**Ulm University** | 89069 Ulm | Germany

**Faculty of Engineering, Computer Science and Psychology**
Institute of Databases and Information Systems

# Design and Implementation of a Dynamic Web-based User Interface for the proCollab System Supporting Knowledge-intensive Business Processes

Master's Thesis at Ulm University

**Submitted by:**
Matthias Gerber
matthias.gerber@uni-ulm.de

**Reviewer:**
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

**Supervisor:**
Nicolas Mundbrod

2017

Version October 10, 2017

Satz: PDF-LaTeX $2_\varepsilon$

# Abstract

Globalization and the change to post-industrial societies have led to an increased value of knowledge-intensive processes. Areas creating and utilizing new knowledge, such as research and development are of high importance for today's companies. In these areas, knowledge workers drive the creation of value in knowledge-intensive processes. However, there is still no established process-based support due to the dynamic nature of these processes. The latter requires a high level of communication and cooperation between all involved workers. The proCollab research project, hosted at Ulm University, aims to holistically support knowledge workers and knowledge-intensive processes. The concept of *proCollab* relies on the lifecycle-based task management in the context of processes. In particular, knowledge workers may use digital task lists to synchronize and coordinate their work more effectively. To demonstrate the capabilities of *proCollab*, a sophisticated proof-of-concept prototype has been developed. This work presents the design and implementation of the dynamic, web-based user interface of the current version of the *proCollab* prototype.

# Contents

**1 Introduction** **1**

1.1 Problem Statement . . . 1

1.2 Contribution . . . 2

1.3 Outline . . . 3

**2 Fundamentals** **5**

2.1 Knowledge-Intensive Processes . . . 5

2.2 Application Case . . . 6

2.3 proCollab . . . 7

2.3.1 KiP Lifecycle . . . 8

2.3.2 proCollab Components . . . 9

2.3.3 Workspaces . . . 12

2.3.4 Template Repository . . . 13

2.3.5 Configuration Management . . . 13

2.3.6 State Management . . . 14

2.3.7 Specialization Types . . . 17

2.3.8 Object-specific Role-based Access Control . . . 18

**3 Requirements** **21**

3.1 Functional Requirements . . . 21

3.2 Non-Functional Requirements . . . 27

3.3 Current State . . . 29

3.3.1 Current State Analysis . . . 29

3.3.2 Results of the Current State Analysis . . . 33

3.4 Comparable Tools and Systems . . . 34

3.4.1 Basecamp . . . 34

3.4.2 active.collab . . . 35

3.4.3 Flow . . . 38

**4 Concept** **41**

4.1 Navigation Structure . . . 42
4.2 Workspace . . . 42
4.2.1 Breadcrumbs Trail . . . 44
4.2.2 Process Overview . . . 45
4.2.3 Process Assessment . . . 48
4.2.4 Management of Task Trees . . . 50
4.2.5 Task Tree Configuration . . . 54
4.2.6 Task Tree Overview . . . 57
4.2.7 Management of Organizational Units . . . 57
4.3 Template Repository . . . 59
4.4 System Settings . . . 59
4.4.1 Management of Roles and Privileges . . . 59
4.4.2 User Applicants . . . 61
4.4.3 State Model Graph . . . 62
4.5 Account Management . . . 65
4.5.1 Password Recovery . . . 65
4.5.2 Registration . . . 65
4.6 Cross-Cutting Components . . . 68
4.6.1 Confirmation Modal Dialog for Destructive Actions . . . 68
4.6.2 Role Selection . . . 68
4.6.3 Caching . . . 70

**5 Implementation** **71**

5.1 Technologies . . . 71
5.1.1 Representational State Transfer (REST) . . . 71
5.1.2 WebSocket . . . 72
5.1.3 JavaScript Object Notation . . . 72
5.1.4 Angular 4 . . . 72
5.1.5 RxJS . . . 73
5.1.6 Typescript . . . 74
5.1.7 SASS . . . 74

5.1.8 Cytoscape.js . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.2 Implementation Excerpts . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.2.1 Deployment Configurability . . . . . . . . . . . . . . . . . . . . . . . 75
5.2.2 Caching and Updating . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.2.3 Navigation Structure . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.2.4 Confirmation Modal Dialog for Destructive Actions . . . . . . . . . 82
5.2.5 Process Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.2.6 Task Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.2.7 Management of Task Trees . . . . . . . . . . . . . . . . . . . . . . . 90
5.2.8 Task Tree Configuration . . . . . . . . . . . . . . . . . . . . . . . . 90
5.2.9 Process Assessment . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.2.10 Task Tree View . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.2.11 Assign Privileges . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.2.12 State Model Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.2.13 Role Selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.2.14 Role Based User Interface . . . . . . . . . . . . . . . . . . . . . . 101
5.2.15 Registration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.2.16 Password Reset . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

**6 Conclusion** **103**
6.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

# 1 Introduction

As a consequence of the rapidly growing globalization, highly developed countries experience a shift away from producing goods and offering services to knowledge-based economies [Kow11]. In such economies, the support of knowledge-intensive processes (KiPs) and knowledge workers is of high importance. In areas like healthcare, research, or engineering, knowledge workers tightly collaborate to reach a common goal, e.g. the treatment of a patient or the production of a new car. This collaborative work, which takes place in KiPs, is difficult to support with traditional information systems, as KiPs are emerging and unpredictable. In particular, traditional approaches with rigid structures are of little value in supporting KiPs, as they have not been designed to adapt to changing processes.

## 1.1 Problem Statement

Dynamic knowledge-intensive processes are hardly supported systematically as they are influenced by multiple factors and strongly depend on the knowledge of the involved knowledge workers [Tie10]. Furthermore, most of the time, knowledge workers from different domains are involved and work together to reach a common goal. For an effective cooperation, knowledge workers need to always be aware of the current state, potential changes, and completed milestones of the KiPs they are working on. This requires much effort for both communication and organizing in order to ensure a smooth and productive process. As of today, knowledge workers widely rely on task lists (e.g., to-do lists or checklists) for organizing and distributing their work tasks. Traditionally, those task lists are paper-based, not synchronized and managed decentralized. These issues clearly impede the effective collaboration of knowledge workers. In particular, the lack of systematic support currently results in oversights, redundant work, and undesired work results. Due to these problems, there is a prevalent demand for an information system that supports knowledge workers in the context of KiPs in a systematic and sustainable way [MKR12]. Furthermore, such a system may allow knowledge workers, in particular, to manage formerly paper-based processes and tasks digitally. This way, tasks are synchronized and media disruptions can be avoided to increase knowledge worker's productivity.

## 1.2 Contribution

As part of the *proCollab* research project, a sophisticated proof-of-concept prototype has been developed, demonstrating the feasibility and potential of the *proCollab* approach. The *proCollab* approach aims to support knowledge workers by enabling them to cooperate using digital task lists, such as checklists and to-do lists. In the context of KiPs, furthermore, *proCollab* incorporates templates for the creation of such task lists. Thereby, it enables knowledge workers to easily create and maintain task lists, which increase productivity and follow best practices. The current *proCollab* prototype is separated into a scalable server application and a lightweight web-based client interacting with the server. The aim of this master thesis is to enhance and improve the web-based client, to further increase the user acceptance. Therefore, detailed requirements are compiled and analyzed first. The result of this analysis is used to discuss the conceived enhancements and their specific implementation. In particular, the performance of the application must be improved through reducing calls to the server. In this context, data received from the server should be cached and kept up to date using information provided by a WebSocket connection. This approach minimizes time consuming calls to the server, while also avoiding stale data in the cache. Besides caching, this work also includes new concepts for displaying task list to knowledge workers. Due to the emergent nature of KiPs, different, but valuable user interfaces presenting task lists and tasks are crucial to let knowledge workers assess the current progress efficiently. Hence, current user interfaces, presenting task lists are reevaluated to provide as much information as possible without sacrificing usability. Additionally, the ability to configure the instantiation of task lists based on pre-defined contextual situations is implemented. The latter are used to include task lists and tasks making task lists more adaptable to different situations. Furthermore, advanced functionality to manage task and task lists is incorporated to enable knowledge workers to adapt task lists to changes in the underlying KiP. In particular, this includes, moving tasks and even entire embedded task lists through drag and drop. Moreover, roles are assigned to users to provide enhanced access control. Thus, support to switch to different roles is implemented. Because the different roles may have varying permissions, the user interface should adapt to the role of the user, displaying only relevant content.

## 1.3 Outline

This thesis is organized as follows. Chapter 2 introduces the fundamental concepts this thesis is based on. Chapter 3 provides a requirements analysis as well as an overview of the current state of the web-based client and comparable tools. Subsequently, Chapter 4 discusses different solutions for realizing the required new features and enhancements discovered in Chapter 3. Chapter 5 provides selected excerpts of their implementation. Finally, Chapter 6 concludes this thesis with a summary and an outlook on future work.

# 2 Fundamentals

In this chapter, fundamentals that are required in the following chapters are introduced and a terminology is given. Section 2.1 introduces the term knowledge-intensive processes. Whereas, Section 2.2 provides an application case to ease the understanding of KiPs. Finally, Section 2.3 describes the *proCollab* research project and its approach for supporting KiPs.

## 2.1 Knowledge-Intensive Processes

To ensure a common and consistent understanding of knowledge-intensive processes, the definition of [Vac+11] is used in this work.

**Knowledge-intensive processes (KiPs)** are processes whose conduct and execution are heavily dependent on knowledge workers performing various interconnected knowledge intensive decision making tasks. KiPs are genuinely knowledge, information and data centric and require substantial flexibility at design- and run-time.

