Leopold-Franzens-University Innsbruck

Institute of Computer Science
Research Group Quality Engineering

MASTER THESIS

Collaborative Process Modeling for Cheetah Experimental Platform

Peter Zangerl

Supervisor: Assoc. Prof. Dr. Barbara Weber

Innsbruck, April 8, 2013

Abstract

Business process modeling widely is used as a tool to represent business processes in a formal and generic way in today's industries and organizations. Various different tools and solutions are readily available, which enable the creation and management of business process models.

However, much fewer research has been done in the area of *how* these business process models are created, and how the process of creating process models can be improved even further. To analyze the *process of process modeling* in more detail, *Cheetah Experimental Platform* has been developed. It records each and every step of a modeling session, and can replay the whole creation process later on. A detailed analysis of all the changes performed by the user can give more insight into how process models evolve over time.

Meanwhile, more and more tools enable the collaborative creation of business process models. This gives a whole group of users the possibility to work together on a process model and thereby presumably improving the overall quality of the model, since it is easier to have all the needed domain experts involved in the creation of a process model.

In analogy to the single user setting, there is still a lack of research on how process models can effectively be created collaboratively. This motivated extending Cheetah Experimental Platform to allow a group of users to create a model collaboratively, while recording every change to the model for later analysis. This will give more insight into the *process of collaborative process modeling* and thus help developing new and better tools for collaborative modeling.

We therefore extended Cheetah Experimental Platform to add support for collaborative modeling. Our extensions should make it easy for a group of

modelers to work together and create a process model collaboratively. We focused on providing the users with a convenient built-in solution for communication and presenting them with additional information supporting the efficient collaborative creation of process models. Finally, the resulting implementation was evaluated using the *Technology Acceptance Model*, in order to assess it's *perceived ease of use* and *perceived usefulness*.

Acknowledgment

I would like to thank **Assoc. Prof. Priv.-Doz. Dr. Barbara Weber** for supervising this thesis. Her valuable support and feedback throughout all the phases of this thesis enabled the development of my implementation.

Furthermore, I would like to thank **Simon Forster M.Sc.**, **Dipl.-Ing. Stefan Zugal** and **Jakob Pinggera M.Sc.** for their help and participation in productive discussions.

Last but not least I would like to thank all of **my family and friends** for all their continuous support and encouragement throughout the time of my study and this thesis.

Declaration of Authorship

I, Bakk.techn. Peter Zangerl, declare that this thesis and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- No part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Date

Signature (Peter Zangerl)

Contents

Abstract iii

Acknowledgment v

Declaration of Authorship vii

Contents ix

1 Introduction 1

1.1 Background 1
1.2 Research Objective 2
1.3 Research Method 3
1.4 Related Work 4
1.5 Structure 5

2 Collaborative Process Modeling 7

2.1 Collaborative Software 7
2.1.1 Classification 8
2.2 Functional Requirements 9
2.2.1 Awareness 9
2.2.2 Communication 11
2.2.3 Coordination 12
2.2.4 Group Decision Making 12
2.2.5 Team-Building 13

2.3 Technical Requirements 14
2.3.1 Sharing of Data 14
2.3.2 Persistence 15
2.3.3 Versioning 16
2.3.4 System Independence 17

3 Implementation 19

3.1 General Overview 19
3.2 Architecture 20
3.2.1 Cheetah Experimental Platform 21
3.3 Extensions to Cheetah Experimental Platform 23
3.3.1 Logging Extensions 23
3.3.2 Revision Object 24
3.4 Collaboration Specific Features and Extensions 26
3.4.1 Client-Server Architecture 26
3.4.2 Client-Server Communication 26
3.4.3 Generic Server 32
3.4.4 Graphical User Interface Overview 34
3.4.5 User Colors 35
3.4.6 Commit History View 36
3.4.7 Messages View 37
3.4.8 Selection Highlighting 39
3.4.9 Linked Messages 40
3.4.10 New Node Feedback 42
3.4.11 Node Tooltips 44
3.4.12 Active Users View 45
3.4.13 Conflict Management 46
3.4.14 Feature Summary 49

4 Evaluation 51

4.1 Method 51
4.2 Results 52

4.2.1 General Questions 53
4.2.2 Feature Usage 54
4.2.3 Messages View 55
4.2.4 Node Tooltips 56
4.2.5 User Colors 57
4.2.6 User Feedback and Suggestions 58

5 Summary 59
5.1 Conclusion 59
5.2 Outlook 60

List of Figures 63

List of Tables 65

Bibliography 67

Chapter 1

Introduction

A *business process model* is a formal representation of all the activities of a business process, along with all participating entities and their role for the execution of this business process. Additionally it also describes the control flow between the different activities [IRRG09]. To make these process models easily understandable for all involved stakeholders, they are most often expressed graphically.

1.1 Background

Traditionally, a process model is created by a system analyst working together with one or more domain experts. The domain experts have all the knowledge needed to create an informal description of the business process to model, while the system analyst knows how to correctly transform informal descriptions into formal process models [FvdW06, HPvdW05].

These participants work together and discuss the business process to model—often divided into repeated iterations of elicitation-, modeling-, verification- and validation phases. During elicitation, information is collected to create an informal specification of the business process. The modeling phase then transforms this informal specification into a formal representation of the business process, which can then be validated against the specification and verified for internal consistency [FvdW06, MRW12].

As opposed to small example process models, real life organizations often have business processes of much higher complexity. The complete knowledge covering a business process might be distributed among several domain experts, possibly spread around the globe. Often the knowledge of multiple experts combined will be needed to create a complete model [MRW12].

Additionally, some of the decisions to be made during the creation and validation of a process model may also be too critical to be taken by just one single domain expert. Having multiple experts working on the same model helps reducing the risk of wrong decisions [MRW12].

Since the collaboration between all participants creating a model collaboratively presumably has an impact on the quality of the resulting business process model, the process of model creation itself along with the interactions between the modelers offers room for further research [PZW+12, CVP+12, CVR+12, PSZ+12, PFM+13].

1.2 Research Objective

The goal of this thesis was to implement a tool, which allows a group of users to work on a process model collaboratively. The tool should provide the group with all the needed functionality to efficiently work together and achieve a common goal.

Our implementation is built on top of *Cheetah Experimental Platform* (CEP) [PZW10], which provides all the needed functionality for a process modeller. Additionally it also enables the replay of a model's progress and thereby gives insight into the *process of process modeling*.

As stated by Pinggera et al., the process of process modeling still offers some room for scientific research [PZW+12]; So following the spirit of CEP, our tool should also give insight into how the collaboratively developed model evolved over time, and thus allow to analyze the *collaborative process of process modeling* in more detail.

Research in other areas—like software engineering—has shown, that collaboration can increase quality and efficiency significantly [WKCJ00]. By enabling a more detailed analysis of the collaborative process of process modeling, we hope to gain more insight into similar effects and results in the domain of process modeling.

Some of the features realized by our implementation *"may change the very nature of process modeling"* [DC11]—like collaborative editing of a model, while also retaining the communication log and providing the user with a list of all the currently online users.

1.3 Research Method

Hevner et al. developed the *Information Systems Research Framework* to evaluate, understand and execute information systems research. The framework incorporates paradigms from *behavioural science* (i.e. *"develop and verify theories that explain or predict human or organizational behavior"*) and *design science* (i.e. *"extend the boundaries of human and organizational capabilities by creating new and innovative artifacts"*) [HMPR04].

The research method used during the course of this thesis followed the information systems research framework. We went through repeated iterations of *development/build* and *justification/evaluation* phases. During each development phase, our implementation was extended and refined to meet the given requirements. Afterwards—in the evaluation phase—the implementation was assessed and evaluated for how well it satisfies the given requirements. The results of the evaluation phase were then used to create new requirements and refine the existing ones for the next development iteration.

1.4 Related Work

Current research on the process of process modeling tries to understand the relationship between how a process model is created and the quality of the resulting models. Different approaches are followed to analyze this relationship and gain more insight into it's effects on model quality.

Some approaches use different graphical representations of the user interactions during a modeling session, in order to recognize different patterns and strategies used by modelers [PZW$^+$12, CVP$^+$12, PSZ$^+$12]. Pinggera et al. gathered data on the the eye movements of users during model creation [PFM$^+$13], while Claes et al. try to relate model structuring, element movement and speed to model quality [CVR$^+$12].

As of today, more and more software solutions support collaborative process modeling—at least to some extent [dH05, DOV00, MRW12]. Still, the collaborative process of process modeling cannot be analyzed effectively with these tools.

The following examples just give a rough overview of some academic prototypes. Presenting an exhaustive list of collaborative process modeling tools (including proprietary software systems) would be beyond the scope of this thesis.

The TeamSCOPE System developed by Steinfield et al. provides a group with a common file space, message boards, mail functionality chat rooms and calendar functionality [SJP99]. The files managed by the system can be accessed by various means like web browsers, SSH[1] or FTP[2].

Dori et al. developed *OPCATeam*—a tool for collaborative business process modeling in OPM[3] notation. The editor allows a group of users to work on a model concurrently and also provides permission management for model elements [DBT04].

[1] Secure Shell (SSH): `http://tools.ietf.org/html/rfc4252`
[2] File Transfer Protocol (FTP): `http://tools.ietf.org/html/rfc959`
[3] Object Process Methodology (OPM) [DC99]

The *COMA tool* (COllaborative Modeling Architecture) allows for collaborative UML[4] modeling, where the participants can also give their support for the current design or propose alternatives [Rit08, Rit09, Rit10].

Thum et al. [TSS09] provide a group of users with a web-based editor to collaboratively model UML diagrams, while Farwick et al. [FAW+10] use a similar approach to enable web-based collaborative modeling of domain specific languages.

Another web-based solution for collaborative business process modeling is *Oryx*[5], which has been developed at the university of Potsdam. This editor has been developed as an open source project, but has been discontinued as of 2011. The commercial process modeling tool *Signavio*[6] is based on Oryx and adds multiple extensions to it.

Gravity[7] is the prototype of another web-based collaborative business process modeling editor developed by *SAP*[8] *Research*. It is based on *Google Wave*[9], which has been shut down in 2012.

1.5 Structure

This chapter gave a brief overview of the goals of our implementation. The rest of this thesis is structured as follows:

Chapter 2 discusses the concept of collaborative process modeling in more detail. The functional requirements for collaborative modeling are explained—together with their derived technical requirements.

[4] Unified Modeling Language (UML): `http://www.uml.org/`
[5] The Oryx Project: `http://bpt.hpi.uni-potsdam.de/Oryx/`
[6] Signavio Process Editor: `http://www.signavio.com/en/bpmn.html`
[7] Gravity—Collaborative Business Process Modelling within Google Wave: `http://scn.sap.com/people/alexander.dreiling/blog/2009/09/02/gravity-collaborative-business-process-modelling-within-google-wave`
[8] SAP AG: `http://www.sap.com/`
[9] Google Wave: `http://wave.google.com`

Our concrete implementation for a collaborative process modeling tool is presented in Chapter 3. This chapter will also introduce Cheetah Experimental Platform on which our implementation is based. Additionally it will highlight the extensions we had to perform in order to fulfill the given requirements, and discuss the challenges we faced, accompanied by their solutions.

Finally, Chapter 4 will present the results of a small evaluation we performed in order to assess the ease of use and the perceived usefulness of our implementation and how well it can support a team on modeling a process collaboratively.

Chapter 2

Collaborative Process Modeling

This chapter will give a general overview on collaborative software. Different classification criteria for collaborative systems will be discussed along with criteria which can be used to compare different systems. We will also introduce functional and technical requirements for collaborative tools in the context of process modeling.

2.1 Collaborative Software

Collaborative software—often referred to as *groupware* or *computer supported cooperative work (CSCW)*—covers a very broad area of systems which aim to help multiple participants to coordinate their efforts and work together to achieve a common goal. Collaborative software provides a group with different services to support it's work. These services include various forms of communication (e.g. message board, chat, voice, video), shared data (e.g. edit files as a group, knowledge database), versioning of data and group management (e.g. add/remove members from a group).

