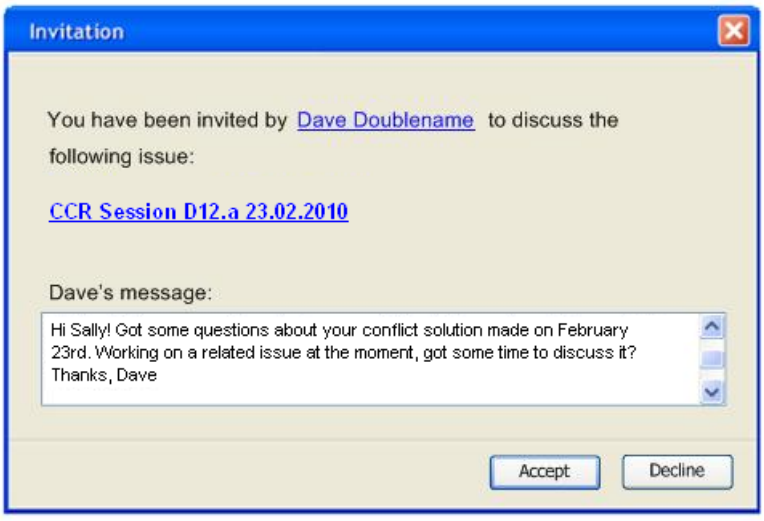


Figure 7.11: Invitation to use the CCR for discussion

CHAPTER 8

Conclusion

With very few exceptions not individuals but teams develop software, still the typical tools for development, such as editors and compilers, are effectively designed for single users. During their work, developers spend much of their time with collaborative activities, including meetings and other communication, coordination of tasks, and collaboratively executing these tasks. Shaping collaboration effectively is vital, for collaborative aspects impact the process at many levels.

Version control is a necessity in software engineering (SE). Optimistic versioning allows parallel development in teams. Due to concurrent changes in one artefact, conflicts may occur. The number of conflicts may be reduced by well working collaboration, or for example by optional lock mechanism in an otherwise optimistic versioning tool. Resolution of conflicts is an especially critical task, but solving these conflicts collaboratively promises a better outcome.

Software developers are often reluctant to satisfactory and descriptively comment produced code, but too general comments or non at all are hindering the maintenance later. In parallel development, these descriptive comments are even more important. Additional comments may be provided through the versioning system, when a user describes the changes made on his/her version of an artefact before committing. Easily removable comments for the sole purpose of informing the next person working on an artefact, some kind of sticky notes, may also be beneficial for collaboration. These comments create awareness of who did what at which time. Collaboration is improved by co-workers being aware of each others responsibilities and authorisations in general, projected and carried out tasks, and their current status, that is for example whether they are available for communication or the jointly execution of a task at a certain time.

Collaboration, working together to achieve common goals, does not only include the collaborative execution of tasks, but communication, and coordination of tasks and communication. Since available software development tools usually do not support communication, users resort to different external tools. It may be that a development team uses e-mail, an instant messaging tool, the phone, a common calendar, and a way of screen sharing all together, while additionally attending regular meetings. The tools intended to support users coordinating their work may create an information and communication overload, and thereby a disturbance of work. There

are two approaches to lessen the negative impact of unregulated communication. First, rules and policies on which tools have to be used in which manner reduces the number of used tools and regulates that use. Second, as it is common practice in various other fields foreign to SE, a built-in communication feature may be highly beneficial.

The possibilities are numerous. Features like video conferencing, screen sharing, the common use of a whiteboard, an instant messaging system, or chat may be integrated into an existing system. The way of communication may be chosen according to the number and location of participants, the situations and the time in the software development life cycle the feature is intended to be used, and other criteria. Distributed SE opens the possibility to work around the clock. Regarding synchronous and asynchronous communication, in global software engineering the latter may be the only available choice through different time zones.

For example, when deciding on a chat feature as means of communication, the size and position of the chat window has to be determined, how this feature is accessed, and whether to add a timestamp to the messages. It may also be beneficial, to distinguish between private and public messages. There has to be a user list for potential communication partners, and for reasons of clarity the chat may be coloured in different ways. The chat may contain user messages only (cf. Section 4.4), additional informations about collaborative activities (cf. Section 4.4), or important system messages (cf. Section 7.2).