KiPs are still not widely supported, as they are unpredictable, emergent, goal-oriented and accompanied by an ever-growing knowledge base [MKR12]. These properties make KiPs hard to support with tools used in traditional business processes. The latter typically comprise routine work that can be standardized well. By contrast, KiPs require constant adjustments due to their emergent nature. Furthermore, KiPs do not always have a fixed outcome, but are characterized through a common goal. This goal can be divided into smaller sub-goals. As soon as these sub-goals are completed, the main goal is completed as well. To achieve their goals, knowledge workers constantly define tasks that need to be accomplished. These tasks can often not be completed by a single knowledge worker, instead they require the collaboration of multiple knowledge workers. This is another important aspect as one KiP may require many knowledge workers, maybe even from different domains, to work together to achieve a common goal.

## 2.2 Application Case

To ease the understanding of KiPs, the process of creating a website is used as application case for this work. This application case was chosen because it reveals many of the difficulties knowledge workers face, while working on KiPs.

The development of a new website starts with an evaluation of the old website, if there is any. Then, the requirements for the new website have to be discussed with the customer. These requirements and technical details have to be documented as well. Next, it has to be checked whether the old website can be updated or whether a migration is necessary. It also has to be checked whether the requirements are consistent and completely documented. Subsequently the server credentials have to be provided and the customer has to select content from the old website that should be migrated later. After reviewing the technical requirements and presettings, a first draft of the new website can be created. This first draft then needs to be compared with the documented requirements. Furthermore, the draft is discussed with the customers and updated according to their wishes. After checking that the draft has been updated, the customers shall approve the changes. Furthermore, the content of the old website, which was selected previously, is migrated to the new one. After assuring that all transferable content has been migrated successfully, the current state of the new website is discussed with the customers. Thereby, the website is presented to the customers and updated based on their feedback. Afterwards the spelling of the contents is checked and issues, such as missing content or low quality images, are identified and fixed. Finally, the website release needs to be approved by the customers. If everything is to their satisfaction, the website is made accessible to the public.

The application case shows many of the characteristics of KiPs. The knowledge workers have to adapt to the customers wishes and incorporate new knowledge into their work at all stages of the process. There is also a strong need for communication between the different actors (i.e. the developers and customers).

## 2.3 proCollab

This work is part of the *proCollab* research project started at Ulm University in the summer of 2012 [Pro17]. The goal of this project is to holistically support knowledge workers and their KiPs. To achieve this, *proCollab* takes into consideration that the tasks comprised in a KiP, often have to perform the stages of planning work, performing work, studying work and optimizing plans. These stages can generally be abstracted by the generic Plan-Do-Study-Act (PDSA) cycle [DJB95][MR17c] (see Figure 2.1).

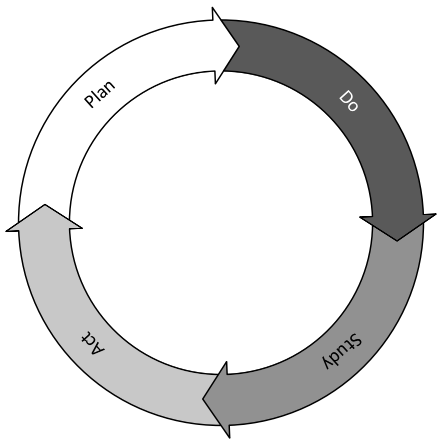


Figure 2.1: PDSA Cycle

Knowledge workers often use tools like task lists (e.g., to-do lists or checklists) to keep track of their tasks while trying to achieve a common goal. Task lists are historically paper-based—this fact obviously entails many disadvantages. It is, for example, impossible for multiple knowledge workers to work on the same task list at once. Another problem arises when multiple task lists are used, as they cannot easily be synchronized. This can lead to consistency and coordination problems. Therefore, the *proCollab* approach provides digital task lists that all involved knowledge workers may access in the context of processes.

### 2.3.1 KiP Lifecycle

In [MKR12], the KiP lifecycle was introduced (see Figure 2.2) as an essential foundation for every approach supporting KiPs. This lifecycle consists of four different phases that are being discussed in the following, building on the definitions provided in [MR14].

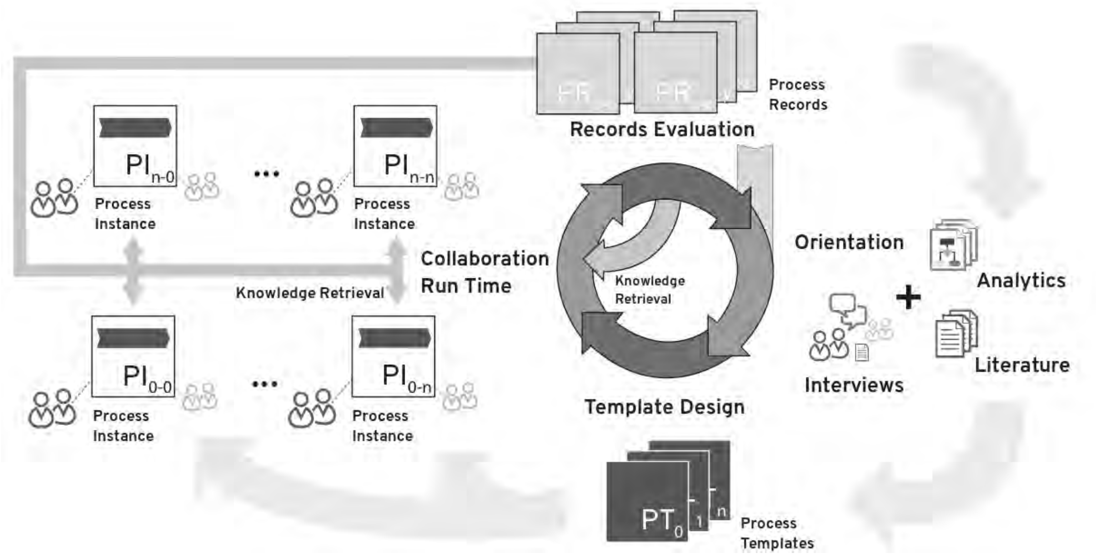


Figure 2.2: KiP Lifecycle [MR14]

**Orientation Phase:** First, the context and goal of the KiP are defined. Subsequently, information is collected to provide an overview on how the knowledge workers collaborate. For this purpose, these knowledge workers are interviewed and expert knowledge and literature is evaluated. In addition, the different tasks, which have to be completed in order to achieve the goal of the KiP, are documented and integrated. Furthermore, it has to be analyzed and documented how the different knowledge workers communicate with each other.

**Template Design Phase:** Based on the collected information, a process template (PT) may be defined for the corresponding KiP (see Section 2.3.2). A PT comprises the coordination artifacts likely used by knowledge workers during KiP enactment. Since the main goal of templates is reusability they should be designed as generic as possible and as specific as required. The created PTs can then be instantiated by the knowledge workers at collaboration run time.

**Collaboration Run Time Phase:** To initiate KiP guidance, knowledge workers may create a process instance (PI). Such a PI may be created using a suitable PT or it is created without any template. The PI then determines the guidance offered by the approach to the knowledge workers involved in a specific KiP instance. Because KiPs are emerging and dynamic, knowledge workers may continuously adjust PIs to achieve their goal and to synchronize each other more effectively according to the given situation.

**Records Evaluation Phase:** Completed PIs can be seen as a valuable resource of knowledge. On the one hand, knowledge workers involved in a PI may make use of insights gained from comparable process records (i.e., archived PIs). On the other hand, process records can be used to systematically examine how knowledge workers collaborate in the context of KiPs and how they accomplish their task. The results of such examinations can then be used to optimize existing PTs to better support future KiPs.

### 2.3.2 proCollab Components

The *proCollab* approach uses a meta-model comprising processes, task trees, and tasks (see Figure 2.3). Task trees are used to support the task lists commonly used by knowledge workers. Each process may contain multiple task trees containing the tasks required to achieve the goal of the process. In addition, each task tree may contain sub-task trees to refine parts of the task tree. The processes, in turn, may also contain subordinated processes in order to address sub-goals or to structure complex KiPs. To enable lifecycle-based task management processes, processes, task trees and tasks are refined by process instances, task tree instances and task instances. The instances may be generally created from scratch or they are derived from templates, i.e., process templates, task tree templates and task templates. The use of templates allows knowledge workers to better optimize processes as the process templates can be periodically updated with the knowledge gained in previous KiPs. To support domain-specific requirements, all *proCollab* key components expose a current state based on a flexible state management concept. This enables these key components to be as specific as needed [MR17c].

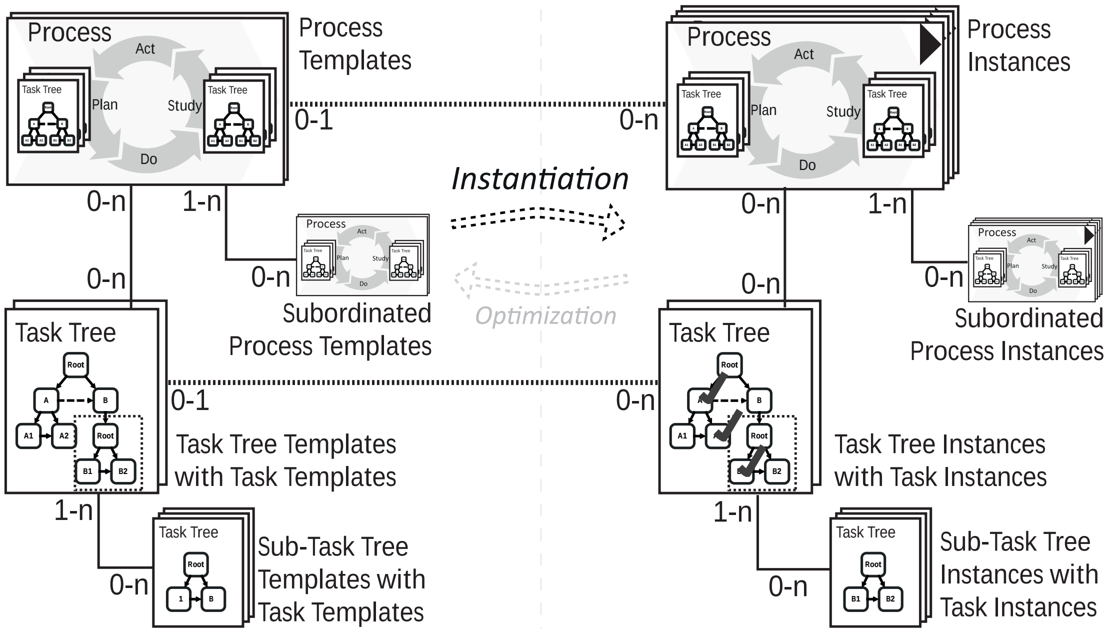

Process
Act
Plan
Study
Do
Task Tree
Process Templates
0-1
0-n
Process Instances
Instantiation
Optimization
0-n
1-n
Subordinated Process Templates
Subordinated Process Instances
Task Tree
Root
A
B
A1
A2
B1
B2
Task Tree Templates with Task Templates
Task Tree Instances with Task Instances
Sub-Task Tree Templates with Task Templates
Sub-Task Tree Instances with Task Instances

Figure 2.3: proCollab Key Components [MR17c]

In the following, the *proCollab* key components are presented in more detail to establish a framework for entities, representing KiPs.