Depending on the requirements and the implementation, collaborative systems can be classified according to different criteria (see Section 2.1.1). The most important requirements of groupware applications will be discussed in detail in Section 2.2 and Section 2.3.

2.1.1 Classification

To define requirements for collaborative software and to enable comparison of different approaches, systems may be classified according to various criteria [BM02]. One of the most generic approaches [GF08] classifies different CSCW systems according to:

Time: A system can be either *synchronous* or *asynchronous*. Synchronous systems allow cooperation between individuals using the system at the same time, while asynchronous systems enable it's participants to work together even though they do not do so at the same time.

Space: Group members may be *co-located* (physically at the same location) or *distributed* over a large—possibly global—area.

A CSCW system is then assigned to one or more of the four possible combinations of these criteria (see Table 2.1). Assigning a specific system to one of these groups may not always be easy. Often a groupware system will support different settings and may therefore belong to multiple categories (e.g. by offering synchronous and asynchronous communication).

		Time	
		synchronous	**asynchronous**
Space	**co-located**	synchronous and co-located	asynchronous and co-located
	distributed	synchronous and distributed	asynchronous and distributed

Table 2.1: Classification scheme for CSCW systems

Bafoutsou and Mentzas analyzed multiple research papers and summarized different classification methods in use. These range from technical aspects of the described systems (i.e. hardware and software) to group characteristics like group size [BM02].

A completely different approach to classify CSCW systems is to compare them according to the requirements they satisfy (i.e. the functionality they offer—see Section 2.2). Some systems serve a very specific task while others may provide a group with many useful features.

In the domain of collaborative process modeling, the work may be done synchronously or asynchronously. Since the required stakeholders and experts may share the same room or be globally dispersed, co-located as well as distributed settings should be supported.

Consequently, a tool for collaborative process modeling should ideally support every possible combination of the above mentioned classifications (see Table 2.1).

2.2 Functional Requirements

Mendling et al. define collaboration between multiple individuals as social interactions on different levels [MRW12]. These levels of social interaction are: *awareness*, *communication*, *coordination*, *group decision making* and *team-building*.

The following sections will describe these levels of social interaction in more detail and link them to their role in process modeling.

2.2.1 Awareness

Today's advances in communication technologies foster geographically distributed groups working together on a project. Team members use various tools to communicate with each other (see Section 2.2.2) and to share common data (see Section 2.3.1). A big problem for team members, which are distributed across multiple locations and time zones, is to keep track of each others work [SJP99].

In CSCW systems, *awareness* designates the information about activities performed by the other members of a group. Ideally, each member should always know which other group members are currently working on the project, and what

they are doing at the moment. This reduces conflicts and helps to achieve a better overall result more efficiently.

Even though awareness is an important factor for group success, many existing applications do not support it well. This has three main reasons:

- Generating awareness data is still a challenge. Automatic generation of awareness data is not always possible and the user should not have to manually create awareness data for every single step he takes.
- Different applications are used for different tasks. Maybe some of these applications will offer awareness information at least to some extent, but others will not support this at all. A central awareness notification functionality would be desirable.
- Especially for larger groups, presenting a multitude of awareness data to the user may easily overload him. There has to be some kind of filtering to extract just the information which is relevant at the moment.

An example for an application providing users with awareness information is the "TeamSCOPE System" developed by Steinfield et al. [SJP99]. This system automatically gathers awareness information (e.g. added or modified files, new calendar entries, group messages) and makes these available for all the team members.

Awareness in Process Modeling

Obviously the most important awareness information during process modeling are model changes. Each and every user working on a model should see all changes on that model made by others immediately. This mitigates misunderstandings and helps reduce conflicts, while also improving modeling efficiency.

When doing process modeling collaboratively, it is important for each group member to know which other users are online at the same time. This enables the users to directly communicate (see Section 2.2.2) with them in case some

complications arise. Furthermore, awareness also forms the basis for coordination of groups (see Section 2.2.3) and team-building (see Section 2.2.5) [MRW12].

2.2.2 Communication

Communication is a key aspect for the success of a group. This is even more the case for geographically distributed teams, since it's members cannot coordinate their efforts effectively without communication [SJP99, FAW+10].

The widespread availability of the *Internet* soon made *email* one of the most used services. Email offers cheap (i.e. the costs do not depend on the distance and the number of recipients) and fast asynchronous communication between an arbitrary number of team members.

With the availability of cheap and fast Internet connections, new applications demanding higher bandwidths were introduced, including *Voice over IP* (VoIP) and *video chats*. Today various software solutions exist to support communication between distributed team members all around the world. The main differences between these systems and early phone-based audio- and video conference systems are additional features (e.g. like awareness—see Section 2.2.1) and associated costs.

Depending on the team's needs there is a multitude of applications with different features to choose from. Skype[1] is a prominent example, offering awareness information (e.g. online status and status messages), synchronous and asynchronous message exchange, file exchange as well as direct and conference audio- and video communication.

Communication in Process Modeling

For efficient collaborative process modeling, communication is a key requirement. It enables a group to coordinate their efforts (see Section 2.2.3), agree on common terminologies, and work towards the same goal [MRW12].

[1] Skype: `http://skype.com`

A tool for collaborative process modeling should have built-in communication support, which enables it's users to easily and directly communicate with each other in an unobtrusive way (i.e. they should not need to switch between different applications). Ideally, all communication should be retained for later review, which helps clarifying discussions and understand the reasoning behind design decisions in the final process model.

2.2.3 Coordination

When working as a group there is a multitude of things to *coordinate*. A team has to find consent on what they want to eventually achieve, how they want to do it and who will be responsible for what.

A collaborative system can support the group and ease the management of things like:

- Appointments, meetings and schedules [SJP99].
- Scarce resources, like special equipment or rooms supporting video conferences [SJP99].
- Help on distributing work among team members [DOV00, dH05].

Communication (see Section 2.2.2) is a key factor for coordinating the efforts of a group—especially when doing process modeling. By investing time to coordinate a shared task, the overall process will be more efficient [MRW12].

2.2.4 Group Decision Making

When working on a project as a group, there may be a lot of decisions to be made. The group has to find a consent on what they want to achieve, how they will do it and what alternatives there are to consider. This aspect is called *group decision making* [MRW12].

Often there are various ways to solve a certain problem resulting from different knowledge and opinions from each participant. These alternatives have to be discussed and evaluated to eventually decide on which will be chosen.

During the development of a process model, but also during it's validation phase, there may be a lot of decisions to be made. A CSCW system might help a group on finding majorities to accept or reject new proposals, or may provide the group with some form of voting system, which helps them to decide which alternative they will eventually implement [Rit09, Rit10, Rit08].

Additionally, there may be situations, where a decision might be too important to be made by just a single group member. In these cases it is crucial to consult the whole team and finally decide on how to proceed [MRW12].

2.2.5 Team-Building

Team-building describes the social process of team members identifying themselves with the project they are working on. Ideally they feel responsible for their tasks and the results they produce.

Group decision making (see Section 2.2.4), awareness (see Section 2.2.1) and communication (see Section 2.2.2) are very important foundations for effective team-building, but still the importance of team-building itself may vary, depending on the organizational setting [MRW12].

In the context of process modeling, team-building is an important factor. As each group member contributed to the final result and the discussions which took place along it's creation, every member is responsible for the outcome. Each member of the group has to be able to identify himself with the resulting process model.

2.3 Technical Requirements

To satisfy all the social interaction levels for collaborative software (i.e. functional requirements; see Section 2.2) in our implementation, there are some additional *technical requirements* to fulfill.

2.3.1 Sharing of Data

There exist some collaborative applications which do not share data between participants—e.g. by providing just awareness (see Section 2.2.1) or communication (see Section 2.2.2), but most often *sharing of data* is the main requirement for collaborative software. Every time a piece of information is needed by more than just one group member, it has to be made available to the group somehow [Rit10, SEA+02, SJP99, LDV97, FAW+10, TSS09].

Shared data may be anything useful to the group and can be available in many different forms. It may be a simple address book, calendar information, the source tree of a program in development or the complete set of blueprints for each part of an airplane a company builds.

Without the use of shared data, a user needing some specific information has to know who possesses it and has to request it explicitly—which may require a lot of time, forcing the user to wait for a reply. Additionally, exchanging all these information by sending them directly to the team member who needs them (e.g. via email) is cumbersome and leads to inconsistencies, conflicts and misunderstandings.

A CSCW system can mitigate these problems by providing a group with a common shared space, where they can store their data. All the data within this space can then be directly accessed by all the group members without first having to ask and wait for it.

Examples of collaborative systems for sharing data include calendar applications, shared screen applications—where all members of a group see the same content on their screen—and group editors, which allow a group to edit a single file

simultaneously. Networking file-systems (e.g. NFS[2] or SMB/CIFS[3]) are prominent examples for systems offering sharing functionalities for generic files. They allow multiple users to access and manipulate shared files over a local network or the Internet.

Sharing of Data in Process Modeling

In the context of process modeling, sharing of data is the most important technical requirement, since it enables all team members to have a common view of the process model under construction. Every user sees the same data, and every change is immediately visible for all participating users (see Section 2.2.1).

This common view on the model under construction enables rapid feedback and discussions, which helps improving the overall model quality. Communication (see Section 2.2.2) can also be seen as sharing of data between the participants, and thus this feature is essential to satisfy all the functional requirements introduced above (see Section 2.2).

2.3.2 Persistence

Sharing of Data (see Section 2.3.1) virtually always implies *persistence* of that data. There are just a few systems to mention as exceptions—meaning that they provide the group with common data but discard all this data once the current session ends.

An example for such a system would be shared screen applications. These applications can help in meetings (either co-located or distributed—accompanied by some remote communication system) giving all participants access to the same screen image. The moderator (which can be any team member, and of course can change during a meeting) is the only one making changes, and his screen image is

[2]Network File System (NFS): `http://tools.ietf.org/html/rfc5661`
[3]Server Message Block / Common Internet File System (SMB/CIFS): `http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx`

then copied to all the other users. This can help to better explain the currently discussed matters, but may not be of any value once the meeting is finished.

In contrast to applications mainly intended for sharing of data, communication systems often—depending on the form of communication used—do not provide persistence in any form (see Section 2.2.2). While text-based conversations can be easily stored for later reference and review, this may not be feasible for audio- or video-streams.

Persistence in Process Modeling

In the context of process modeling, there is no doubt that persistence is a vital requirement. Developing a process model without being able to store it somehow would be pointless.

Still, some systems may not persist all the data produced during modeling sessions. As an example, some systems might not store communication between the participants—either because it is not considered necessary or because it would not be feasible to do so (e.g. storing audio- or video communication and assigning different excerpts to modeling steps may not be possible).

In general it is advantageous to store as much information as possible during a modeling session, since it gives the possibility for more insight into the modeling process, and thus provide the team with more awareness data (see Section 2.2.1) or guidance for group decision making (see Section 2.2.4).

2.3.3 Versioning

For systems offering persistence (see Section 2.3.2), *versioning* of stored data may be a requirement. Versioning enables the individual user or the group to revert the shared data back to the state it had at some point in the past. Additionally, *undo*- and *redo* for more than just the last couple of operations may not be available in systems without versioning support.

Versioning all the changes also enables additional functionalities; When all the data is versioned, then it is easy to see which team member changed which part of a file, and when he did so. When some information has been deleted or modified accidentally it can be easily restored again.

Comparing different versions of a single piece of data and tracking it's changes is another feature enabled by versioning.

Some systems also allow for special versions to be created (e.g. when a project is considered stable or a certain milestone has been reached), often denominated as a *tag* or *branch*. These versions can then be used to derive a different branch and alter it independently of the main data.