In any case, the users have to be provided with means for synchronous and asynchronous communication, as well as spontaneous and prearranged. What features are incorporated by an integrated communication tool has to be decided by keeping in mind, that any not supported but necessary way of communication will eventually be substituted by other, external means of communication.

Well-working collaboration is certainly not restricted to proper tool support and rules on how to use those. The usability of a collaborative tool as well as collaboration in general is influenced by different social aspects, which are likely to be disregarded, although having a great impact. Collaboration is formed through human characteristics, individual behavior, and cooperation. It is influenced by geographical and cultural aspects, as well as organisational structure and culture. Next to the individuals' position, the degree of acquaintance affects the collaboration. Team members not knowing each other may be reluctant to communicate at all. To properly support and facilitate collaboration, these aspects have to be minded.

Next to this specific proposal for collaborative conflict resolution, this thesis provides a guideline on how to efficiently support collaboration in model versioning. We present an overview of social aspects influencing the collaborative process, and explain the importance of taking these aspects into account. We list possibilities for collaborative support and describe the usability—advantages and disadvantages—of these possibilities, thereby providing our readers with a manual on how to efficiently improve collaboration in conflict resolution in model versioning in particular and in software engineering projects in general.

If there is a software system—instead of many—used to accomplish the collective work, integrated collaborative features facilitate the direct and the indirect work. Collaborative features contribute to finding a solution in conflict situations and to avoiding conflicts. In this thesis we present a detailed suggestion on how to improve conflict resolution in model versioning by facilitating the collaborative process. That is in particular the integration of the communication

feature chat, extended by user list and profiles containing the user's status to further awareness. To help the resolution process, we propose to add freehand drawing of sketches and notes in an additional shared drawing canvas. Simultaneous use of relevant information is provided with the graphical representation of not only the conflicting versions, but the base version of those, relationships to the surrounding area of the fragment in question, and suggestions for solution. The conflict may be solved by one person solely, but is recommended to be solved collaboratively. Thereby all motivations are taken into account, which leads to the best possible solution. This might be to go back to the base version, chose one of the conflicting versions, or create a new merged version. This approach may also be used for common discussion and collaborative development without being presented with a conflict before.

Mnemonics

The following abbreviations and acronyms appear in this thesis.

short	**means**
AMOR	Adaptable Model Versioning
CASE	Computer Aided/Assisted Software Engineering
CCR	Collaborative Conflict Resolver, planned in project AMOR
CSCW	Computer supported collaborative work
CSE	Collaborative software engineering
CMC	Computer mediated communication
CVS	Concurrent Versions System
DE	Development environment
DSML	Domain-specific modelling languages
GUI	Graphical User Interface
IDE	Integrated Development Environment
IM	Instant messaging
MDE	Model-driven engineering
MDE	Multiple display environment (in Section 4.3)
MDD	Model-driven development
MDSD	Model-driven software development
MMORPG	Massively multiplayer online role-playing game
SCM	Software configuration management
SD	Software development
SE	Software engineering
UI	User Interface
VCS	Version Control System
WoW	World of Warcraft (a MMORPG)

Table 1: Abbrevations and acronyms

Bibliography

[1] Adelson, B. A Collaborative Negotiation Tool. *ACM SIGCHI Bulletin 23*, 4 (1991), 77.

[2] Afonso, M., Vogel, R., and Teixeira, J. From Code Centric to Model Centric Software Engineering: Practical Case Study of MDD Infusion in a Systems Integration Company. In *Proc. of the 4th Workshop on Model-Based Development of Computer-Based Systems and 3rd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software* (2006), IEEE Computer Society, pp. 125–134.

[3] Altmanninger, K., Bergmayr, A., Schwinger, W., and Kotsis, G. Semantically Enhanced Conflict Detection between Model Versions in SMoVer by Example, 2007.