### Processes

Processes represent the collaboration in projects, cases or temporary endeavors. Processes can be arbitrarily nested but are always aimed at achieving one goal. Processes contain conditions, linked resources, and organizational assignments (e.g. roles and corresponding rights) which are needed to achieve the pre-defined goal of the process. Every process may link to an arbitrary number of task trees.

### Task Trees

To provide task list support, *proCollab* uses tasks and task trees, in the following called task tree elements. Task trees are generic data structures that can be used to constitute any task list, such as checklists (e.g., for quality assurance) or to-do lists (e.g., to expedite a process). Task trees have a recommended order in which they should be processed. This order, however, is not set in stone and can be altered if need be. Moreover, task trees allow knowledge workers to iteratively specialize tasks by defining more detailed sub-tasks. These sub-tasks are supposed to be accomplished in order to finally complete the parental task.

### Tasks

Tasks always feature a work description and a current state. Each task can additionally reference necessary resources for its completion.

### Templates

*proCollab* templates may be used to accelerate the planning and coordination of new or changed processes. Templates can be instantiated to create a new instance with all the linked task trees, tasks and sub-processes defined in the template. In general, *proCollab* features process, task tree and task templates.

**Process Template (PT):** When starting a new process, knowledge workers may look for pre-defined process templates that fit their current goal. A PT comprises pre-defined roles with corresponding rights, conditions (e.g. relative due dates), linked resources and linked task tree templates. Using PTs enables knowledge workers to save time, when creating a new process instance.

**Task Tree Template (TTT):** Task tree templates consist of task templates and subordinated task tree templates. They constitute best practices for planning or quality assurance. If a PT contains TTTs, they will be automatically instantiated as soon as the PT is instantiated. Alternatively, a TTT can also be instantiated in the context of an existing process instance.

**Task Template (TT):** TTs are used in one or more TTTs. They may comprise several predefined conditions (e.g., duration, assignments, or connected resources).

### Instances

In *proCollab*, knowledge workers shall collaborate based on instances. *proCollab* provides process instances, task tree instances and task instances. They can either be created from scratch or instantiated using an existing template.

**Process Instance (PI)**: Process instances represent running projects, cases or loose collaborations. They may contain several subordinate process instances. They can be created by instantiating a process template or from scratch. If the process instance is instantiated the linked task tree templates, task templates and subordinate process instances will also be instantiated. A process instance comprises a start date, an end date, assigned goals and linked resources.

**Task Tree Instance (TTI):** Task tree instances represent task trees, e.g., to-do lists or checklists. They can be created through the instantiation of a template or from scratch. Task tree instances comprise task instances and subordinate task tree instances. Every task tree instance is either linked to a process instance or embedded in another task tree instance.

**Task Instance (TI)** Task instances may be added to task tree instances. They can be created through the instantiation of a task template or from scratch. After creation, they expose a state that can be changed depending on the progress of the associated work task.

## 2.3.3 Workspaces

The different *proCollab* components are linked to either a workspace or a template repository (see Section 2.3.4). Thereby, allowing different workspaces to be used to separate different domain form one another. This allows for *proCollab* to be used in different domains at once. For example in a healthcare setting, one workspace could be used for patient care, while another one is used for research projects.

### 2.3.4 Template Repository

*proCollab* provides a template repository, enabling knowledge workers to share templates across multiple workspaces, in order to provide templates that follow best practices. The templates provided by the template repository may be imported into a workspace to apply changes and to share it with other knowledge workers.

### 2.3.5 Configuration Management

To make task tree templates reusable in different contexts and to decrease the effort required to build up a task tree template, configuration support is needed. A configuration support enables knowledge workers to use generic TTTs that more fine-grained TTTs can be integrated into, in order to create the overall TTT matching the present requirements. Thereby, TTTs can be designed in a reusable and modular way, minimizing the efforts needed for creating a TTT for a specific situation, significantly [MR17a]. This approach is implemented in *proCollab* using configuration parameters, contextual situations and configuration specifications, which are described in the following.

**Configuration Parameters:** A configuration parameter consists of a name, a type (e.g., string or Boolean) and a default value. Configuration parameters are used in regular expressions that are part of contextual situations (see below).

**Contextual Situations:** Contextual situations can be used to adjust process templates to specific situations, for example, a process template supporting a surgery can be adjusted to support an emergency surgery by providing additional tasks or omitting them. Contextual situations are triggered by regular expressions using configuration parameters (see Section 2.3.5). These regular expressions are applied to parameters, which are provided when a PT containing a contextual situation is being instantiated.

**Configuration Specifications:** A configuration Specification contains a map data structure that lists all the TTTs and TTs belonging to a contextual situation. This map also provides information regarding into which task tree and at what position each TTT and TT should be injected.

### 2.3.6 State Management

*proCollab* aims to support a wide range of KiPs along with their individual methodologies (see Section 2.3.2). For this purpose, *proCollab* incorporates a generic and flexible state management concept that is able to support different, specialized states. In particular, the *proCollab* state management employs *reference state models*, *state models*, and *state model instances* (see Figure 2.4). The stateful key components comprise PTs, PIs, TTTs, TTIs, TTs, and TIs (see Section 2.3.2).

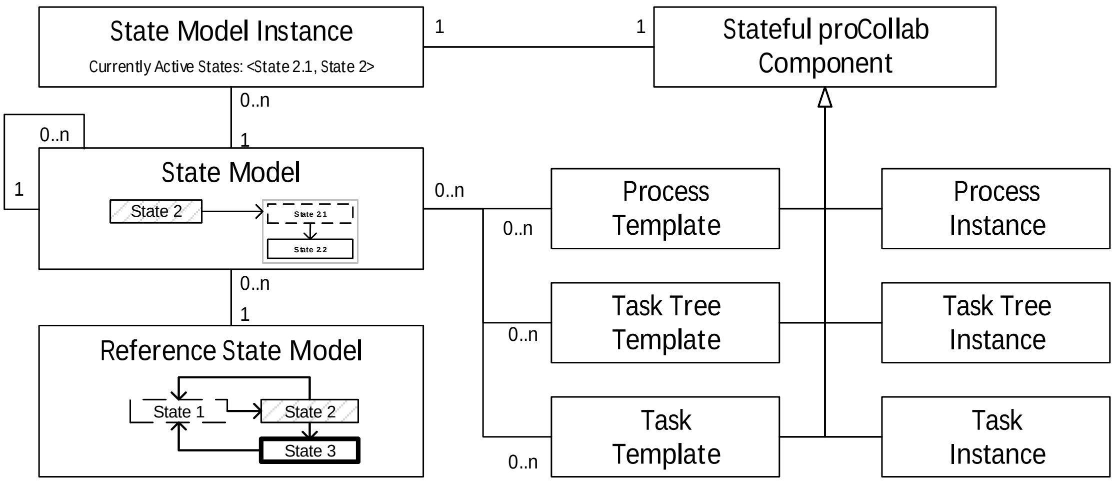


Figure 2.4: *proCollab* State Management Entities [MR17c]

Reference state models declare the states and state transitions that entities of a particular type share. For example, all PTs share the states "Deposited", "Available", and "Archived". In turn, state models may refine any given reference state model to meet domain-specific requirements. For example, the knowledge workers of the application case (see Section 2.2, may want to employ the Scrum procedure [Sch04], to develop the website. The sprints of Scrum are a variation of the PDSA cycle [Bus12] and, therefore, the key phases of this PDSA cycle may be refined to match the specific requirements. Finally, every *proCollab* component references a state model instance that is relying on a pre-selected state model [MR17c].

### Reference State Models

A *reference state model* contains a *state transition graph*, a set of refinable states and a scope. The reference state models for the *proCollab* components introduced in Section 2.3.2 can be seen in Figure 2.5. In general, every state transition graph consists of states and transitions. The latter define the states that can be reached from any given state. The state transition graph also denotes a start state, one or more final states as well as a scope to define for which *proCollab* components it may be used. For example, a reference state model with the scope "Process Instance" constitutes the basis for all state models of process instances. To support domain-specific requirements, the reference state model may expose states that can be further refined using state models.

### State Models

*ProCollab* provides state models enabling users to map the different work phases to different states. State models are linked to reference state models and inherit the scope of the corresponding reference state model. Figure 2.6 shows an example of a state model that refines the refinable state "Running" of the reference state model discussed in Section 2.3.6.

### State Model Instances

State model instances are used to make *proCollab* components stateful. Every state model instance references a state model and contains a sequence of currently active states and a constraint named *strict*. The strict constraint determines whether all refined states have to be in the completed state before the next outgoing transition can be called.