A prominent example for systems offering versioning is CVS[4]. More recent examples are SVN[5] or Git[6]. All these systems enable it's users to work on a set of files together, while tracking all the changes performed.

Versioning in Process Modeling

In the context of process modeling, versioning also helps generating awareness data (see Section 2.2.1). Versioned data allows the user to review the history of the model under construction step by step. Additionally versioning enables the modeler to review the complete history of any single element in the model.

Versioning also helps to satisfy the requirement for team-building (see Section 2.2.5), since it enables more detailed analyses of the history of a process model under construction [MRW12].

2.3.4 System Independence

Especially in distributed environments the hardware and software configurations of the used workstations can vary greatly. These heterogeneous environments can

[4]Concurrent Versions System (CVS): `http://savannah.nongnu.org/projects/cvs`
[5]Apache Subversion (SVN): `http://subversion.apache.org/`
[6]Git: `http://git-scm.com/`

pose additional difficulties when selecting a system to use or implementing a new CSCW respectively.

Reducing the platform dependence to a minimum (e.g. by using a platform independent programming language) will allow for more flexibility of the resulting product. Often web-based solutions are used (e.g. [SJP99, TSS09, FAW+10]), since there are well defined standards supported by browsers available on practically all platforms, and no installation is required on the client side. Of course this solution comes at a price—limited flexibility in the user interface and practically no possibility to store large data on the client itself.

Even though *system independence* is not needed to satisfy the functional requirements introduce above (see Section 2.2), it is still often a requirement imposed on collaborative systems.

Chapter 3

Implementation

This chapter will give more details on the implementation that was realized as part of this master thesis. Following a brief general overview in Section 3.1, Section 3.2 will give an overview of the implementation's architecture, and also show how we based our work on Cheetah Experimental Platform. Section 3.3 will then show how we adapted and extended Cheetah Experimental Platform for our needs, while Section 3.4 will highlight the most important features and design considerations of our final implementation in more detail.

3.1 General Overview

Our implementation consists of two main parts—a server component and a client application. The server is responsible to manage multiple modeling sessions and to communicate the changes performed by the users to each associated client application instance (see Section 3.4.1 and Section 3.4.2). It has intentionally been kept small and simple, to make it more flexible and usable in different scenarios (see Section 3.4.3).

The client application provides a group of modelers with all the functionality they need in order to collaboratively create a process model. It's main component is the model editor, which shows the process model in it's current state and also allows the users to perform changes on the model (see Section 3.4.4).

Another important component of the client is the messages view (see Section 3.4.7), which enables the users to directly communicate with each other in the form of exchanging text messages. These messages can also be marked as related to specific model elements, and thereby get semantically linked to that elements. This provides additional data for model analysis and creates awareness information.

Additional views and functionalities have been implemented in order to provide the user with more awareness information during modeling sessions. These views help the user to get an overview of the changes performed by all the group members (see Section 3.4.6), or show all online users (see Section 3.4.12). Another notable awareness feature are user colors—a user selectable color, which will then be used to indicate all the user's changes in the model (see Section 3.4.5).

3.2 Architecture

Our implementation is built on top of Cheetah Experimental Platform (see Section 3.2.1). The application is implemented as a plugin for the *Eclipse Rich Client Platform*[1], and is written in *Java*[2]. We also use various third party libraries to ease the implementation burden of tedious and recurring tasks. An overview of the architecture of our implementation is depicted in Figure 3.1.

Our implementation stores all the data of a modeling session in an XML[3] file. The resulting model itself is not stored, but all the steps taken by all the user during it's construction. This enables a detailed analysis of the process of process modeling in concurrent settings. Of course this also implies that we have to store more information than just the changes themselves (see Section 3.3.1).

We chose XML as data format, since it offers a lot of flexibility—the files can be written and read by different programming languages using readily available

[1]Eclipse RCP: `http://www.eclipse.org`
[2]Java: `http://www.oracle.com/technetwork/java`
[3]Extensible Markup Language (XML): `http://www.w3.org/TR/rec-xml`

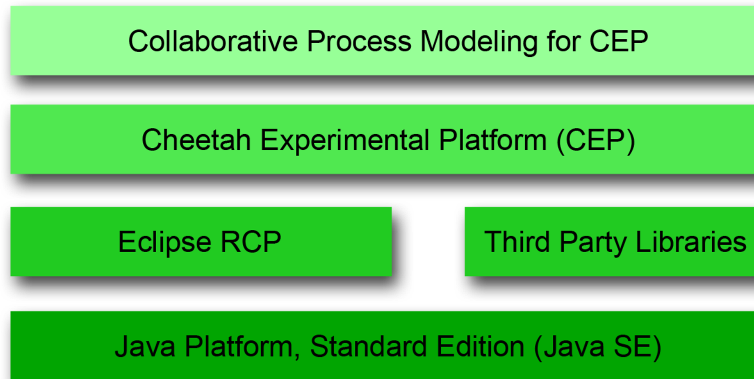

Figure 3.1: Architectural overview

libraries, they can be modified and analyzed using various scripts, and even easily be read by humans.

The requirement to provide system independence (see Section 2.3.4) was satisfied by deciding to build the implementation on top of Cheetah Experimental Platform. As mentioned above, Cheetah Experimental Platform is based on GEF[4], Eclipse RCP and Java, and therefore can be run on multiple platforms without the need to change or recompile the code.

3.2.1 Cheetah Experimental Platform

Cheetah Experimental Platform (CEP) was designed to enable a detailed analysis of the process of process modeling [PZW10]. CEP provides a single user with an editor interface to model processes. All the user's interactions with the editor are recorded and stored for later analysis.

To analyze the process of process modeling, CEP offers a replay feature, which allows to gain insight in how a model has been created and evolved over time[5].

[4]Eclipse Graphical Editing Framework (GEF): `http://www.eclipse.org`

[5]Too see a demonstration of CEP's replay feature, visit `http://cheetahplatform.org`

This helps reproducing the steps the modeler took in order to come up with a final model.

All the different kinds of user interactions in CEP during a modeling session can be seen in Table 3.1. Each row in the table represents a different kind of user interaction, and also a different type of entry which will be logged during a modeling session along with the attributes needed to replay this change later on.

User Interaction	**Attributes**	**Description**
CreateNode	Node type, name, position	Creates a new node
DeleteNode	ID	Deletes an existing node
MoveNode	ID, old position, new position	Moves a node
Rename	ID, old name, new name	Renames a node
ResizeNode	ID, old size, new size	Resizes a node
CreateEdge	Edge type, from node, to node	Creates a new edge
DeleteEdge	ID	Deletes an existing edge
ReconnectEdge	ID, old and new source and target	Reconnects an edge
MoveEdgeLabel	ID, old position, new position	Moves the label of an edge
CreateEdgeBendpoint	ID, position, index	Creates a new bendpoint
MoveEdgeBendpoint	ID, old position, new position	Moves a bendpoint
DeleteEdgeBendpoint	ID	Deletes a bendpoint

Table 3.1: Full list of CEP user interaction types

In CEP, all the user interactions shown in Table 3.1 are represented as *Command Objects* [GHJV94]. This way, any change can be *undone* and *redone* during replay, which enables the user to observe the model at any point during it's creation, and also move back and forth in time. Each entry in the log of a modeling session also contains a timestamp of the exact instant when the command has been performed by the user. This enables further analysis of the process of process modeling, as it allows to observe factors like modeling speed or thinking time between various steps.

3.3 Extensions to Cheetah Experimental Platform

Cheetah Experimental Platform has been developed for a single user setting. To enable a group of users to collaboratively work on the same model, some parts of CEP had to be extended. This section will describe the extensions we had to implement to store all the changes which take place during a modeling session. In order to also keep the communication between the users—along with additional awareness information—we introduced a new data structure to record all the data for later analysis.

3.3.1 Logging Extensions

In the single user setting of Cheetah Experimental Platform, it was sufficient to merely store the changes made by the user in a serialized form, along with the timestamp when this change has been executed. When working in a group, there are more events to record, in order to be able to reproduce the whole history of a model later on.

We do not just have to keep track of all the changes made to the model itself, but also have to store who performed a particular change. This enables us to provide the user with more awareness data (see Section 2.2.1).

In order to observe the creation of a model—just as it really happened—during the analysis, we also have to record which users were online at what time. This information allows to present the user with a list of all online users (see Section 3.4.12).

One of the most notable additions to the logging scheme from Cheetah Experimental Platform is the storage of communication between the participants (see Section 3.4.7). Every message is stored together with all the model changes. This enables us to reproduce the complete communication which took place between multiple modelers. Each message may also be linked to one or more model elements, and thus create even more possibilities for deeper analysis (see Section 3.4.9).

In order to store these diverse types of data, we came up with a *Revision Object* (see Section 3.3.2), which is a generic container for model changes. Our central Server keeps a list of all the revisions for a particular model under construction and can save this list to a file for later analysis.

3.3.2 Revision Object

Every change made by any user is stored as a new revision object by our central server. The server then broadcasts the newly created revision object to all the clients in order for them to update their data model accordingly (see Section 3.4.2).

Table 3.2 shows all the properties of our revision object, along with a description. All revision objects for a specific model are managed in a list on the server side—the central *repository*. Whenever the repository is stored in a file, the list of revision objects is transformed to XML and written to disk.

Property	**Description**
ID	A unique ID representing this particular revision
Timestamp	The date and time when this revision has been created
Username	The username of the user who performed this change
Type	A field indicating what kind of change this revision represents
Content	The actual content of the revision (Depends on the type)

Table 3.2: Full list of properties of revision objects

As already stated in Table 3.2, the content of a revision object depends on the type of the revision. All the different revision types can be seen in Table 3.3, along with a description.

Type	Description
CHANGE	A model change performed by the originating user
MESSAGE	A message sent from the originating user
JOIN	The originating user joined the modeling session
LEAVE	The originating user left the modeling session
COLOR_CHANGE	The originating user changed the color associated with him
CONFLICT	A conflicting revision

Table 3.3: Full list of all revision types

The content of a *CHANGE* revision is the CEP command, which represents the change. *MESSAGE* revisions carry a special message object with attached linked elements (see Section 3.4.9), while *COLOR_CHANGE* revisions (see Section 3.4.5) carry the new color. Revisions of type *JOIN* and *LEAVE* do not need to have any content, since the creating user already indicates which user left or joined the session (see Section 3.4.12). Finally, *CONFLICT* revisions do not carry any content too, as the conflicting change has been removed from the repository (see Section 3.4.13).

Note that revision objects are never changed after they are created by the server. There are just two exceptions to this rule—when messages are linked retroactively and when a conflict arises.

When messages between the participating users are linked to model elements retroactively (see Section 3.4.9), the content of a revision is changed. This feature was introduced in order to better track the discussions which took place during a modeling session. In addition to the change of the revision object itself, all active clients are notified of this change, so that they can update their repository accordingly.

In case of conflicting changes, the revision causing the conflict will be replaced with a special revision indicating this conflict. The conflicting revision is not simply removed from the repository, because this would imply a loss of information. Instead the conflict revision indicates, that a conflict happened, and whose changes caused it. Unlike for linking messages, the information that a conflict happened does not have to be broadcasted to all active clients (see Section 3.4.13).

3.4 Collaboration Specific Features and Extensions

This section will present some more extensions to Cheetah Experimental Platform and additional features we had to implement, in order to effectively support collaborative process modeling.

3.4.1 Client-Server Architecture

Because the software should support asynchronous collaboration and distributed environments (see Section 2.1.1), a central server instance—which is available at all times—had to be used. The server is responsible for managing multiple repositories, which the clients can access and modify. Clients cannot communicate directly with each other, so the server is responsible to synchronize and distribute any changes the clients performed.

As shown in Figure 3.2, each client has the same role within our setting. All the clients communicate directly with our generic server (see Section 3.4.3) by calling *Remote Methods* (see Section 3.4.2).