[4] Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., and Wimmer, M. AMOR—Towards Adaptable Model Versioning. In *Proc. of the 1st International Workshop on Model Co-Evolution and Consistency Management* (2008), IEEE Computer Society.

[5] Altmanninger, K., Seidl, M., and Wimmer, M. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems 5*, 3 (2009), 271–204.

[6] Ambriola, V., Bendix, L., and Ciancarini, P. The Evolution of Configuration Management and Version Control. *Software Engineering Journal 5*, 6 (1990), 303–310.

[7] AMOR Project. http://www.modelversioning.org. Last visit: 2009-12-14.

[8] ArgoUML. http://argouml.tigris.org. Last visit: 2010-04-04.

[9] Barone, I., De Lucia, A., Fasano, F., Rullo, E., Scanniello, G., and Tortora, G. COMOVER: Concurrent Model Versioning. In *Proc. of the IEEE International Conference on Software Maintenance* (2008), IEEE Computer Society, pp. 462–463.

[10] Barrett, S., Chalin, P., and Butler, G. Model Merging Falls Short of Software Engineering Needs. In *Proc. of the 2nd Workshop on Model-Driven Software Evolution* (2008).

[11] Berliner, B. CVS II: Parallelizing Software Development. In *Proc. of the USENIX Winter Technical Conference* (1990), USENIX Association, pp. 341–352.

[12] Bézivin, J. On the Unification Power of Models. *Software and System Modeling 4*, 2 (2005), 171–188.

[13] Biehl, J. T., Baker, W. T., Bailey, B. P., Tan, D. S., Inkpen, K. M., and Czerwinski, M. IMPROMPTU: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and its Field Evaluation for Co-Located Software Development. In *Proc. of the 26th Annual SIGCHI Conference on Human Factors in Computing Systems* (2008), ACM, pp. 939–948.

[14] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[15] Brignall, T. W., and Van Valey, T. L. An Online Community as a New Tribalism: the World of Warcraft. In *Proc. of the 40th Hawaii International Conference on System Sciences* (2007), IEEE Computer Society, p. 179b.

[16] Brosch, P. Improving Conflict Resolution in Model Versioning Systems. In *Proc. of the 31st International Conference on Software Engineering, Companion Volume* (2009), IEEE Computer Society, pp. 355–358.

[17] Brosch, P., Kappel, G., Seidl, M., Wieland, K., Wimmer, M., Kargl, H., and Langer, P. Adaptable Model Versioning in Action. In *Modellierung* (2010), LNI 161, pp. 221–236.

[18] Brosch, P., Langer, P., Seidl, M., and Wimmer, M. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Proc. of the ICSE Workshop on Comparison and Versioning of Software Models* (2009), IEEE Computer Society, pp. 55–60.

[19] Brosch, P., Seidl, M., Wieland, K., Wimmer, M., and Langer, P. We can work it out: Collaborative Conflict Resolution in Model Versioning. In *Proc. of the 11th European Conference on Computer Supported Cooperative Work* (2009), Springer-Verlag, pp. 207–214.

[20] Brosch, P., Seidl, M., and Wimmer, M. Mining of Model Repositories for Decision Support in Model Versioning. In *Proc. of the 2nd European Workshop on Model-Driven Tool and Process Integration* (2009), CTIT Workshop Proceedings, pp. 25–33.

[21] Bruegge, B., and Dutoit, A. A. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, 1999.

[22] Candrlic, S., Pavlic, M., and Poscic, P. A Comparison and the Desireable Features of Version Control Tools. In *Proc. of the 29th International Conference on Information Technology Interfaces* (2007), IEEE Computer Society, pp. 121–126.

[23] Carver, J., Jaccheri, L., Morasca, S., and Shull, F. Issues in Using Students in Empirical Studies in Software Engineering Education. In *Proc. of the 9th International Symposium on Software Metrics* (2003), IEEE Computer Society, p. 239.

[24] Cataldo, M., Shelton, C., Choi, Y., Huang, Y.-Y., Ramesh, V., and Saini, Darpan andWang, L.-Y. CAMEL: A Tool for Collaborative Distributed Software Design. In *Proc. of the 4th IEEE International Conference on Global Software Engineering* (2009), IEEE Computer Society, pp. 83–92.