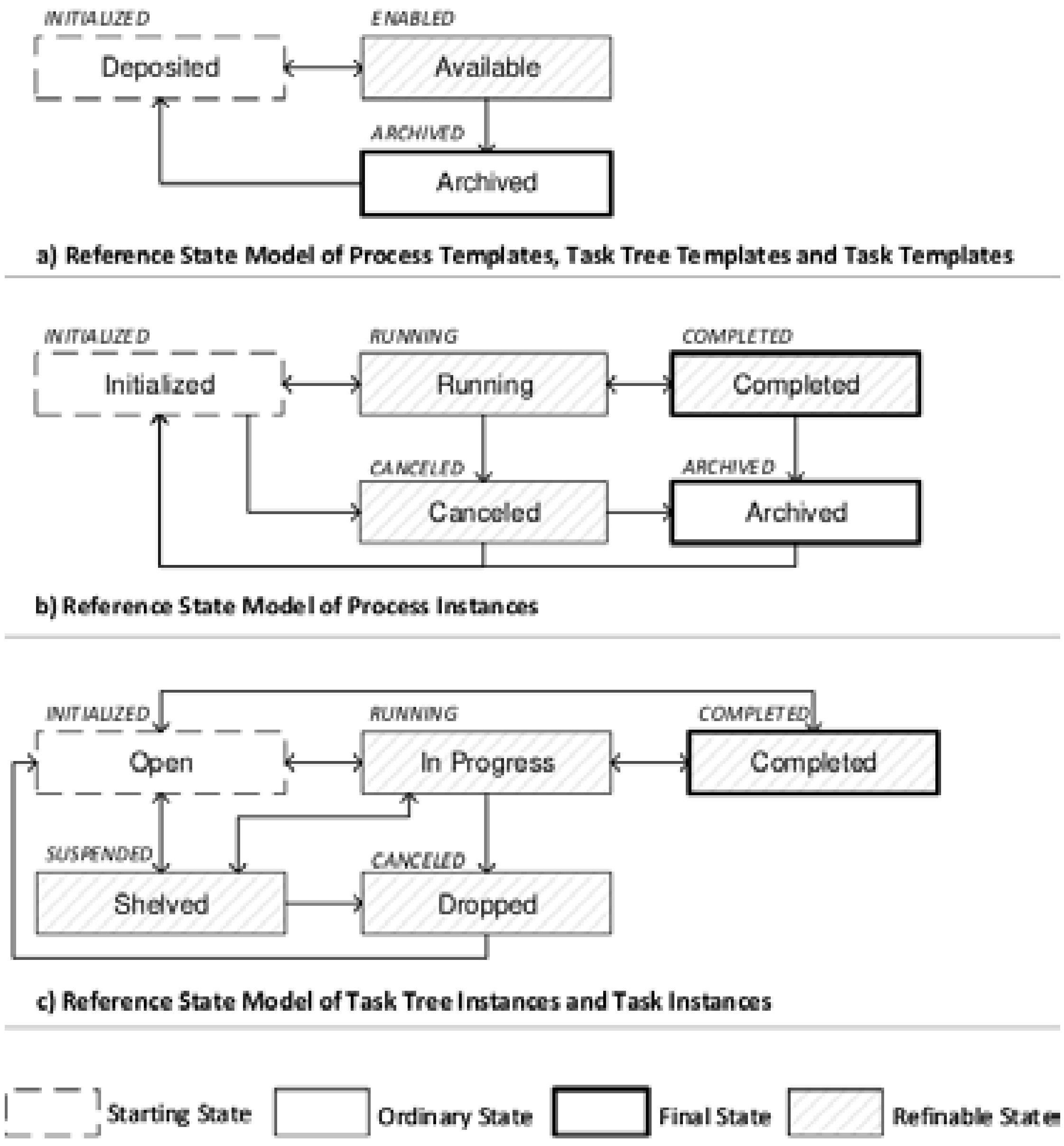


Figure 2.5: Reference State Models [MR17c]

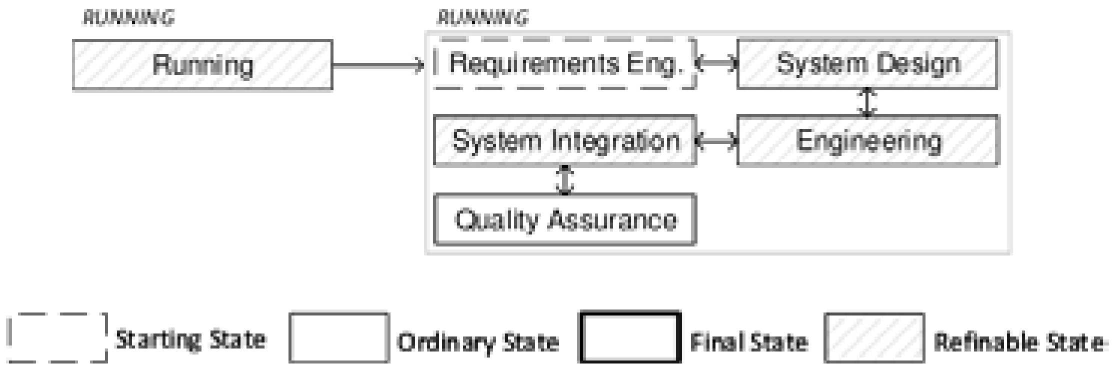


Figure 2.6: Individual State Model for *proCollab* Process Instance [MR17c]

### 2.3.7 Specialization Types

To enhance the generic data structures of processes, task trees and tasks *proCollab* employs specialization types (see Figure 2.7). Consequently, the most common types are process types, task tree types and task types. For example, the application case is using the process type project to specialize the process templates and instances. Projects contain to-do lists and checklists, which both are specialized task tree types. To-do items and check items in turn are the specialized task types.

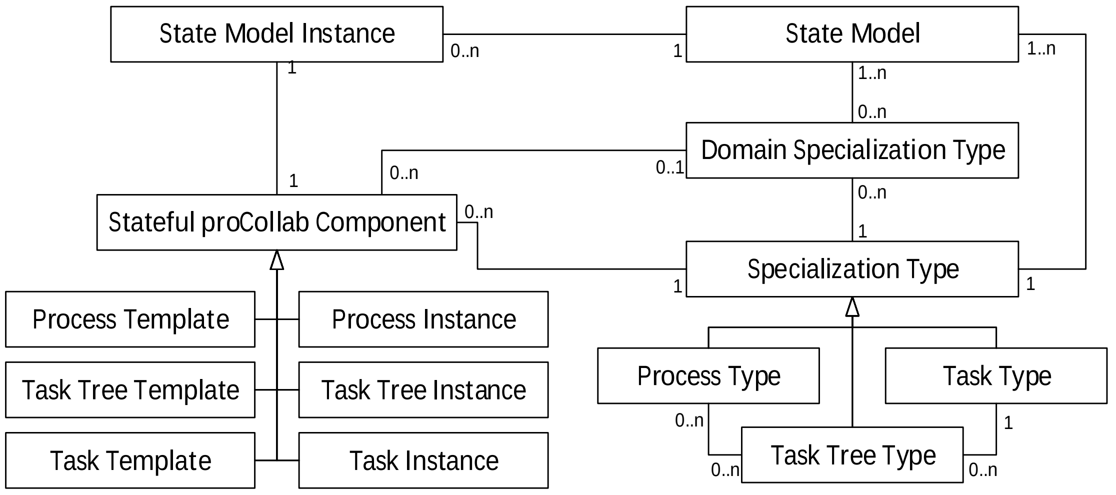


Figure 2.7: Specialization Entities [MR17c]

To support a wide variety of use cases, every specialization type may reference a set of state models that can be used when creating templates and instances.

Furthermore, every specialization type exposes a temporal perspective to support knowledge workers according to the PDSA cycle (see Section 2.3.2). The temporal perspective determines whether a *proCollab* component is supposed to be used for planning (*prospective temporal perspective*), checking (*retrospective temporal perspective*) or if it is a hybrid for both planning and checking (*hybrid temporal perspective*). For example, in the application case the to-do lists expose the prospective temporal perspective as they are used for planning whereas the checklists expose the retrospective temporal perspective as they are used for quality assurance.

### 2.3.8 Object-specific Role-based Access Control

To provide access control, *proCollab* incorporates a powerful role and permissions model, based on the concept of object-specific role-based access control (ORAC) [MR17d]. ORAC enables the close integration of access control with the given object model as well as the support of roles in a fine-grained, object-specific way. It comprises the components *guarded objects*, *privileges*, *object-aware roles*, *organizational entities*, *agents* and *object aware role assignments* (see Figure 2.8). *Guarded objects* are objects that are protected by ORAC, i.e., an agent can only manipulate data if he was granted access to this action by ORAC. *Organizational entities* allow to model the organizational context of agents, i.e. they are made up of organizational units (e.g., HR department), organizational roles (e.g., director), and abilities (e.g., office skills). These organizational units can be nested to model the structure of an organization. The organizational role may then be used to indirectly assign *privileges* the users with that role.