3.4.2 Client-Server Communication

There are various possibilities to realize the communication between the server and it's clients via a network. They range from a simple custom protocol built on top of plain Java-*Sockets*, via the utilization of some already established protocol (e.g.

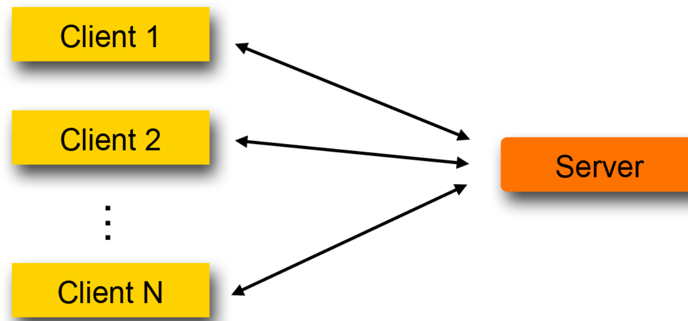

Figure 3.2: Client-Server architecture

HTTP[6]), to some high level abstraction like *CORBA*[7], *Web Services* or—especially on the Java Platform—*Remote Method Invocation* (RMI).

While low level protocols have their advantages when it comes to speed and low overhead, they also imply a big burden on the implementation. It is often much more effective to use an already established technology and thereby exploit it's advantages.

We decided to use RMI, since it is the protocol of choice, when communicating with other Java programs. It makes it easy to call methods on remote computers and takes care of all the low level details like call dispatching, parameter passing and error handling. The use of RMI enables our implementation to simply tell the other side of the communication what we want to do, and with what arguments this should be done, without worrying about the details how this request is to be transferred via the network.

However, as with most high level communication mechanisms, there is still the problem that our implementation needs bi-directional communication. Not only have the clients to connect to the server and interact with it, most of the time it is the server which has to send messages to the clients for them to process.

[6]Hypertext Transfer Protocol (HTTP): `http://tools.ietf.org/html/rfc2616`
[7]Common Object Request Broker Architecture (CORBA): `http://www.omg.org/spec/CORBA/Current/`

There are two basic mechanisms to send messages and notifications from an RMI server to a client:

Polling The client repeatedly calls a *remote method* on the server. In case the server has some command he wants to send to the client, this method will return the command—otherwise it returns a special return value indicating that nothing is to be done.

Callback The client too creates a *remote object*, and passes it to the server for later use. Once the server wants to send a command to the client it can call a remote method on this callback object which will then be executed on the client side.

Generally, a callback (or explicit notification) is to be preferred over polling, since polling is always a tradeoff between longer latencies and higher wasted network bandwidth. Calling the server too often will cause many calls to return with no new data—calling the server too rarely will increase the response time, so it will take longer until the client receives commands intended for him.

As indicated above, callbacks are a good alternative to polling. They are resource friendly and do not cause any additional artificial delays in command delivery. Using callbacks all the commands will get delivered immediately and no command will ever be sent without a reason. Callbacks are also easily implemented in RMI. The client can—just like the server—create a remote object with methods for the server to call in case a new command should be handled.

However, creating a remote object on the client will cause a new *ServerSocket* to be opened at the client side, on which the client will then be listening for incoming connections. In a local network this is not a problem, but since our implementation should also support co-located environments (see Section 2.1.1), this may be problematic.

We cannot assume that an arbitrary, Internet connected computer is reachable from the outside. Most consumer PCs have client side firewalls installed and enabled by default in their operating system, which will prevent connections from the Internet to local services. Additionally most consumer *residential gateways*

(i.e. home routers, DSL[8]- or cable modems) are configured to block any external connection in their firewalls too. Even worse—most often these devices perform NAT[9], which would require a special forwarding rule to be set up in order to reach the consumer PC from the Internet, even if the firewall(s) would otherwise permit external connections.

So to summarize, a user may not want to configure all the firewalls and routers between the PC and the Internet to allow incoming connections. In some situations he may not even be able to do so, because not all the nodes between the PC and the Internet can be configured by the user (e.g. corporate environment).

In case of residential gateways performing NAT, there may be even more complications if our implementations would run on multiple PCs sharing the same Internet connection, and thus having the same IP[10] address.

Since the problems mentioned above effectively render RMI callbacks useless, we had to use some other means to send commands to the clients. We also did not want to resort to polling because of the above mentioned drawbacks, so evenually we came up with the concept of *blocking polling*.

Blocking polling works just like polling, but the remote call on the server will block until new data is available. So unlike polling, we will never return a value telling the client, that nothing changed. Every time the method call actually returns from the server it has some useful data to process. Since no frequent calls will waste additional network bandwidth, we also do not have to introduce some artificial delay between server calls. We can immediately call the server again once we are done with processing of the last received command, which means we will not experience long latencies due to extra delays.

This approach has many advantages over callbacks. The biggest advantage of course, is that this solution does not require the clients to be able to accept connections from the Internet, and thus will work regardless of how the client is

[8] Digital Subscriber Line (DSL)
[9] Network Address Translation (NAT): `http://tools.ietf.org/html/rfc2663`
[10] Internet Protocol (IP): `http://tools.ietf.org/html/rfc791`

connected to the network. It also is just as fast as callbacks and does not waste any additional network bandwidth.

Figure 3.3 shows a sequence chart of how blocking polling works in our implementation. The client asks the server for new commands to process. At that moment there are no new commands for the client, so the call will simply block on the server side. Once the server enqueued some *Command A* (i.e. because some other client performed some changes), the call will return with that command. Meanwhile the server can enqueue commands *B* and *C*, so when the client later asks for the next command it can immediately return with the result.

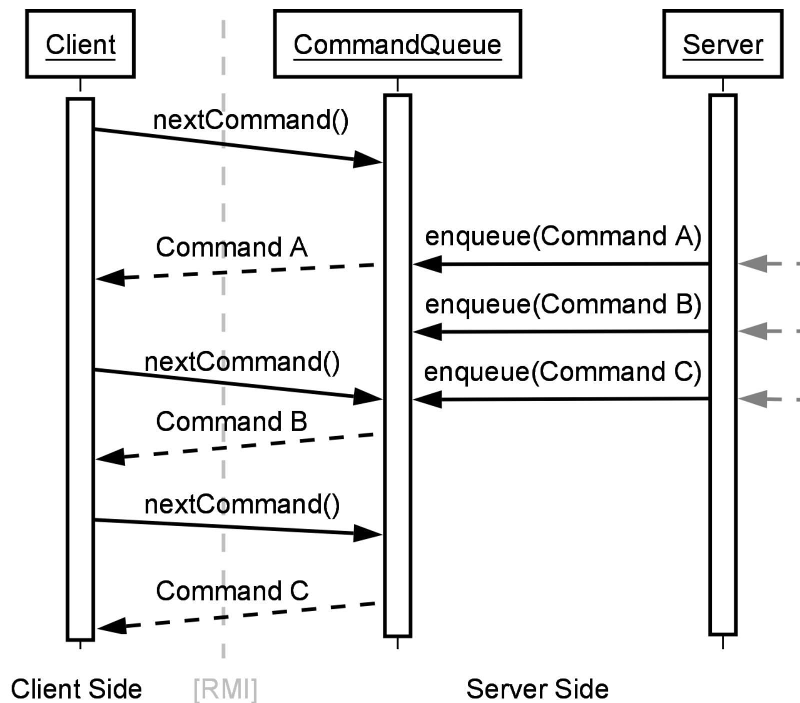

Figure 3.3: Blocking polling sequence chart

In contrast to callbacks, the client dictates the pace. A client will ask for the next command, when it is ready to process it (i.e. it is done with processing the last command). When using callbacks, the server will push messages to the client.

If the client is already busy, they then either have to be put into a queue (which requires more complex code on the client side), or the server thread will block until the client can accept the command. This will require the use of *worker threads* on the server side in order to keep the server responsive to new requests.

Using blocking polling, the server does not have to deal with worker threads to dispatch client notifications. The commands for each clients are simply put into a *Blocking Queue* [Blo08, PGB+05], where they can then be retrieved by the client. Whenever the queue is empty the client call will automatically block until a new command is added to the queue.

This eased thread management comes at the price of more complexity when sending different types of commands. Using direct callbacks the server can just call the appropriate client side method for the request. Since we are using a queue now we have to store the request and the associated data in the queue until it is retrieved by the client. The dispatching of different types of server notifications is now done at the client side. To store the server command together with its data, and to dispatch this command to the correct method on the client side, a combination of the *Command Pattern* and the *Strategy Pattern* [GHJV94] is used.

Another disadvantage of blocking polling is that the server cannot reliably detect if a client crashed abruptly. Using direct callbacks, the RMI subsystem will raise an *Exception* when the server calls a method on a client which no longer is running. The exception can be caught and the server will immediately recognize the client's disappearance. Using blocking polling the server will not recognize an unresponsive client. New commands are simply put in the queue for this client, regardless of whether or not the client is still alive.

One way to recognize unresponsive clients would be to limit the number of elements the command queue can hold. When the server then tries to add a new element and the client did not remove elements for a while, the server would recognize, that the client is not active anymore. This solution may of course not be very exact, since a slow client may not be able to keep up with a server producing a lot of commands for a while, and thus be considered unresponsive.

Another approach would be to update a timestamp every time a client retrieved it's last command. Whenever a client does not return for the next element after a sufficient amount of time, it can be considered unresponsive. This solution would not cause false assumptions from the server when a reasonable time limit is used.

We chose to detect unresponsive clients by means of a limited command queue size. Despite it's drawbacks, we decided for this alternative because it is very simple to implement on top of Java's *bound queue* implementations. Given a reasonable queue size, this approach works very well.

In case further testing would show problems with wrongly detected unresponsive clients in the future, it would be easy to switch to the more robust timestamp based detection method, as this change just affects the server.

A small variation worth considering may be the use of a *Priority Queue* in case prioritization of commands would be needed. In this case, even slow clients would read high-prioritized commands before unimportant ones.

3.4.3 Generic Server

When implementing a client-server architecture like we did, one has to make a choice between:

Thin Client A thin client usually just establishes a connection to a server, which then performs most of the work. The client cannot do anything for itself and completely relies on the server. Clients forward all the user's input to the server, which will then react and give the client something to display to the user in response. Thin clients are thus usually rather small and can be easily adapted to run on different kind of hardware and software. On the downside they may produce a higher load on the server and will need more network bandwidth to communicate with the server. The fact that the server reacts on user input may also cause noticeable delays for the user.

Fat Client A fat client usually has most of the core functionality implemented itself, and does not shift much of it's work to the server. The client reacts on

user input and performs the needed changes. The server then just handles all the communication between the clients. The main advantages of fat clients are faster response time and reduced network usage. The disadvantages are that a more complex client maybe cannot be adapted to all the platforms it should run on, and some work has to be done on both sides—the clients and the server (e.g. validity checks).

Of course we are not restricted to choose one of those extremes mentioned above, but can also create a solution which lies somewhere in between. Since our implementation is based on Eclipse RCP (see Section 3.2) we chose a fat client, with a very small and generic server implementation.

We decided against a complex server—which does all sort of validity checks on the model changes the clients send him—because this would imply losing a lot of flexibility. In keeping the server small and generic, and implementing just the bare minimum of required functionality, we can use it for different kinds of clients. Since the server does not know about the internal structure of the model edited by the clients, different types of clients (i.e. a client using some other modelling notation) can connect and take advantage of the services offered by the server implementation.

Of course such a simple server also has it's drawbacks. Since the server does not check the changes sent by clients for validity, it will also broadcast conflicting and illicit changes to all the clients (see Section 3.4.13). This means that the clients themselves must not assume anything about the data coming from the server, and thus have to perform their own validity checks when executing the changes.