[25] Cheng, L.-T., De Souza, C., Hupfer, S., Patterson, J., and Ross, S. Building Collaboration into IDEs. *ACM QUEUE 1*, 9 (2003-2004), 40–50.

[26] Cheng, L.-T., Hupfer, S., Ross, S., and Patterson, J. Jazzing up Eclipse with Collaborative Tools. In *Proc. of the OOPSLA Workshop on Eclipse Technology Exchange* (2003), ACM, pp. 45–49.

[27] Cheng, L.-T., Hupfer, S., Ross, S., Patterson, J., Clark, B., and De Souza, C. Jazz: A Collaborative Application Development Environment. In *Proc. of the 18th Annual Conference on Object Oriented Programming, Languages, and Applications* (2003), ACM, pp. 102–103.

[28] Cheng, L.-T., Patterson, J., Rohall, S. L., Hupfer, S., and Ross, S. Weaving a Social Fabric into Existing Software. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development* (2005), ACM, pp. 147–158.

[29] Chu-Carroll, J., and Carberry, S. Communication for Conflict Resolution in Multi-Agent Collaborative Planning. In *Proc. of the 1st International Conference on Multiagent Systems* (1995), MIT Press, pp. 49–56.

[30] Chu-Carroll, J., and Carberry, S. Response Generation in Collaborative Negotiation. In *Proc. of the 33rd Annual Meeting on Association for Computational Linguistics* (1995), Association for Computational Linguistics, pp. 136–143.

[31] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. *Version Control with Subversion*. O'Reilly Media, 2008.

[32] Conradi, R., and Westfechtel, B. Version Models for Software Configuration Management. *ACM Computing Surveys 30*, 2 (1998), 232–282.

[33] Cook, C., Churcher, N., and Irwin, W. Towards Synchronous Collaborative Software Engineering. In *Proc. of the 11th Asia-Pacific Software Engineering Conference* (2004), IEEE Computer Society, pp. 230–239.

[34] Damm, C. H., Hansen, K. M., Thomsen, M., , and Tyrsted, M. Supporting Several Levels of Restriction in the UML. In *Proc. of UML2000* (2000), Springer-Verlag, pp. 396–409.

[35] De Lucia, A., Fasano, F., Scanniello, G., and Tortora, G. Concurrent Fine-Grained Versioning of UML Models. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering* (2009), IEEE Computer Society, pp. 89–98.

[36] De Souza, C., Sharp, H., Singer, J., Cheng, L.-T., and Venolia, G. Cooperative and Human Aspects of Software Engineering. In *Proc. of Cooperative and Human Aspects of Software Engineering, Guest Editors' Introduction* (2009), IEEE Computer Society, pp. 17–19.

[37] Dearle, A. Software Deployment, Past, Present and Future. In *Proc. of the International Conference on Software Engineering, Future of Software Engineering* (2007), IEEE Computer Society, pp. 269–284.

[38] Delinchant, B., Gerbaud, L., Wurtz, F., and Atienza, E. Concurrent Design Versioning System, Based on XML File. In *Proc. of the 28th Annual Conference of the Industrial Electronics Society* (2002), IEEE Computer Society, pp. 2485–2490.

[39] DeMarco, T., and Lister, T. Peopleware, 1987. Dorset House, New York.

[40] Dewan, P., and Hegde, R. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *Proc. of the 10th European Conference on Computer-Supported Cooperative Work* (2007), Springer-Verlag, pp. 159–178.

[41] Edwards, W. K. Flexible Conflict Detection and Management in Collaborative Applications. In *Proc. of the 10th Annual ACM Symposium on User Interface Software and Technology* (1997), ACM, pp. 139–148.

[42] Fabro, M. D. D., Bézivin, J., and Valduriez, P. Weaving Models with the Eclipse AMW Plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe* (2006).

[43] Favela, J., and Pena-Mora, F. An Experience in Collaborative Software Engineering Education. *IEEE SOFTWARE 18*, 2 (2001), 47–53.