Object-aware role assignments are used to tie together the different components comprising ORAC, namely agents, object-aware roles and guarded objects. Every object-aware role contains a key scope and optionally a number of additional scopes. The scopes are used to assign privileges regarding different object types to the object roles and allow the creation of object-aware role assignments. These privileges determine which actions can be performed on a guarded object and in which context the privileges are applicable. To allow for a hierarchical application of privileges an object-aware role may reference a set of hierarchical privileges for every scope. Furthermore, a set of entity-related privileges may be referenced to provide a rich modeling of object-aware roles. In the application case ORAC can also be applied to provide role-based

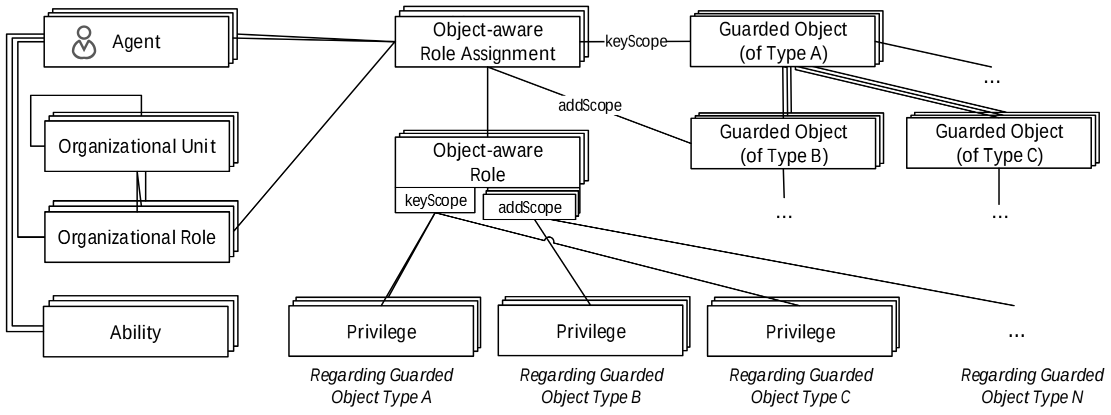

Agent
Organizational Unit
Organizational Role
Ability
Object-aware Role Assignment
keyScope
addScope
Guarded Object (of Type A)
Guarded Object (of Type B)
Guarded Object (of Type C)
Object-aware Role
keyScope
addScope
Privilege
Privilege
Privilege
...
Regarding Guarded Object Type A
Regarding Guarded Object Type B
Regarding Guarded Object Type C
Regarding Guarded Object Type N

Figure 2.8: Overview of Object-Specific Role-Based Access Control [MR17d]

access control. First, the key scope of the object-aware role *Website Developer* would target the process instance *Website Development Process*. The key scope is linked to the various privileges needed to update the development process as well as to the hierarchical privileges to manage child (guarded) objects (i.e., to-do lists, checklists and sub-processes). As a result, an agent, who is assigned to the object-aware role *Website Developer* and a *Website Development Process* guarded object, may manage all child *Website Development Process* objects, needed to accomplish the goal.

# 3 Requirements

This chapter discusses the different requirements that the *proCollab* client needs to fulfill in order to holistically support knowledge workers and their KiPs. Previously conducted research, at Ulm University, included several case studies, primarily in healthcare (e.g., ward rounds and patient treatment) and in the automotive domain (e.g., E/E engineering) [LR07] [MKR12][Pry+14][TRH13]. As a result, a set of key requirements, for the systematically support KiPs was derived [MR14]. Based on these requirements and the guiding principles for web usability (see [Kru14]) a set of requirements has been derived for the web client. Furthermore, the current state of the *proCollab* proof-of-concept prototype is discussed to assess which requirements can already be met with the *proCollab* client in its current state. Finally, comparable tools and systems are discussed to gain information how others have addressed similar challenges.

## 3.1 Functional Requirements

Functional requirements are defined as *statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations* [Som10]. The following sections discuss the functional requirements that apply to the *proCollab* client.

#### FR1: Support of Processes

*proCollab* uses processes to holistically support knowledge workers and their KiPs (see Section 2.3.2). Therefore, the client needs to enable knowledge workers to browse, update, and delete existing processes. Furthermore, functionality to create or instantiate new processes is needed.

### FR2: Support of the Task-centric proCollab Approach

Knowledge workers rely on task lists as central entities for planning and performing their work (see Section 2.3). Since, KiPs are unpredictable and emergent (see Section 2.1), knowledge workers must be able to create and continuously update tasks tree elements. Therefore, functionality is needed to create, edit, update, and delete them.

### FR3: Advanced Task Tree Management

In addition to FR2, knowledge workers should also be able to move task tree elements within a task tree and even between different task trees.

### FR4: Support of Stateful proCollab Components

To support the stateful components provided by *proCollab* (see Section 2.3.6), a user should always be aware of the current state of stateful components. Furthermore, the user should know what states the stateful entity can obtain and how he gets to a desired state. For this purpose, the *proCollab* client needs to provide a way to display the different states and state transitions.

### FR5: Dedicated Task Tree Views

To allow knowledge workers to focus on specific task trees dedicated views are needed to manage the different task tree types (e.g., to-do lists and checklists). These views should enable the user to assess and to update task trees of any kind (see FR2, FR3).

**FR6: Template Support**

To better support knowledge workers while creating new processes or task tree elements, an appropriate template support is needed (see Section 2.3.2). This includes dedicated views to manage existing templates (i.e., to browse, update, and delete them) as well as a view to create new ones. Finally, knowledge workers need the ability to instantiate templates (see also FR9).

**FR7: Template Repository**

Knowledge workers should be able to share templates, following best practices, across workspaces (see Section 2.3.4). Therefore, a template repository should be provided, to host such templates. In addition, the client should allow importing the templates, from the template repository, into the different workspaces.

**FR8: Situation-Specific Task Tree Configuration**

In specific situations (e.g., a case of emergency) only selected parts of a task tree may be needed or additional task tree elements shall be included. To avoid having to create a template for every situation, situation-specific task trees should be implemented (see Section 2.3.5), thereby increasing efficiency and usefulness.

**FR9: Configuration Management**

To fully support FR8, knowledge workers must be also able to manage existing task tree configurations. Therefore, a view is needed to enable users to browse, update and delete existing task tree configurations. Furthermore, users must be able to create new task tree configurations.

#### FR10: Configuration Simulation

Knowledge workers should be able to preview situation-specific task trees before they are instantiated (see FR11). As a result, a view is needed to simulate situation-specific task trees. This view should display the task tree elements that an instantiated situation-specific task tree will comprise.

#### FR11: Configuration Instantiation

In order to instantiate situation-specific task trees a view is needed, enabling knowledge workers to select which task tree configuration they want to instantiate. This view may be used to enter applicable parameters to select the appropriate task tree configuration.

#### FR12: Adaptive, Role-Based User Interfaces

Knowledge workers using the *proCollab* client may obtain different roles with varying permissions and duties (see Section 2.3.8). Hence, the user interface should adapt to the current role and its permissions. As a result, the navigation structure and other views should be adjusted to only display accessible features. Thereby, errors due to insufficient permissions can be avoided.

#### FR13: Adaptive Navigation

To increase learnability and efficiency of the navigation menu, it is important not to overwhelm the user with information. This can be done through changing the content of the navigation menu according to the context in which it is displayed. For example, a view displaying a workspace overview should provide a different navigation menu, than a view displaying a process. Thus, enabling access to context specific views.

**FR14: Navigation Awareness**

A user should always know where in the client he is and what he can do there. In particular, this includes, informing the user about what view he is currently looking at and the path from the initial view of the client to this view. Thereby, the context of the current page can be quickly established. Additionally, the displayed path should enable him to easily switch to previously viewed pages.

**FR15: Support the Sharing of URLs**

To improve collaboration, knowledge workers should be able to easily share URLs pointing to specific views of the *proCollab* client. Thus, a link should include all essential information needed to display any given view. For example, the information needed for navigation awareness or context-specific views.

**FR16: Switching Roles (Manually/Automatically)**

By default, in *proCollab*, the user should always operate with the least privileged role possible. Switching to the least privileged role should be performed automatically. This ensures that knowledge workers can focus on the task at hand. To perform administrative duties, users should be able to switch their roles manually. This helps to separate administrative actions from normal use as well as improve the data quality of process records.

**FR:17 Role Management**

To support the role-based *proCollab* approach (see FR16), privileged users have to be able to manage and to assign the different roles. This requires a user interface that presents the different roles and allows users to create new roles. Moreover, views are needed to update and delete existing roles. Finally, users need the ability to assign privileges to the different roles (see FR19).

### FR18: Privilege Management

To manage the many different privileges assignable to a role, a view is needed, that allows filtering the privileges according to their context. This view should also enable the user to see at once, which privileges are available and which have been already assigned.

### FR19: Adding Role Assignments

To make use of the available roles, it has to be possible to assign these roles to different users. Therefore, a user interface is needed, that allows assigning existing roles to the users based on ORAC (see Section 2.3.8).

### FR20: Organizational Model Support

To allow *proCollab* to model organizational structures, a view is needed that displays the organizational structure. Additionally, functionality to update the organizational structure as well as to add and remove organizational roles is needed.

### FR21: Authorization

To ensure privacy and accountability, only authorized users should have access to the *proCollab* client. To achieve this, knowledge workers can be assigned to different roles with varying permissions. Based on these roles the access to different entities (e.g., of PTs and PIs) can be defined.

### FR22: User Registration

In order to properly implement authorization (see FR21), a new user should be able to create new accounts through filling out a respective application. System administrators, in turn, should be able to accept or reject those applications.

#### FR23: User Management

If, for example, a user is unable to register himself with the system (see FR22), administrators should be able to manually create new accounts, to ensure access to the system.

#### FR24: Password Recovery

If a user has lost his password, he should be able to recover his account. In particular, without the need to involve an administrator, thus supporting scalability. Therefore, a view providing a simple account recovery is needed.

#### FR25: Mobility

To increase user commitment, mobile access is needed, as knowledge workers may work from different locations, and on different devices.

#### FR26: Events

To increase awareness of knowledge workers, the *proCollab* server emits information about occurring events (e.g., the arrival of work results). The client view should use these events to keep the user interface always up to date.

## 3.2 Non-Functional Requirements

Non-functional requirements are defined as *constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards* [Som10].

This section discusses the non-functional requirements that apply to the *proCollab* client.

#### NFR1: Maintainability

The code should be, well organized and documented to enable easy maintenance. For example, this means to adhere to common best practices. For example, the style guides set forth by the angular team [Ang17c]. Furthermore, the project should be organized in a modular way, allowing for easy reuse and replacement of the different components.

#### NFR2: Performance

It is crucial for user acceptance that waiting times are kept at a minimum. Thus, powerful caching needs to be incorporated in the *proCollab* web client used to avoid duplicate requests or stale data.