However, even a small server implementation has to perform some modifications to the data coming from clients and has to offer some additional functionality. It has to manage multiple modelling sessions, and hold the respective repositories of changes. These repositories have to be persisted to disk on shutdown and restored on startup.

In our implementation, all change commands sent by the clients (see Section 3.3.2) are treated uniformly. The only exception are commands, which create a new model element. In order for all the clients to be able to address each object in the same way, we have to assign each object a *unique ID*. Every time a client creates a new object, the server will change this command and assign the current revision-ID to the newly created object. Since the revision-ID is a running number and incremented with each change, newly created objects are assigned unique IDs.

Along with these persistent revisions, the server also broadcasts different types of volatile data to it's clients. These changes are of no use in the repository itself but of interest to the clients currently connected to the server. This information is needed to provide the user with various awareness data like visual feedback on node creation (see Section 3.4.10) or selection highlighting (see Section 3.4.8).

Our server also keeps track of all connected users. The clients have no direct way to ask the server for a list of currently active users, but they can gather this information themselves (see Section 3.4.12), since the server will create presence notifications (i.e. special join- and leave-revisions) every time a user connects or disconnects (see Section 3.3.2).

3.4.4 Graphical User Interface Overview

Before giving more details about the additional features and extensions, we want to show a general overview of the GUI[11] of our implementation. This enables us to easily reference specific parts of the user interface in the following sections.

Figure 3.4 shows a general overview of the user interface. The main section of the user interface is the editor area which is shared between all participating users. On the right hand side there is the tool palette, which allows the user to add different types of nodes and edges to the model. Below the editor area there are three views showing all modifications since the session was started (see

[11] Graphical User Interface (GUI)

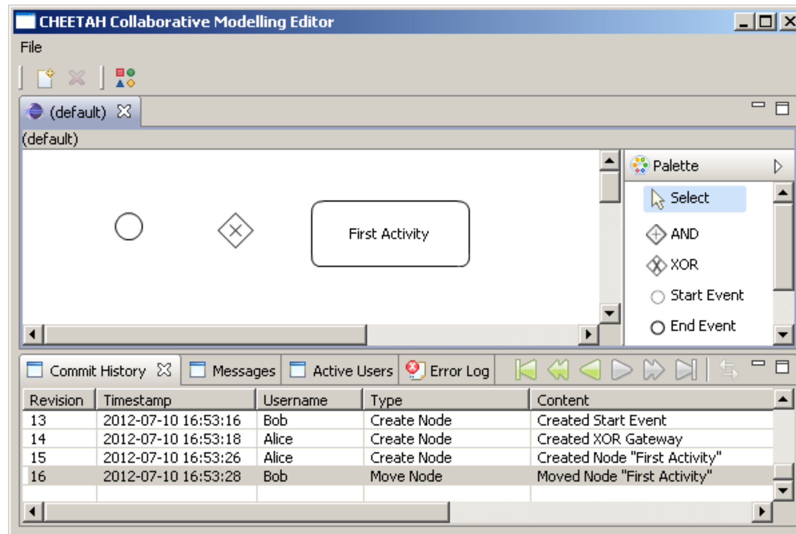

Figure 3.4: GUI overview during a modelling session

Section 3.4.6), messages and presence notifications (see Section 3.4.7) and a list of all currently active users (see Section 3.4.12).

3.4.5 User Colors

Our first prototype implementations provided the user with awareness information in the form of the commit history view (see Section 3.4.6), the messages view (see Section 3.4.7) and the active users view (see Section 3.4.12).

While testing these prototypes, we learned that users need more awareness information in order to effectively work together collaboratively on the same model. As a consequence, our implementation now provides the user with additional awareness information in the form of *user colors*.

Initially we added some code allowing each user to select a color of his choice, which will then be used to display this user's messages, changes in the commit history view and the username itself in the active users view. In order for this

information to also be available while analyzing a finalized model, we decided to also store color changes in the repository (see Section 3.3.2).

As additional awareness features like selection highlighting (see Section 3.4.8), new node feedback (see Section 3.4.10) and node tooltips (see Section 3.4.11) were added to our implementation, these features were also augmented with the usage of user colors, in order to provide the users with more awareness data and ease communication and coordination among the group.

3.4.6 Commit History View

Probably the most important view in our implementation is the *Commit History View*. This view shows columns for the revision-ID, the username and timestamp, the change type and a detailed description in the user's color (see Figure 3.5). Each and every change made to the model, along with messages, presence notifications and color changes is displayed in this view.

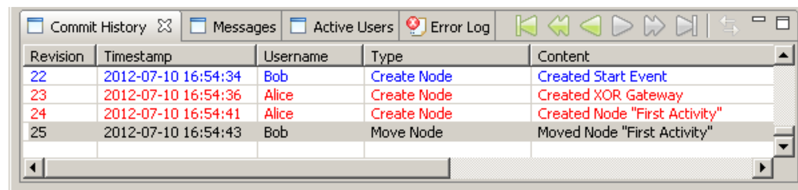

Figure 3.5: The commit history view

This enables every user to review the creation process—and also the discussions which took place along it—step by step. The commit history also allows to replay all the changes made by each user. We can even go back and forth in time and move to any state the process model had during it's creation, by using the toolbar buttons (see Figure 3.6).

There are buttons to go back and forward for one or five entries respectively, along with buttons to jump to the beginning of the session or the current state of the model. By clicking on a specific entry in the commit history view, we will

Figure 3.6: The commit history view toolbar

directly jump to the state the model had at the time this revision was created. The last button in the view's toolbar can be used to link messages to model elements (see Section 3.4.9).

Going back in time is not just limited to the analysis of a finalized model, but can also be done during modeling at any time by each participating user. This also enables the users to revise the creation history and take a look at the state a model had at some point in the past.

Whenever a user goes back in time, the editor itself does not permit any modifications. The model is then in a read-only state for viewing purposes. This prevents conflicts and branching repositories. The only exception to this is linking messages, which can also be done retroactively (see Section 3.4.9).

Once a user leaves a session and logs back in later on, he will not be shown the model in it's current state, but will find it just like it was when he last left. This will then enable him to quickly review what changed during his absence, utilizing the commit history view.

The commit history view is the key element to satisfy the functional requirement of awareness (see Section 2.2.1). It is also helpful for effective communication (see Section 2.2.2) and coordination (see Section 2.2.3). When analyzing the process of process modeling, this view is the central element to consider, as it provides the user with most of the information regarding model changes.

3.4.7 Messages View

To satisfy the fundamental functional requirement of communication (see Section 2.2.2), we provide the users with a text-based chat view (see Figure 3.7)

which they can use to communicate with each other to discuss the model and clarify problematic situations.

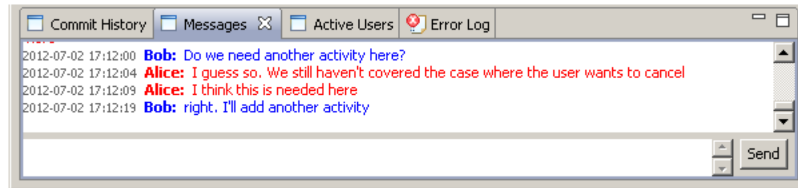

Figure 3.7: The messages view

Especially when modelling asynchronously and in distributed environments, the users have to coordinate their efforts (see Section 2.2.3) in order to work towards a common goal (see Section 2.2.4). By highlighting the messages from each user in their color (see Section 3.4.5) we make it easier to distinguish between messages from different participates.

Every time a user logs in or leaves the current session, a special message is created by the server and sent to all active clients, which will be shown in the messages view (see Figure 3.8). Together with a dedicated view (see Section 3.4.12), this helps making the user aware which other users are connected to the current session.

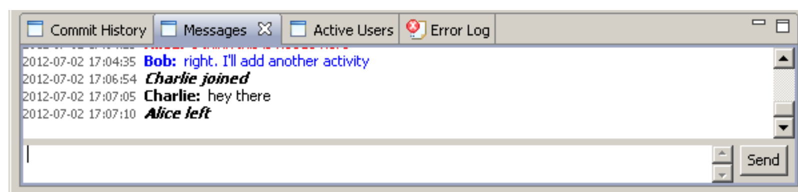

Figure 3.8: Messages view showing presence notifications

3.4.8 Selection Highlighting

As already discussed in Section 3.4.5, some awareness features have been added to our implementation after the first test sessions with early prototypes. During those tests we recognized that it would be of advantage for a user to know what the other users are currently doing.

We discussed various options to gather this information and to present it to the users. Eventually we decided, that a good way to give the other users feedback of one's actions would be to show them what we currently have selected in our editor window. All the selected elements are highlighted in the selecting user's color (see Figure 3.9). This enables everyone to instantly see, if someone is working on a particular model element.

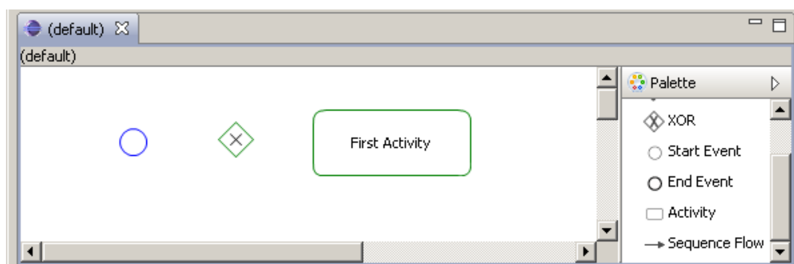

Figure 3.9: Elements highlighted according to the selecting user's color

Indicating whether another user just selected a particular element in the editor can help reducing conflicts (see Section 3.4.13) by reducing the number of concurrent modifications on the same elements. Additionally this improves communication (see Section 2.2.2) and coordination (see Section 2.2.3), since it makes it more clear to which elements the current discussions are referring to.

Our server keeps track of all the user's selections. It knows which clients selected which model elements, and thereby can determine which model elements are selected by some client and which are not selected at all. Note that selection information is volatile—it is not stored in the repository (i.e. there is no new revision created when a user changes his selection), since it is of no value for users

which join the modelling session later on. Instead, the information is kept in the server's memory only, and will be discarded when the server is shut down.

Whenever multiple users have the same node selected, the node will show in the color of the user which selected it first. Every time a user changes the selection in his editor, the client will notify the server about this change. The server will then determine which nodes' highlightings need to be changed and broadcast this information to all active clients. Once an element is not selected by any user anymore, the server will tell all clients to remove the highlighting from that element.

Active clients will remove any highlightings from their elements in the editor in case they are not following the model's progress anymore (i.e. they stepped back in time to observe the state of the model at some arbitrary point in the past using the commit history view—see Section 3.4.6), and will ignore highlighting related updates from the server.

Once they again actively follow the current state of the model, they will need to get an update on all the current selections. To achieve this, clients can request a full selection update from the server. After receiving this information and applying the highlightings, they can continue to process highlighting data received from the server and will again show the current state of the model.

3.4.9 Linked Messages

By looking at the message view (see Section 3.4.7) of a modeling session, it may be hard to figure out which elements the users were referring to when they wrote their messages. Sometimes elements may have been renamed afterwards and will therefore not be easy to identify; Sometimes there may be two or more elements with the same name. Even worse, when conversations concerned a group of elements this may be still more complicated.

In order to effectively analyze a finalized model and to observe what discussions took place around a specific element, the messages the users sent during modelling have to be associated with the elements in the model. To accomplish this, each

message in the repository stores a set of associated elements. This set may also be empty in case the message is not linked to any specific element. However, when a user has one or more elements selected while he sends a message, this specific message is linked to the selected elements.

The link between a message and nodes can also be established after the message has been submitted. To do so, one can go back to the message which should be linked in the commit history view (see Section 3.4.6), select one ore more nodes which should be linked, and then create the actual link by using a button in the view's toolbar.