[44] FileInfo.com. http://www.fileinfo.com/extensions/ics. Last visit: 2009-12-09.

[45] Fish, R. S., Kraut, R. E., Leland, M. D. P., and Cohen, M. Quilt: a Collaborative Tool for Cooperative Writing. In *Proc. of the ACM SIGOIS and IEEECS TC-OA 1988 Conference on Office Information Systems* (1988), ACM, pp. 30–37.

[46] Fowler, M. Is There Such a Thing as Object-Oriented Analysis? *Distributed Computing Magazine* (1999), 40–41.

[47] Fowler, M. *UML konzentriert*, 3rd ed. Addison-Wesley, 2004.

[48] France, R., and Rumpe, B. Model-Driven Development of Complex Software: A Research Roadmap. In *Proc. of the 29th International Conference on Software Engineering, Future of Software Engineering* (2007), IEEE Computer Society, pp. 37–54.

[49] France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer 39*, 2 (2006), 59–66.

[50] Frost, R. Jazz and the Eclipse Way of Collaboration. 114–117.

[51] Fussell, S. R. Culture and Collaborative Technologies. In *Proc. of the Conference on Human Factors in Computing Systems* (2007), ACM, pp. 2845–2848.

[52] Glassy, L. Using Version Control to Observe Student Software Development Processes. *Journal of Computing Sciences in Colleges 21*, 3 (2006), 99–106.

[53] Grinter, R. E. Using a Configuration Management Tool to Coordinate Software Development. In *Proc. of the Conference on Organizational Computing Systems* (1995), ACM, pp. 168–177.

[54] Grochow, K. COVE: A Visual Environment for Multidisciplinary Science Collaboration. In *Proc. of the International Conference on Supporting Group Work* (2009), ACM, pp. 377–378.

[55] Hansen, K. M., and Ratzer, A. V. Tool Support for Collaborative Teaching and Learning of Object Oriented Modeling. In *Proc. of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (2002), ACM, pp. 146–150.

[56] Hartness, K. T. N. Eclipse and CVS for Group Projects. *Consortium for Computing Sciences in Colleges 21*, 4 (2006), 217–222.

[57] Herrmannsdoerfer, M. Operation-Based Versioning of Metamodels with COPE. In *Proc. of the ICSE Workshop on Comparison and Versioning of Software Models* (2009), IEEE Computer Society, pp. 49–54.

[58] Herzig, K., and Zeller, A. Mining the Jazz Repository: Challenges and Opportunities. In *Proc. of the 6th IEEE International Working Conference on Mining Software Repositories* (2009), IEEE Computer Society, pp. 159–162.

[59] Hovsepyan, A., Van Baelen, S., Vanhooff, B., Joosen, W., and Berbers, Y. Key Research Challenges for Successfully Applying MDD within Real-Time Embedded Software Development. In *Proc. of the International Workshop on Embedded Computer Systems: Architectures, Modeling and Simulation* (2006), Springer-Verlag, pp. 49–58.

[60] ICQ. http://www.icq.com. Last visit: 2009-12-09.

[61] Ideogramic ApS. http://www.ideogramic.com. Last visit: 2010-02-01.

[62] IEEE. lEEE Std 610.121990, Standard Glossary of Software Engineering Terminology, 1990. The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY 10017, USA.

[63] IEEE Computer Society. IEEE Standard for Software Configuration Management Plans.

[64] Ignat, C.-L., and Norrie, M. C. Draw-Together: Graphical Editor for Collaborative Drawing. In *Proc. of the 20th Anniversary Conference on Computer Supported Cooperative Work* (2006), ACM, pp. 269–278.

[65] Jazz. http://jazz.net. Last visit: 2010-01-11.

[66] Jones, T. C. Programming Productivity, 1986. MCGraw-Hill, New York.

[67] Jorda, S. Faust Music On Line (FMOL): An Approach to Real-Time Collective Composition on the Internet. *Leonardo Music Journal 9* (1999), 5–12.