#### NFR3: Robustness

The program should be able to recover from errors, including invalid data and unexpected operating conditions, without inconveniencing the user. This helps to increase the user acceptance and productivity [CL05].

#### NFR4: Learnability

To improve learnability, the different menus and inputs of the client should follow consistent principles and use a consistent layout. In particular, confirmation modal dialogs and forms should always be designed in the same way.

#### NFR5: Efficiency

To avoid unnecessary work, changes to the *proCollab* entities should be visible to the user instantly. This especially includes updating the view without reloading the client. Thereby, user acceptance is improved through avoiding unnecessary reloads and errors when users inadvertently work on data that is not up to date.

#### NFR6: Memorability

Memorability can be a big factor in whether people adopt an application for regular use [Kru14]. Therefore, it is important to create a pleasant user experience, to ensure that users will continue to use the client.

#### NFR7: Error Handling

To minimize the negative effects errors may cause, knowledge workers should always receive clear and succinct messages when an error occurs. The message should include the reason for the error, how it can be fixed, and what can be done to avoid it in the future.

## 3.3 Current State

This section examines which of the requirements discussed in Section 3.1 are already addressed by the *proCollab* web client in its current state and which require additional functionality. To achieve this, the current state of the *proCollab* web client is analyzed with the discovered functional and non-functional requirements in mind (see below). Subsequently, the result of this analysis is discussed (see Section 3.3.2).

### 3.3.1 Current State Analysis

The *proCollab* client is a web-based application providing platform independence and mobile access (see FR25). In its current state the client allows knowledge workers to log into the system (see FR21). After logging in, a user can view and edit his profile using a drop-down menu in the upper right corner (see Figure 3.1). The navigation sidebar always displays the same items regardless of the context in which it is shown.

## Workspaces

The topmost entry of the sidebar menu displays the name of the current workspace (see Section 2.3.3). A click on it leads to a new view displaying all available workspaces. In this view, the user may select the workspace within which he wants to work. Additionally, new workspaces can be created and existing ones can be updated or deleted.

## Process Assessment

Next, the sidebar menu lists the entries for the different process types (i.e., "Project" and "Case"). Clicking on these entries leads to a view displaying all available process instances with the corresponding process type. This view also allows knowledge workers to create new PIs. Furthermore, existing PIs can be updated and deleted (see FR1). If a PI is selected using a "Select" button, a new view is shown, presenting all the task trees and sub-projects the process comprises (see Figure 3.1).

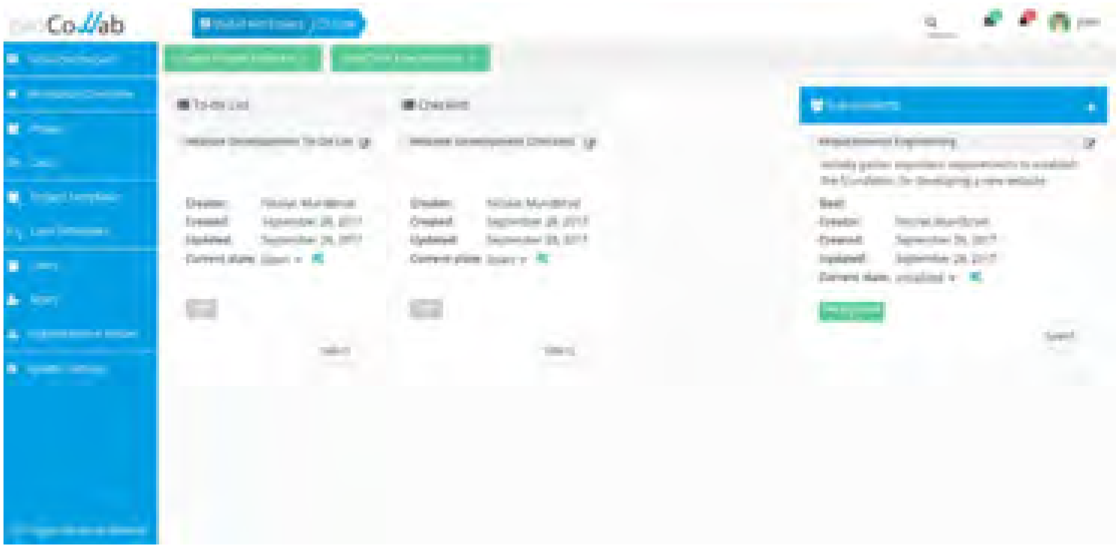


Figure 3.1: Process Instance Overview

In this view new task trees and sub-projects can be added to the PI and existing ones can be updated or deleted. The task trees are shown in containers, providing more information (e.g., its current state and its creator) and the number of tasks each task tree comprises. To view the task tree, it has to be selected by the user. Then another view is shown that presents the task tree and its tasks as a hierarchical list (see Figure 3.2).

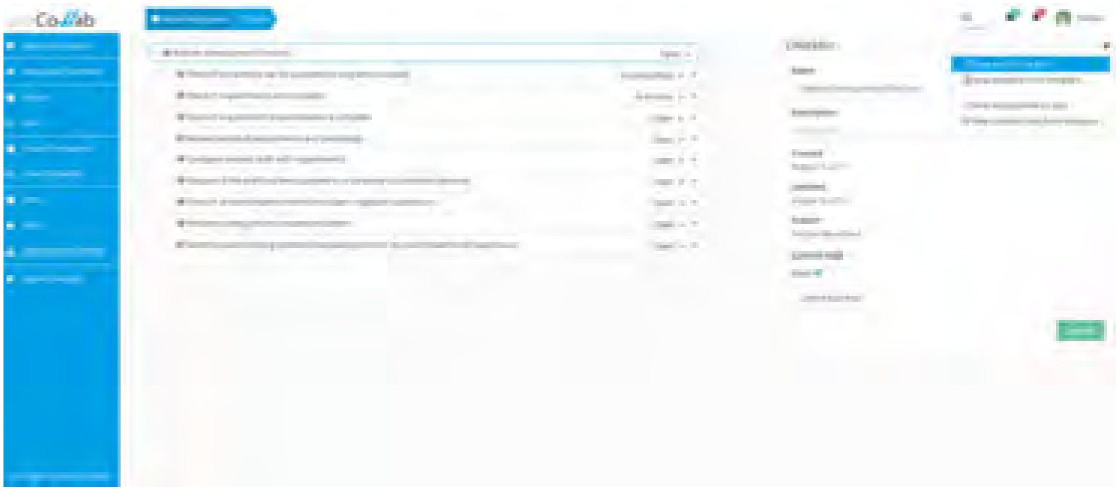


Figure 3.2: Checklist Overview

This view allows knowledge workers to assess the progress of the task tree (see FR5). Using the plus button in the upper right corner, new tasks and task trees may be added to the selected task tree element (denoted with a blue border). Additionally, this view allows a user to change the states of the different tasks and task trees as well as deleting and updating them (see FR2, FR4).

After clicking the icon with a blue cogwheel on the right hand side, a modal dialog is opened showing a graphical representation of the different states and state transitions (see Figure 3.3).

### Working with Templates

The next entries in the sidebar menu (i.e., "Project Templates" and "Case Templates") allow a user to browse the process templates of the corresponding type (see FR6). The workflow for working with templates is analogous to working with instances. First all process templates of the selected type are listed, allowing a user to browse and

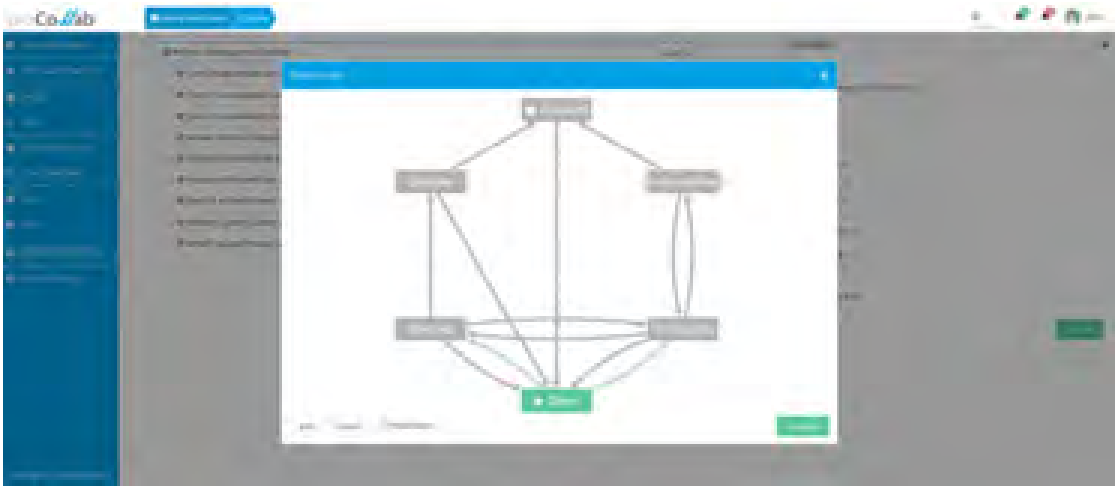


Figure 3.3: View of the States and Transitions of a Stateful Entity

manage them. PTs featuring the *available* state have an "Instantiate" button, enabling their instantiation. Clicking the "Select" button of a PT leads to a new view, listing all the task tree templates and sub-process templates the PT comprises. In this view, the user can manage existing TTTs and PTs as well as create new ones. After the user selected a task tree template using the "Select" button, another view is displayed. This view presents the task tree template and its task templates as a hierarchical list. Additionally, this view may be used by the user to manage the task tree template.

### Additional Views

The "Users" entry of the sidebar menu leads to a view listing all the users of the system in a simple table. This view also allows adding new users to *proCollab*.

The roles view lists all the available roles and allows the creation of new ones. The organizational model enables users to represent organizational structures in a hierarchical tree (see FR26).

Finally, the system settings allow users to create new process, task tree and task types (see Section 2.3.7), as well as to update and delete existing ones.