Linking nodes to a message, which has already been submitted, will cause the server to actually change the message object in the repository. Together with conflicting changes (see Section 3.4.13) this is the only situation where some revision in the repository will be changed after it has been committed—normally the server just adds new elements to the list of revisions. This change in the repository is done to give new clients the message-element relationship right away when they connect to the repository. Linking messages also does not influence the model in any way and thus can be done safely without the risk of inconsistencies. The server will also instruct all the currently active clients to change their local copy of the repository, in order to reflect the new message-element relationship.

Linked messages are also inserted into the node history of the element's tooltips, to which they are linked to (see Section 3.4.11). Additionally, when selecting a message which is linked to some elements in the commit history, these elements will be selected in the model editor. If the user changes the selection (or even deselects everything) and executes the linking action again, the message will be re-linked in the server's repository and all the clients.

Linking messages to elements creates a relationship between them. This relationships can be used to observe the discussions and arguments when analyzing a finalized model, and thereby give a more detailed view of the overall modeling session.

Figure 3.10 shows an example of this process. *Client 2* sent a new message, which was distributed to all clients by the server with revision-ID *[x]*. *User 1* then decides, that this message is related to element A, and links the message to this element by telling the server to do so. The server then changes it's repository, and tells all clients to do likewise.

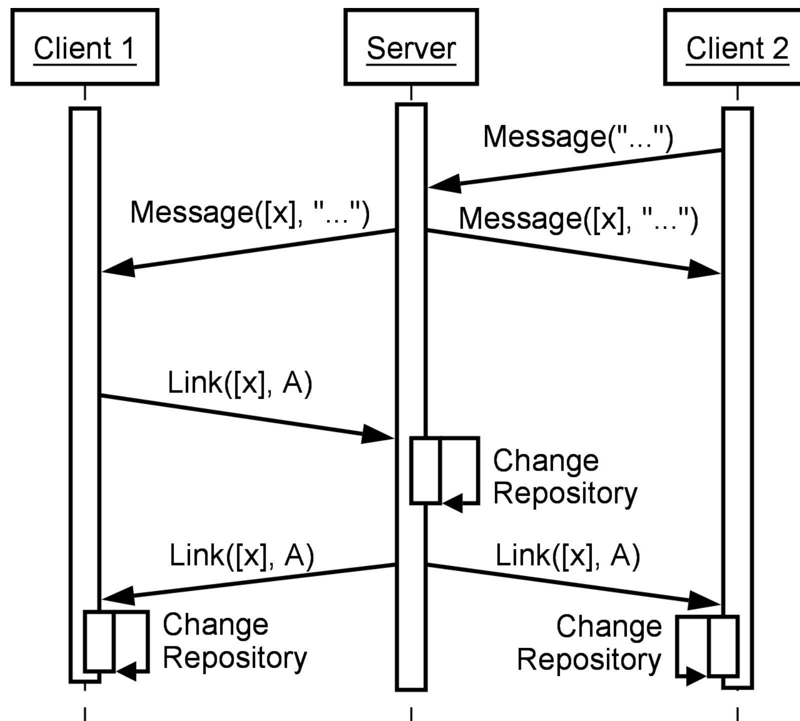

Figure 3.10: Example for linking a message after it has been sent

3.4.10 New Node Feedback

During testing sessions of early prototypes, we often ended up in a situation where we were discussing the model by exchanging messages. Once we decided that some elements were still missing, it often happened, that multiple users created a new node for this missing element.

Eventually, we had to agree on which of these elements we wanted to keep and delete all the redundant ones. This process just causes unnecessary clutter in the commit history, as multiple elements get created needlessly, another unrelated discussion takes place, and superfluous elements getting deleted again.

The result are a lot of changes in the history, which just resulted in one single new element. These artifacts in the revision history make the analysis of the final model more cumbersome and tedious, so we finally decided that we had to mitigate this problem somehow by helping the users to better coordinate their work (see Section 2.2.3).

Clients now notify the server when the user creates a new modeling element. This is done exclusively for new elements which take some time to create (i.e. where the user has to enter the name of an activity), but not for elements which are created immediately (e.g. edges).

Once the server learns of a new node creation, it will tell all active clients about this creation. The clients can then show some feedback to the user (see Figure 3.11) to make him aware of the creation of a new model element, which is taking place somewhere else.

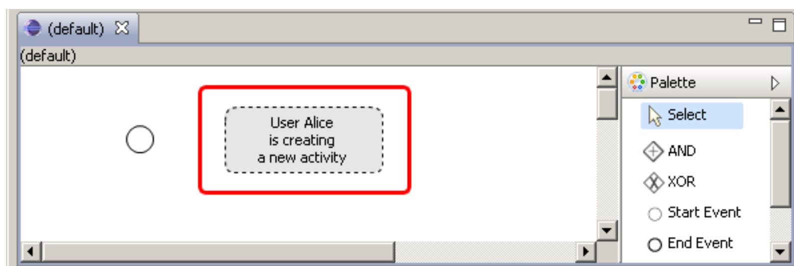

Figure 3.11: Visual feedback on node creation

If the originating user decides to cancel the element creation, then the same process is repeated; The client notifies the server about the cancellation, which then tells all active clients to remove the feedback element. Completed element creations will not trigger a notification of all clients; Instead, each client will know

upon receipt of a new creation command, that any active feedback elements for the same user can now be removed, since he obviously decided to finally create the element.

3.4.11 Node Tooltips

Especially when analyzing a finalized model, but also during the modeling session itself, it may be helpful to take a look at an element's complete history. The user may want to see who created it, who moved it to the location it is now and who else modified it in any way. Utilizing the commit history view (see Section 3.4.6) a user can see any modification to any specific element, but searching for the entire change history of a particular element can be very hard and time consuming.

We therefore decided to enrich the tooltips of elements with their complete history. From their creation on, each change to an element will also create a new entry in it's *node history*. The entry shows the date and time of each and every change, which user performed it and what has been done with the element (see Figure 3.12). Note that linked messages (see Section 3.4.9) will also be shown in the node history.

With this feature at hand it is now convenient to review the history of any element in the model. Since the entries in the node history are highlighted in the users' color, it is also easy to see who performed a particular change (see Section 3.4.5).

This feature is especially helpful when analyzing the final model. A quick glance at the node history will show immediately all the element's changes and related messages in a central place, which should already give a good overview on what happened with this element during the modeling session.

Simply by taking a look at the tooltips of various model elements one can immediately see whether these elements have been discussed or changed a lot or not. Elements with short node histories obviously did not create a lot of confusion or cause problems among the modelers, while elements with long

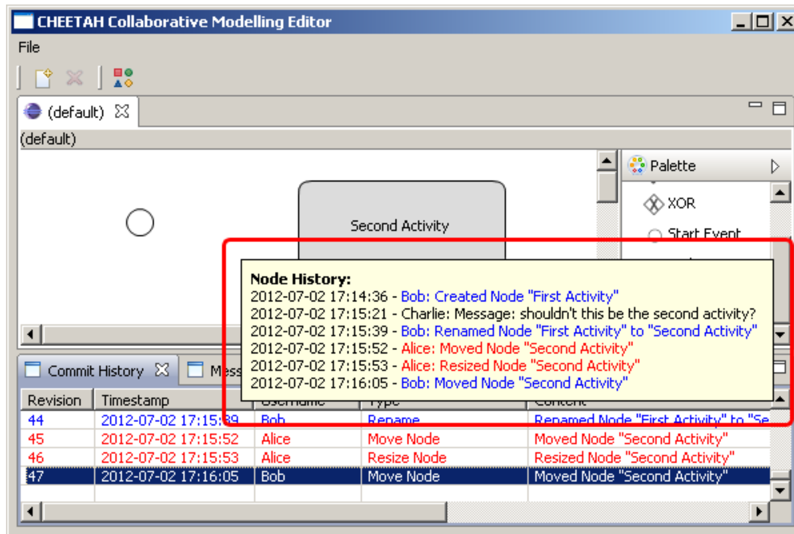

Figure 3.12: The tooltip for a node which has been modified multiple times

histories may indicate imprecise specifications or disagreements between users during the modeling session.

3.4.12 Active Users View

Our implementation provides the user with a dedicated view, showing all the currently online users (see Figure 3.13).

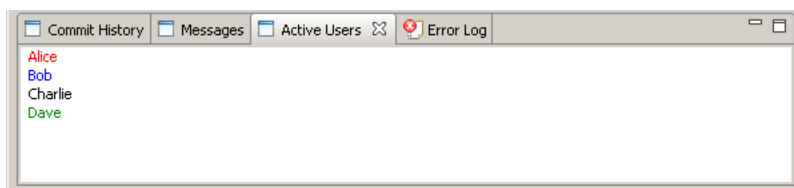

Figure 3.13: The active users view

The user's names are also displayed in the color they chose, which gives additional awareness information (see Section 2.2.1). Whenever a new user joins the modelling session, he can use this view to immediately see what other group members are currently online without having to look through all the presence notifications in the messages view (see Section 3.4.7).

Since join- and leave-events are also stored in the repository (see Section 3.3.2), even during analysis of a final model the users displayed in the *active users view* will reflect which users were present at any time during the creation of a model. This may yield additional information for analysis (e.g. whether all needed domain experts were present at the time important decisions were made).

3.4.13 Conflict Management

In our distributed setting where multiple users can work on a model simultaneously, conflicting changes may occur. When for example one user renames an element, while some other user deletes the same element—depending on the order of these actions—these may be legitimate changes or result in a conflict, because a non-existent element should be given a different name.

The obvious place to handle these kinds of problems is the server as the central instance. Since the server has to serialize all requests from clients, the final order of all commands will be determined by the server. A central server could check each change for validity before actually executing it and broadcasting the change to all active clients. This would however imply, that the server would need to know more details about the objects it manages than initially necessary. Since our server was intentionally designed to be generic (see Section 3.4.3), it cannot check commands for conflicts.

To stay with the example described above, both clients could perform the change within a timeframe of a few milliseconds (something between zero and maybe a few hundred—depending on processing- and network speed), and for both clients the change they made was absolutely legitimate at the time they sent it to the server (i.e. the element still existed when the user renamed it).

When both changes happen to get the wrong order on the server, this will result in a conflict when the commands are then finally applied to the client's models, because the rename operation will try to rename an element, which has been deleted by the preceding command.

These conflicts are quite rare. Of all the revision types (see Section 3.3.2), just *CHANGE* revisions can actually cause a conflict. The other types of revisions like messages and presence notifications will never cause a conflict to occur. Even more—not even all *CHANGE* revisions (see Section 3.2.1) can cause conflicts (e.g. two concurrent *MoveNode* operations will yield the same result, no matter in what order they are performed).

Still, the smaller the model and the more users actively work on it, the higher the probability for conflicts. During all our testing sessions we never managed to run into such a conflict. To test our implementation's conflict management we had to artificially create them using program logic, which executed conflicting commands in two clients and sent them to the server at the same time.

Nevertheless, conflicts can happen and have to treated accordingly. Whenever a command received from the server causes a conflicting change, an exception will be raised in the client. This way each client can determine if a command was executed successfully or not. So in order to recognize conflicts, our client always monitors commands coming from the server for exceptions during their execution.

Once the client learns of a command causing a conflict, it will mark the associated revision as conflicting in it's repository. This is also indicated in the commit history view (see Figure 3.14).

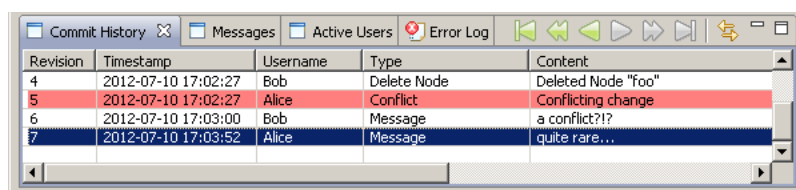

Figure 3.14: A conflicting change highlighted in the commit history view

Since all clients have their models synchronized at all times, and all of them execute the same changes in the same order, each client will recognize the conflict and mark the revision accordingly. The conflicting change will be removed from the repository and replaced by a special revision object indicating that this revision caused a conflict (i.e. a revision of type *CONFLICT*—see Section 3.3.2).