[68] Jorda, S., and Wust, O. A System for Collaborative Music Composition over the Web. In *Proc. of the 12th International Workshop on Database and Expert Systems Applications* (2002), IEEE Computer Society, pp. 537–542.

[69] Junqueira, D. C., Bittar, T. J., and Fortes, R. P. M. A Fine-Grained and Flexible Version Control for Software Artifacts. In *Proc. of the 26th Annual International Conference on Design of Communication* (2008), ACM, pp. 185–192.

[70] Kögel, M. Towards Software Configuration Management for Unified Models. In *Proc. of the International Workshop on Comparison and Versioning of Software Models* (2008), ACM, pp. 19–24.

[71] Könemann, P. Model-Independent Differences. In *Proc. of the ICSE Workshop on Comparison and Versioning of Software Models* (2009), IEEE Computer Society, pp. 37–42.

[72] Ladas, C. The Difference between SE and SD; http://leansoftwareengineering.com/2007/11/10/the-difference-between-software-development-and-software-engineering.

[73] Langer, P. Konflikterkennung in der Modellversionierung. Master's thesis, Vienna University of Technology, 2009.

[74] Lin, Y., Zhang, J., and Gray, J. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *Proc. of the OOPSLA Workshop on Best Practices for Model-Driven Software Development* (2004).

[75] Louridas, P. Version Control. *IEEE Software 23*, 1 (2006), 104–107.

[76] McGinnes, S. CASE Support for Collaborative Modelling: Re-Engineering Conceptual Modelling Techniques to Exploit the Potential of CASE Tools. *Software Engineering Journal 9*, 4 (1994), 183–189.

[77] Mens, T. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering 28*, 5 (2002), 449–462.

[78] MetaCase. *MetaEdit+ User's Guide*, 2005.

[79] Microsoft Office Online. http://office.microsoft.com. Last visit: 2010-02-04.

[80] Murta, L., Corrêa, C., Prudêncio, J. G., and Werner, C. Towards Odyssey-VCS 2: Improvements over a UML-Based Version Control System. In *Proc. of the International Workshop on Comparison and Versioning of Software Models* (2008), ACM, pp. 25–30.

[81] Nardi, B. Collaborative Play in World of Warcraft. In *Proc. of the 4th Latin American Web Congress* (2006), IEEE Computer Society, p. 3.

[82] Object Management Group, Inc. Unified Modeling Language (UML); http://www.uml.org. Last visit: 2010-01-30.

[83] Oliveira, H., Murta, L., and Werner, C. Odyssey-VCS: a Flexible Version Control System for UML Model Elements. In *Proc. of the 12th International Workshop on Software Configuration Management* (2005), ACM, pp. 1–16.

[84] Oliveira, K., and Oliveira, T. A Guidance for Model Composition. In *Proc. of the International Conference on Software Engineering Advances* (2007), IEEE Computer Society, p. 27.

[85] Over, R. Collaborative Research and Publication in Psychology. *American Psychologist 37*, 9 (1982), 996–1001.

[86] Price, D. R., and Ximbiot. www.nongnu.org/cvs. Last visit: 2009-11-23.

[87] Prinz, W., Loeh, H., Pallot, M., Schaffers, H., Skarmeta, A., and Decker, S. ECOSPACE Towards an Integrated Collaboration Space for eProfessionals, 2006.

[88] Ramirez, A., Vanpeperstraete, P., Rueckert, A., Odutola, K., Bennett, J., Tolke, L., and van der Wulp, M. *ArgoUML User Manual: A Tutorial and Reference Description*, 2001.

[89] Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W., and Kotsis, G. Models in Conflict—Detection of Semantic Conflicts in Model-Based Development.

[90] Robillard, P. N., and Robillard, M. P. Types of Collaborative Work in Software Engineering. *Journal of Systems and Software 53*, 3 (2000), 219–224.

[91] Robinson, W. N., and Volkov, V. Supporting the Negotiation Life Cycle. *Communications of the ACM 41*, 5 (1998), 95–102.