### 3.3.2 Results of the Current State Analysis

This section provides an overview of the results, discovered in the current state analysis (see Section 3.3.1).

#### Functional Requirements

Section 3.3.1, provided an analysis of the current state of the *proCollab* client. Table 3.2 was created based on this analysis. It shows an overview of the functional requirements (see Section 3.1) that are already fulfilled (+) or partly fulfilled (−).

Table 3.1

| Functional Requirement | Fulfilled |
|---|---|
| FR1 | + |
| FR2 | + |
| FR3 | |
| FR4 | − |
| FR5 | − |
| FR6 | − |
| FR7 | |
| FR8 | |
| FR9 | |
| FR10 | |
| FR11 | |
| FR12 | |
| FR13 | |
| FR14 | |
| FR15 | − |
| FR16 | |
| FR17 | + |
| FR18 | |
| FR19 | |
| FR20 | − |
| FR21 | + |
| FR22 | |
| FR23 | + |
| FR24 | |
| FR25 | + |
| FR26 | |

Table 3.1: Fulfilled Functional Requirements

#### Non-Functional Requirements

The *proCollab* web client was developed in a modular way, and with common best practices in mind. Therefore, it provides good maintainability (see NFR1). Furthermore, it uses a consistent design across different components, improving learnability (see NFR4). During the analysis of the current state, it was established that the client already provides a good overall performance (see NFR2). However, the performance suffered noticeably when large task trees were loaded. As the *proCollab* web client is still in an early development stage, multiple errors occurred. The client recovered from these errors without problems (see NFR3). However, the errors were not always displayed using a clear error message, if any at all (see NFR6). Additionally, the memorability of the client was negatively affected by these errors (see NFR6). The overall efficiency of the client is acceptable but it still misses some vital features, such as a compact overview of a process (see NFR5).

Table 3.2 shows an overview of the non-functional requirements (see Section 3.2) that are already fulfilled (+) or partly fulfilled (−).

| Non-Functional Requirement | Fulfilled |
|---|---|
| NFR1 | + |
| NFR2 | − |
| NFR3 | + |
| NFR4 | − |
| NFR5 | − |
| NFR6 | |
| NFR7 | |

Table 3.2: Fulfilled Non-Functional Requirements

## 3.4 Comparable Tools and Systems

There are many tools that strive to support knowledge workers on the base of a task list. In this section, three of these tools will be more closely examined and discussed.

### 3.4.1 Basecamp

The first tool that is being examined is Basecamp [Bas17]. Basecamp aims to support KiPs by providing a centralized platform to manage task lists, data, and documents. KiPs are organized in projects which comprise a group chat (called campfire), a message board, to-do lists associated with the project, a calendar for scheduling, and linked documents and files. A circular progress bar shows the progress of the project, which is tracked by using to-do lists. The individual to-do list entries have a state (open or checked off) and optionally a start and finish date. Furthermore, it is also possible to assign to-do list entries to specific users and attach associated files to the to-do list entries (see Figure 3.4). Knowledge workers may cooperate in teams, however there is no support for a role based approach similar to the one discussed in Section 2.3.8. Basecamp also does not offer the ability to create templates for different use cases.

Figure 3.4: Basecamp To-do List Overview

To keep its users up to date, Basecamp offers a dedicated view that shows all conducted changes. Additionally, Basecamp offers to send a daily e-mail, containing the latest activities, to the user (see Figure 3.5).

### 3.4.2 active.collab

The next tool that was examined is active.collab [Act17]. KiPs are organized in projects which comprise task lists, discussions related to the KiP, linked files and notes. Task

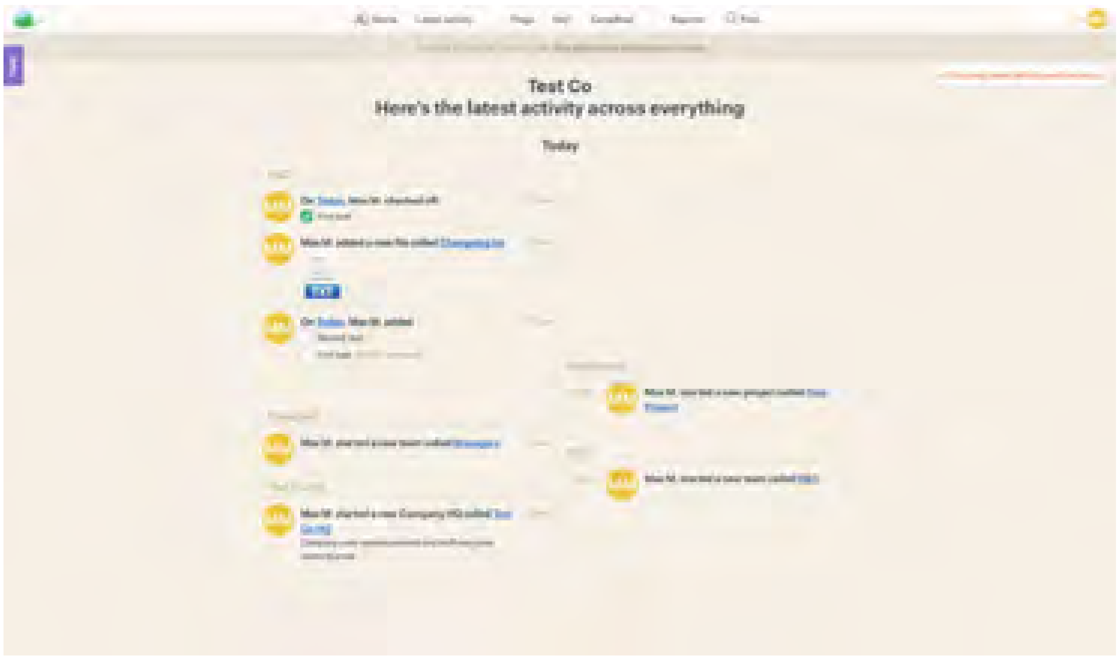


Figure 3.5: Basecamp Latest Activities

lists are used to track progress and guide projects. Tasks feature the states open and completed and can be assigned to specific users. A progress bar shows the current state of the project by tracking how many tasks have been completed and how many are still open. It is also possible to provide further information such as labels, due dates and, priority (see Figure 3.6).

Users of active.collab can have one of the three roles: leader, member or client. Leaders have the ability to manage projects, members can create add and manage task lists in a project and clients can only view task lists and comment on them. To allow for an easier creation of recurring tasks, active.collab provides the ability to create templates that can be used to create new projects with predefined task-lists and files.

The activity view can be used to get an overview of the last activities in the project to retrieve a quick impression of its current state (see Figure 3.7).