Additionally, the client who initially committed the conflicting revision will display a message to the user informing him, that his last change caused a conflict and was therefore undone. The user can then review the changes made by others and act accordingly. The client which caused the conflict also has the responsibility to tell the server about this conflict.

Once the server learns about a conflicting revision, it will also mark this revision as conflicting in the repository, in order to give new clients a consistent and conflict-free view of the model. Together with linked messages (see Section 3.4.9), this is the only situation were already committed revision get modified in the repository. Here it is safe to do so, since the removed revision was conflicting in the first place.

Figure 3.15 depicts an example sequence of a conflicting change. *Client 1* wants to rename element A, while at the same time *Client 2* wants to delete element A. The *Server* happens to send out the delete command (revision-ID *[x]*) before the rename command (revision-ID *[y]*).

Both clients receive the delete command first and execute it, so when they finally want to execute the following rename command, the element to rename no longer exists. They will catch an exception and therefore know that this command caused a conflict.

Each client now modifies it's repository and replaces the conflicting revision *[y]* with a *CONFLICT* revision. *Client 1*, which initially sent the command causing the conflict to the server will display a message to the user informing him, that his change has been rejected. Additionally it will also inform the server about the conflict in revision *[y]*, since the server cannot recognize conflicts for itself.

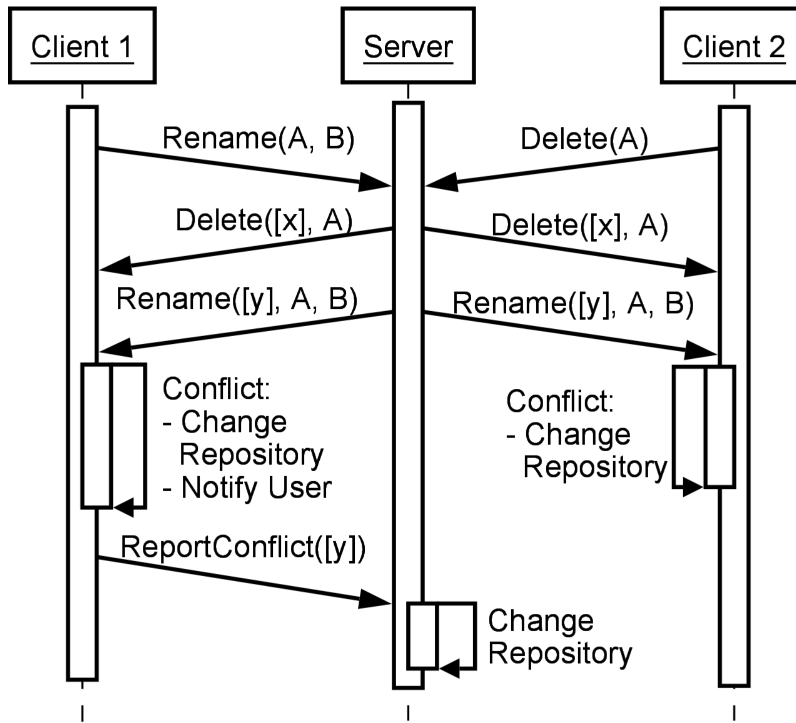

Figure 3.15: Sequence chart of a conflicting change

The server will then also modify it's own repository and mark the revision as conflicting.

3.4.14 Feature Summary

Most of the features described in the previous section aim to satisfy one or more of the functional requirements introduced earlier (see Section 2.2). Table 3.4 summarizes the relationship between the features we implemented and the functional requirements they satisfy.

Feature		Functional requirement				
		Awareness	Communication	Coordination	Group decision making	Team-building
Feature	User colors	x	x			
	Commit history view	x	x	x	x	x
	Messages view	x	x	x	x	x
	Selection highlighting	x		x		
	Linked messages	x	x			
	New node feedback	x		x		
	Node tooltips	x		x		
	Active users view	x	x	x	x	x

Table 3.4: Relationship between implemented features and functional requirements

Chapter 4

Evaluation

Our implementation should be easy to use and it's handling be intuitive for the users. We wanted to evaluate whether our implementation can support collaborative process modeling and provide a group with the required features to effectively work together. In order to assess the ease of use of our implementation and how well it can help a team on modeling a process collaboratively, we performed a small evaluation.

4.1 Method

Groups participating in our evaluation were given the task to collaboratively create a small business process model from a textual description using our implementation.

Before actually using our implementation, the users had to answer a set of demographic questions. These questions covered topics like their current work status, their experience with business process modeling in general, how many process models the participants analyzed and created during the last year and their familiarity with the process they should model.

Following this, the participants used our implementation to collaboratively create a process model in pairs. They did so synchronously (i.e. they were working on the model at the same time) and in a distributed setting.

Once finished with the modeling task, the users were given a set of questions regarding our implementation. The first set of questions covered how well our implementation supports the users by satisfying the different functional requirements (i.e. *awareness*, *communication*, *coordination*, *group decision making* and *team-building*).

Following this, we focused on evaluating the ease of use and the perceived usefulness of our *messages view* (see Section 3.4.7), *commit history view* (see Section 3.4.6), *node tooltips* (see Section 3.4.11), and *user colors* (see Section 3.4.5).

In order to accomplish this, we applied the *Technology Acceptance Model* (TAM) [Dav86] to evaluate our implementation. The TAM introduces the two theoretical constructs *perceived usefulness* (i.e. how much will a given tool ease a user's job) and *perceived ease of use* (i.e. how easy is it to use a given tool) to describe how likely a user is to actually use a given application [Dav89].

For each question, the participants had to choose one of seven possible answers ranging from 1 (*"strongly agree"*) to 7 (*"strongly disagree"*) corresponding with a 7-point Likert scale [Lik32].

Finally, we asked participating users to give feedback and suggestions for improvements regarding our implementation.

4.2 Results

This section covers the questions we asked our participants, accompanied by the results we obtained. A total of 10 users participated in our evaluation.

The results for our general questions regarding the functional requirements are discusses in Section 4.2.1, while Section 4.2.2 shows the extent to which the participants used the provided collaboration features. Several features were not used by some of our participants. In these cases, the gathered data was excluded from our evaluation. The answers of our participants to questions regarding specific collaboration features are shown in Sections 4.2.3, 4.2.4 and 4.2.5.

Finally, Section 4.2.6 discusses the feedback and suggestions we obtained from our evaluation.

4.2.1 General Questions

Table 4.1 shows the participant's answers to questions covering general aspects of collaborative modeling using our implementation, along with a calculated average value. Every participant except for one answered this block of questions.

The questions in this block cover the different levels of social interaction (see Section 2.2)—*awareness*, *communication*, *coordination*, *group decision making* and *team-building*. The average results we obtained are between *"quite agree"* and *"slightly agree"*.

We can therefore conclude that our implementation seems to satisfy the functional requirements of the various levels of social interaction reasonably well.

n = 9	strongly agree	quite agree	slightly agree	neither	slightly disagree	quite disagree	strongly disagree	average value
	1	2	3	4	5	6	7	
At all times I have been able to see on which parts the other participants were working.	1	2	4	1	0	1	0	3.00
It was easy to communicate with the other participants.	4	3	0	1	1	0	0	2.11
It was easy to exchange knowledge with the other participants.	2	4	1	0	2	0	0	2.56
It was possible to coordinate the modeling process and distribute the tasks over the participants.	2	4	2	0	1	0	0	2.33
It was easy to find a consensus about the model with the other participants.	2	6	0	1	0	0	0	2.00

Table 4.1: Evaluation results regarding general questions

4.2.2 Feature Usage

In total, 10 users participated in our evaluation. Table 4.2 shows how many of our participants used the features we wanted to evaluate.

Feature	Usage
Messages View	10
Node Tooltips	3
User Colors	4
Commit History View	0

Table 4.2: Usage of collaboration features by evaluation participants

The messages view was used by all the participants. This emphasizes the need for communication to coordinate multiple users. More advanced features, like node tooltips and user colors were used only by a small subset of our participants, while no one used the commit history view of our implementation. We think, this may result from the relative simplicity of the tasks given to the participants, which probably were not complex enough to warrant the use of these features.

Our more advanced collaboration features are also designed with a focus on the analysis of created models and further research on the collaborative process of process modeling itself, rather than just easing the task to model collaboratively. Additionally, these advanced features are more helpful in an asynchronous setting (i.e. with participants working on the model not at the same time). Working on a model synchronously already eases the task of following the model's progress and keeping track of the changes someone else performed.

4.2.3 Messages View

Every participant used our messages view. Table 4.3 shows the results of our evaluation. The results indicate that the messages view seems to be perceived as quite useful and quite easy to use.

n = 10	strongly agree	quite agree	slightly agree	neither	slightly disagree	quite disagree	strongly disagree	average value
	1	2	3	4	5	6	7	
I used the chat window extensively.	7	2	0	1	0	0	0	1.50
Using the introduced chat window would enable me to accomplish modeling tasks more quickly.	0	3	5	2	0	0	0	2.90
Using the introduced chat window would improve my modeling performance.	1	1	5	3	0	0	0	3.00
Using the introduced chat window for modeling would increase my productivity.	0	3	5	2	0	0	0	2.90
Using the introduced chat window would enhance my effectiveness on modeling.	0	1	5	3	0	1	0	3.50
Using the introduced chat window would make it easier to create a model.	2	3	4	1	0	0	0	2.40
I would find the introduced chat window useful for modeling.	3	5	1	0	1	0	0	2.10
Learning to operate the introduced chat window is easy for me.	8	1	1	0	0	0	0	1.30
I find it easy to get the introduced chat window to do what I want it to.	7	2	1	0	0	0	0	1.40
My interaction with the introduced chat window is clear and understandable.	7	2	1	0	0	0	0	1.40
I find the introduced chat window to be flexible to interact with.	3	1	4	1	1	0	0	2.60
It would be easy for me to become skillful at using the introduced chat window.	6	1	2	1	0	0	0	1.80
The introduced chat window is easy to use.	7	1	1	1	0	0	0	1.60

Table 4.3: Evaluation results for the usage of the messages view

4.2.4 Node Tooltips

Just three participants used our node tooltips. Table 4.4 shows the results of our evaluation. The few participants which used this feature were neutral about it's usefulness, but still considered it to be easy to use.

n = 3	strongly agree	quite agree	slightly agree	neither	slightly disagree	quite disagree	strongly disagree	average value
	1	2	3	4	5	6	7	
I used the node's tooltip extensively.	1	0	0	1	0	1	0	1.67
Using the introduced node's tooltip would enable me to accomplish modeling tasks more quickly.	0	1	0	1	1	0	0	3.67
Using the introduced node's tooltip would improve my modeling performance.	0	1	0	1	1	0	0	3.67
Using the introduced node's tooltip for modeling would increase my productivity.	0	1	0	1	0	1	0	4.00
Using the introduced node's tooltip would enhance my effectiveness on modeling.	0	1	0	1	1	0	0	3.67
Using the introduced node's tooltip would make it easier to create a model.	0	1	0	2	0	0	0	3.33
I would find the introduced node's tooltip useful for modeling.	0	1	1	1	0	0	0	3.00
Learning to operate the introduced node's tooltip is easy for me.	0	3	0	0	0	0	0	2.00
I find it easy to get the introduced node's tooltip to do what I want it to.	1	1	0	1	0	0	0	2.33
My interaction with the introduced node's tooltip is clear and understandable.	1	2	0	0	0	0	0	1.67
I find the introduced node's tooltip to be flexible to interact with.	1	1	0	1	0	0	0	2.33
It would be easy for me to become skillful at using the introduced node's tooltip.	0	2	0	1	0	0	0	2.67
The introduced node's tooltip is easy to use.	2	1	0	0	0	0	0	1.33