[92] Ropa, A., and Ahlström, B. A Case Study of Multimedia Co-Working Task and the Resulting Interface Design of a Collaborative Communication Tool. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems* (1992), ACM, pp. 649–650.

[93] Schmidt, D. C. Model-Driven Engineering. *Computer 39*, 2 (2006), 25–31.

[94] Schmidt, M., Wenzel, S., Kehrer, T., and Kelter, U. History-Based Merging of Models. In *Proc. of the ICSE Workshop on Comparison and Versioning of Software Models* (2009), IEEE Computer Society, pp. 13–18.

[95] Schneider, C., and Zündorf, A. Experiences in Using Optimistic Locking in Fujaba. *Softwaretechnik Trends 27*, 2 (2007).

[96] Secretan, J., Beato, N., D'Ambrosio, D. B., Rodriguez, A., Campbell, A., and Stanley, K. O. Picbreeder: Evolving Pictures Collaboratively Online. In *Proc. of the 26th Annual SiGCHI Conference on Human Factors in Computing Systems* (2008), ACM, pp. 1759–1768.

[97] Semenov, V. Collaborative Software Engineering Using Metamodel-Driven Approach. In *Proc. of the 16th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (2007), IEEE Computer Society, pp. 178–179.

[98] Sidner, C. L. The Role of Negotiation in Collaborative Activity. Tech. rep., AAAI FS-93-05, 1993.

[99] Skype. http://skype.com. Last visit: 2009-12-09.

[100] Sourceforge. http://sourceforge.net. Last visit: 2009-10-27.

[101] Spinellis, D. http://www.spinellis.gr/tot/vcs-glossary.html. Last visit: 2010-01-10.

[102] Spinellis, D. Version Control Systems. *IEEE SOFTWARE 22*, 5 (2005), 108–109.

[103] Streitferdt, D., Wendt, G., and Nenninger, P. Model Driven Development Challenges in the Automation Domain. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference* (2008), IEEE Computer Society, pp. 1372–1375.

[104] Tang, J. C., and Isaacs, E. Why Do Users Like Video? Studies of Multimedia-Supported Collaboration. Tech. rep., Vol. TR-92-5. Sun Microsystems, Inc., 1992.

[105] Teamspeak. http://www.teamspeak.com. Last visit: 2010-01-07.

[106] Thum, C., Schwind, M., and Schader, M. SLIM—A Lightweight Environment for Synchronous Collaborative Modeling. In *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems* (2009), Springer-Verlag, pp. 137–151.

[107] Tragatschnig, S. Ein Ansatz zur Auflösung von Konflikten bei der Versionierung von Modellen. Master's thesis, Vienna University of Technology, 2009.

[108] TS-Overlay. http://www.teamspeakoverlay.com. Last visit: 2010-01-07.

[109] Ventrilo. http://www.ventrilo.com. Last visit: 2010-01-08.

[110] Vessey, I., and Sravanapudi, A. P. CASE Tools as Collaborative Support Technologies. *Communication of the ACM 38*, 1 (1995).

[111] Walz, D. B., Elam, J. J., and Curtis, B. Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration. *Communication of the ACM 36*, 10 (1993), 62–77.

[112] Whitehead, J. Collaboration in Software Engineering: A Roadmap. In *Proc. of the International Conference on Software Engineering, Future of Software Engineering* (2007), IEEE Computer Society, pp. 214–225.

[113] Wikipedia. http://en.wikipedia.org. Last visit: 2009-11-03.

[114] Wong, F., Fernandez, G., and McGovern, J. CONFER: Towards Groupware for Building Consensus in Collaborative Software Engineering. In *Proc. of the 8th Australasian Conference on User Interface* (2007), Australian Computer Society, Inc., pp. 31–38.

[115] Zhu, L., and Liu, Y. Model Driven Development with Non-Functional Aspects. In *Proc. of the ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design* (2009), IEEE Computer Society, pp. 49–54.

[116] Zhu, L., Liu, Y., Gorton, I., and Kuz, I. Tools for Model Driven Development. In *Proc. of the 40th Annual Hawaii International Conference on System Sciences* (2007), IEEE Computer Society, p. 284.