Table 4.4: Evaluation results for the usage of node tooltips

4.2.5 User Colors

Four participants used the option of user colors. Table 4.5 shows the results of our evaluation. The few participants which used this feature, perceived it to be quite useful and easy to use.

n = 4	strongly agree 1	quite agree 2	slightly agree 3	neither 4	slightly disagree 5	quite disagree 6	strongly disagree 7	average value
I used the coloring option extensively.	1	2	0	0	0	1	0	2.75
Using the introduced coloring option would enable me to accomplish modeling tasks more quickly.	3	1	0	0	0	0	0	1.25
Using the introduced coloring option would improve my modeling performance.	2	2	0	0	0	0	0	1.50
Using the introduced coloring option for modeling would increase my productivity.	3	1	0	0	0	0	0	1.25
Using the introduced coloring option would enhance my effectiveness on modeling.	3	1	0	0	0	0	0	1.25
Using the introduced coloring option would make it easier to create a model.	2	2	0	0	0	0	0	1.50
I would find the introduced coloring option useful for modeling.	3	1	0	0	0	0	0	1.25
Learning to operate the introduced coloring option is easy for me.	1	3	0	0	0	0	0	1.75
I find it easy to get the introduced coloring option to do what I want it to.	0	3	1	0	0	0	0	2.25
My interaction with the introduced coloring option is clear and understandable.	1	3	0	0	0	0	0	1.75
I find the introduced coloring option to be flexible to interact with.	0	4	0	0	0	0	0	2.00
It would be easy for me to become skillful at using the introduced coloring option.	1	3	0	0	0	0	0	1.75
The introduced coloring option is easy to use.	1	3	0	0	0	0	0	1.75

Table 4.5: Evaluation results for the usage of user colors

4.2.6 User Feedback and Suggestions

Some users claimed, that they were not used to work collaboratively on a model. They already created process models in a single-user setting, but never collaboratively, and thus did not use the advanced collaboration features to their fullest extent.

One user voiced the wish to be able to divide the model under construction into a hierarchy of sub-models, which can then be worked on independently of the rest of the model. These sub-parts should then later on be combined to an overall model. This would reduce conflicts and ease work distribution and assignment among group members.

Another user wanted to have the feedback on node creation (see Section 3.4.10) extended to some more long-lasting operations, like the renaming of elements or the creation and modification of edge labels.

Finally, some users wished for more awareness during communication (e.g. display when another user is currently typing a new message) or the use of graphical smileys in the messages view (see Section 3.4.7). One user criticized the relative slowness of typing messages in the chat window and would prefer some faster means of communication.

Chapter 5

Summary

The extensions and additions to Cheetah Experimental Platform developed for this thesis enable the collaborative modeling of business processes. Following the spirit of CEP itself, every change during the modeling phase—along with communication and presence information—is logged for later use. This enables a detailed analysis of the *process of collaborative process modeling*.

5.1 Conclusion

Our implementation enables a group of users to collaboratively create a process model—regardless if they are co-located or distributed, working at the same time or asynchronously.

The user interface takes advantage of the intuitive modeling editor of Cheetah Experimental Platform, and extends it to communicate with a central server, which distributes changes performed by one client to all the other currently connected clients.

A lot of effort has been put into providing the users with as much awareness data as possible in order to facilitate effective modeling. We also built in a simple, text-based chat, which enables communication and coordination between the group members. All these features also foster group decision making as well as

team-building, and thereby satisfy the fundamental functional requirements for collaborative software systems.

5.2 Outlook

This thesis just provides a basis for more research on the process of collaborative process modeling. The gathered data during modeling sessions is stored in an XML file, and can be analyzed in various ways using different metrics.

The results of such detailed analysis may lead to improved modeling tools, better modeling techniques and an overall better understanding of collaborative process modeling.

From the results of our evaluation we conclude, that users do not seem to be used to create a model collaboratively and thus do not take advantage of all the collaboration features our implementation provides. This of course may be different for larger process models or in cases where the users do not work on the model at the same time. However, the feedback and suggestions gathered from our participants clearly show room for further improvements and additional features, which may ease the task of collaborative modeling in the future.

Future work on our implementation therefor may include more detailed evaluations of our advanced collaboration features, possibly using a different setting like asynchronous modeling. The results of these evaluations can then be used to adapt and extend our current implementation in order to provide the users with more assistance in the collaborative creation of process models.

Another feature worth considering would be the introduction of audio-based communication, combined with the ability to record the resulting audio-streams for later analysis. This would pose less distractions and ease the communication effort—especially when modeling synchronously.

A different additional feature to implement would be hierarchical modeling support. This would enable the users to divide a model into sub-parts and edit them independently. A feature like this could be useful especially for larger

models, since it would be much easier to get a good overview of the model, while still being able to look at all the details when necessary. Furthermore this would ease the task of work distribution among team members and reduce the potential for conflicting changes.

List of Figures

3.1 Architecture Overview 21
3.2 Client-Server Architecture 27
3.3 Blocking Polling Sequence Chart 30
3.4 Graphical User Interface Overview 35
3.5 Commit History View 36
3.6 Commit History View Toolbar 37
3.7 Messages View 38
3.8 Presence Notifications 38
3.9 Selection Highlighting 39
3.10 Message Linking Sequence Chart 42
3.11 Feedback on Node Creation 43
3.12 Node Tooltip 45
3.13 Active Users View 45
3.14 Conflict Highlighting in the Graphical User Interface 47
3.15 Conflict Management Sequence Chart 49

List of Tables

2.1 Classification Scheme for Colaborative Software 8

3.1 List of Cheetah Experimental Platform User Interactions 22
3.2 Properties of Revision Objects 24
3.3 List of Revision Types 25
3.4 Feature Summary 50

4.1 Evaluation Results: General Questions 53
4.2 Evaluation Results: Feature Usage 54
4.3 Evaluation Results: Messages View 55
4.4 Evaluation Results: Node Tooltips 56
4.5 Evaluation Results: User Colors 57

Bibliography

[Blo08] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[BM02] Georgia Bafoutsou and Gregoris Mentzas. Review and functional classification of collaborative systems. *International Journal of Information Management*, 22(4):281–305, 2002.

[CVP+12] Jan Claes, Irene Vanderfeesten, J. Pinggera, H.A. Reijers, B. Weber, and G. Poels. Visualizing the Process of Process Modeling with PPMCharts. In *Proc. TAProViz '12*, 2012.

[CVR+12] Jan Claes, Irene Vanderfeesten, H.A. Reijers, J. Pinggera, M. Weidlich, S. Zugal, D. Fahland, B. Weber, J. Mendling, and G. Poels. Tying Process Model Quality to the Modeling Process: The Impact of Structuring, Movement, and Speed. In *Proc. BPM '12*, pages 33–48, 2012.

[Dav86] F. D. Davis. *A technology acceptance model for empirically testing new end-user information systems: theory and results*. Doctoral dissertation, MIT, 1986.

[Dav89] F. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q.*, 13(3):319–340, September 1989.

[DBT04] Dov Dori, Dizza Beimel, and Eran Toch. OPCATeam - Collaborative Business Process Modeling with OPM. In *Business Process Management*, pages 66–81, 2004.

[DC99] Dov Dori and Edward F. Crawley. *Object-Process Methodology: A Holistic Systems Paradigm.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[DC11] Manoj Das and Linus Chow. Next generation bpm suites: Social and collaborative. In *Social BPM - Work, Planning and Collaboration Under the Impact of Social Technology.* Workflow Management Coalition, 2011.

[dH05] Marielle den Hengst. Collaborative Modeling of Processes: What Facilitation Support Does a Group Need? In *AMCIS 2005 Proceedings*, 2005.

[DOV00] Douglas Dean, Richard Orwig, and Douglas Vogel. Facilitation Methods for Collaborative Modeling Tools. *Group Decision and Negotiation*, 9:109–128, 2000. 10.1023/A:1008702604327.

[FAW+10] Matthias Farwick, Berthold Agreiter, Jules White, Simon Forster, Norbert Lanzanasto, and Ruth Breu. A Web-Based Collaborative Metamodeling Environment with Secure Remote Model Access. In *ICWE*, pages 278–291, 2010.

[FvdW06] P.J.M. Frederiks and Th.P. van der Weide. Information modeling: The process and the required competencies of its participants. *Data & Knowledge Engineering*, 58(1):4–20, 2006.

[GF08] P Perez Gonzalez and JM Framinan. Tools for Collaborative Business Process Modeling. In *Encyclopedia of Networked and Virtual Organizations*, pages 1643–1652. Goran D. Putnik and M. M. Cruz-Cunha, 2008.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, November 1994.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004.

[HPvdW05] S.J.B.A. Hoppenbrouwers, H.A.(Erik) Proper, and Th.P. van der Weide. A fundamental view on the process of conceptual modeling. In *Conceptual Modeling – ER 2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 128–143. Springer Berlin Heidelberg, 2005.

[IRRG09] Marta Indulska, Jan Recker, Michael Rosemann, and Peter Green. Business process modeling: Current issues and future challenges. In *Advanced Information Systems Engineering*, volume 5565 of *Lecture Notes in Computer Science*, pages 501–514. Springer Berlin / Heidelberg, 2009.

[LDV97] James D. Lee, Douglas L. Dean, and Douglas R. Vogel. Tools and methods for group data modeling: a key enabler of enterprise modeling. *SIGGROUP Bull.*, 18:59–63, August 1997.

[Lik32] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 1932.

[MRW12] Jan Mendling, Jan C. Recker, and Johannes Wolf. Collaboration features in current bpm tools. *EMISA Forum*, 32(1):48–65, January 2012.

[PFM+13] J. Pinggera, M. Furtner, M. Martini, P. Sachse, K. Reiter, S. Zugal, and B. Weber. Investigating the Process of Process Modeling with Eye Movement Analysis. In *Proc. ER-BPM '12*, pages 438–450, 2013.

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[PSZ+12] J. Pinggera, P. Soffer, S. Zugal, B. Weber, M. Weidlich, D. Fahland, H.A. Reijers, and J. Mendling. Modeling Styles in Business Process Modeling. In *Proc. BPMDS '12*, pages 151–166, 2012.

[PZW10] J. Pinggera, S. Zugal, and B. Weber. Investigating the Process of Process Modeling with Cheetah Experimental Platform. In *Proc. ER-POIS '10*, pages 13–18, 2010.

[PZW+12] J. Pinggera, S. Zugal, M. Weidlich, D. Fahland, B. Weber, J. Mendling, and H. Reijers. Tracing the Process of Process Modeling with Modeling Phase Diagrams. In *Proc. ER-BPM '11*, pages 370–382, 2012.

[Rit08] Peter Rittgen. COMA: A Tool for Collaborative Modeling. In *CAiSE Forum*, pages 61–64, 2008.

[Rit09] Peter Rittgen. Collaborative Modeling - A Design Science Approach. In *HICSS*, pages 1–10, 2009.

[Rit10] Peter Rittgen. Collaborative Business Process Modeling - Tool Support for Solving Typical Problems. In *Proceedings of the Conf-IRM 2010 "Collaboration and Community in a Global World"*, 2010.

[SEA+02] York Sure, Michael Erdmann, J́'urgen Angele, Steffen Staab, Rudi Studer, and Dirk Wenke. OntoEdit: Collaborative Ontology Development for the Semantic Web. In *International Semantic Web Conference*, pages 221–235, 2002.

[SJP99] Charles Steinfield, Chyng-Yang Jang, and Ben Pfaff. Supporting virtual team collaboration: the TeamSCOPE system. In *GROUP*, pages 81–90, 1999.

[TSS09] Christian Thum, Michael Schwind, and Martin Schader. SLIM - A Lightweight Environment for Synchronous Collaborative Modeling. In *MoDELS*, pages 137–151, 2009.

[WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, July 2000.