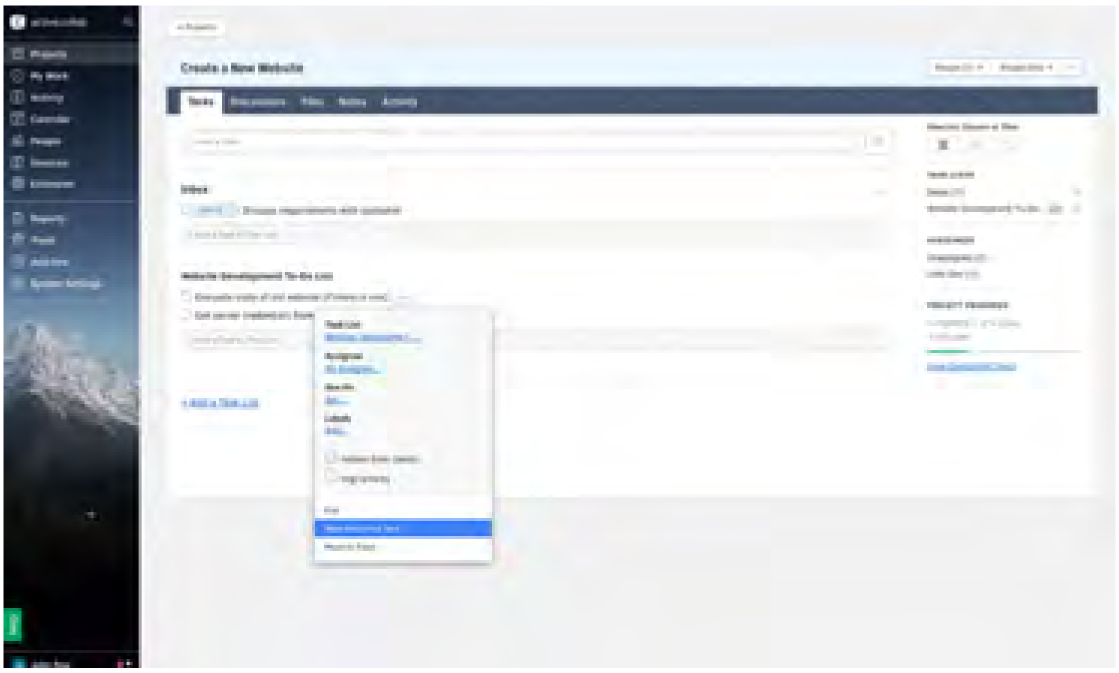


Figure 3.6: active.collab Task List View

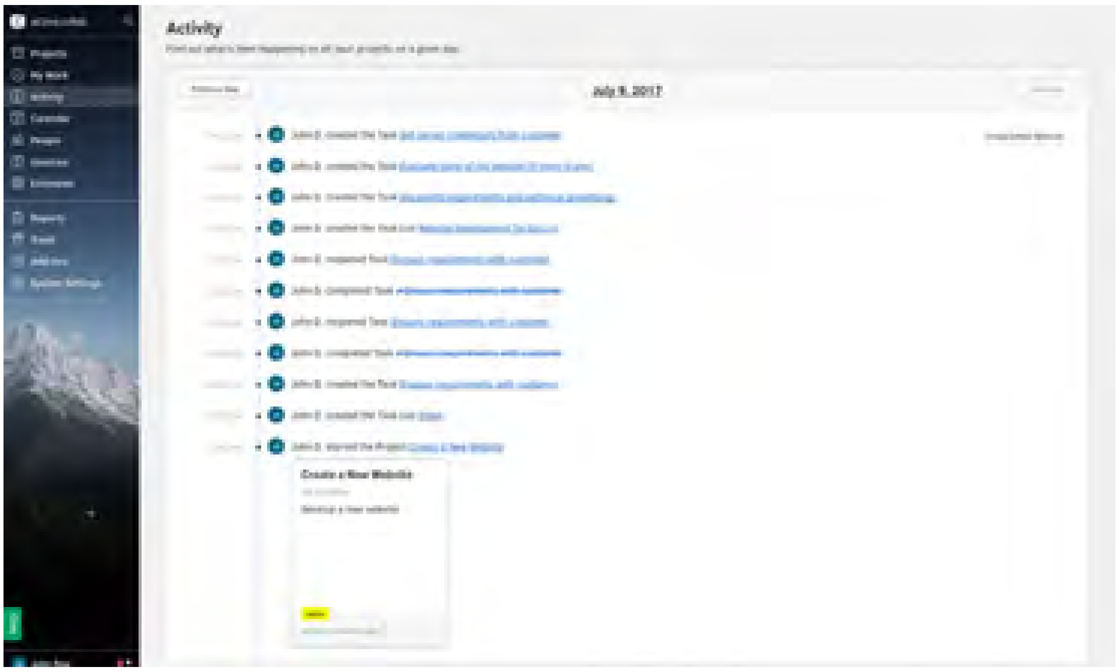


Figure 3.7: active.collab Latest Activities

### 3.4.3 Flow

The last tool that was examined is Flow [Flo17]. Flow uses projects to organize KiPs and progress is tracked and guided using task lists. Tasks comprise a name and a state (open or checked). Optionally tasks can have attached files, sub-tasks and a due date. Users also can comment on the tasks (see Figure 3.8). Teams are used by Flow to keep different types of work separate and to group teams together. Flow does not support the use of templates to create projects with predefined tasks, although it is possible to duplicate existing projects.

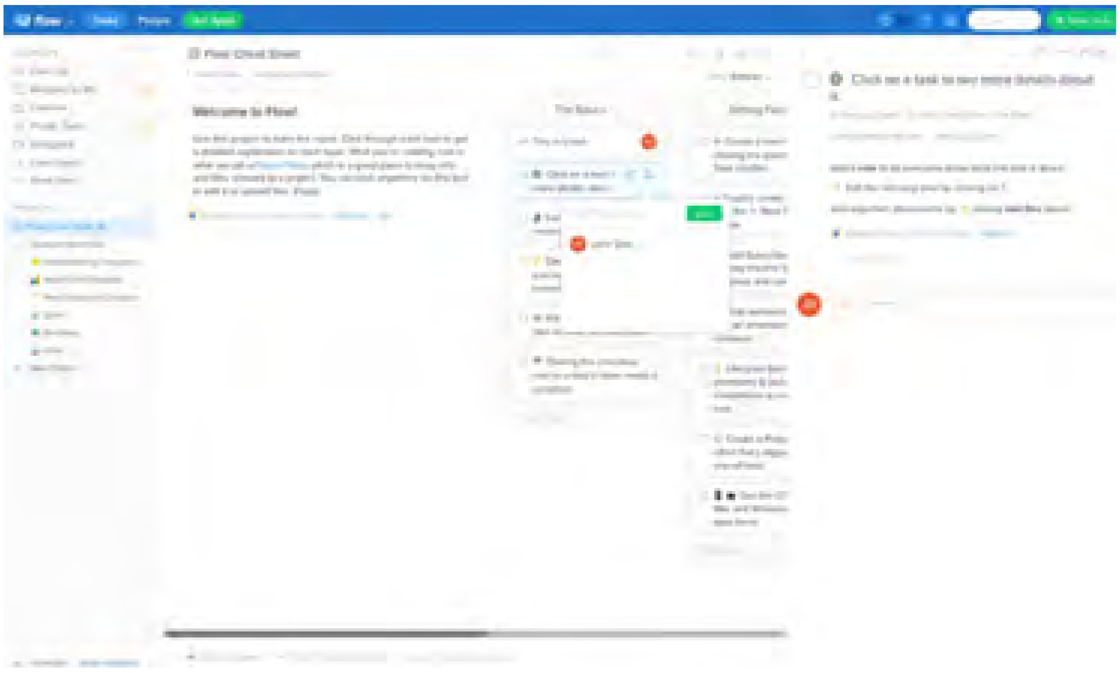


Figure 3.8: Project Overview in Flow

Flow offers a so called *Catch Up* view to keep users up to date on projects by showing them recent changes and suggesting tasks that can be worked on next (see Figure 3.9).

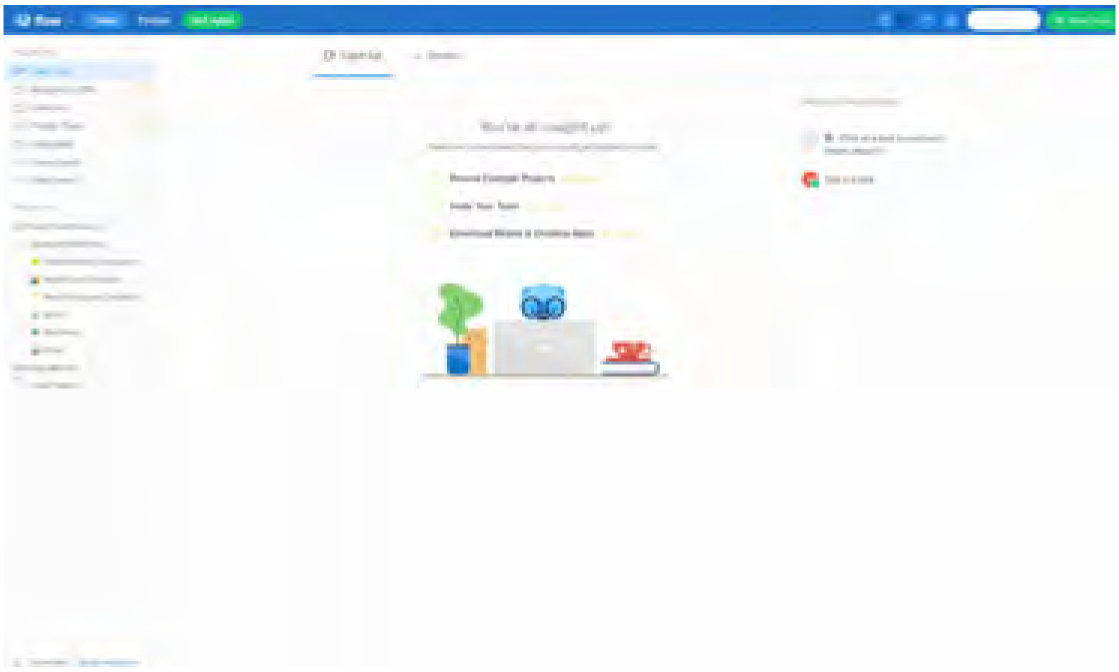
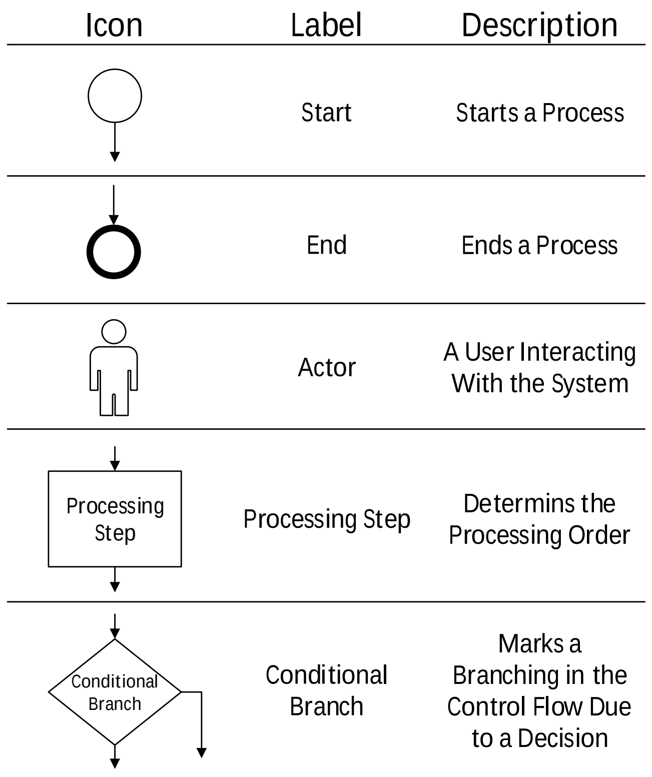


Figure 3.9: Catch Up View of Flow

# 4 Concept

In this chapter, different concepts addressing the requirements specified in Chapter 3 are discussed. The subsequent Chapter 5 focuses on how selected concepts have been implemented. Flow charts are used to illustrate selected procedures using the notation introduced in Figure 4.1.



| Icon | Label | Description |
|---|---|---|
|  | Start | Starts a Process |
|  | End | Ends a Process |
|  | Actor | A User Interacting With the System |
| Processing Step | Processing Step | Determins the Processing Order |
| Conditional Branch | Conditional Branch | Marks a Branching in the Control Flow Due to a Decision |

Figure 4.1: Notation Used in the Flow Charts

## 4.1 Navigation Structure

To provide an adaptive navigation for the *proCollab* web client (see FR13), a concept comprising different navigation menus has to be developed. Based on a thorough analysis, a concept for an adaptive navigation structure was created. Figure 4.2 provides an overview of its different navigation menus and when they are displayed. The different arrows show which menu item has to be selected to reach the subsequent navigation menus. In the following, each menu item is discussed to provide an overview of the navigation structure. For the discussion, the numbers to the right of the items in Figure 4.2 are used.

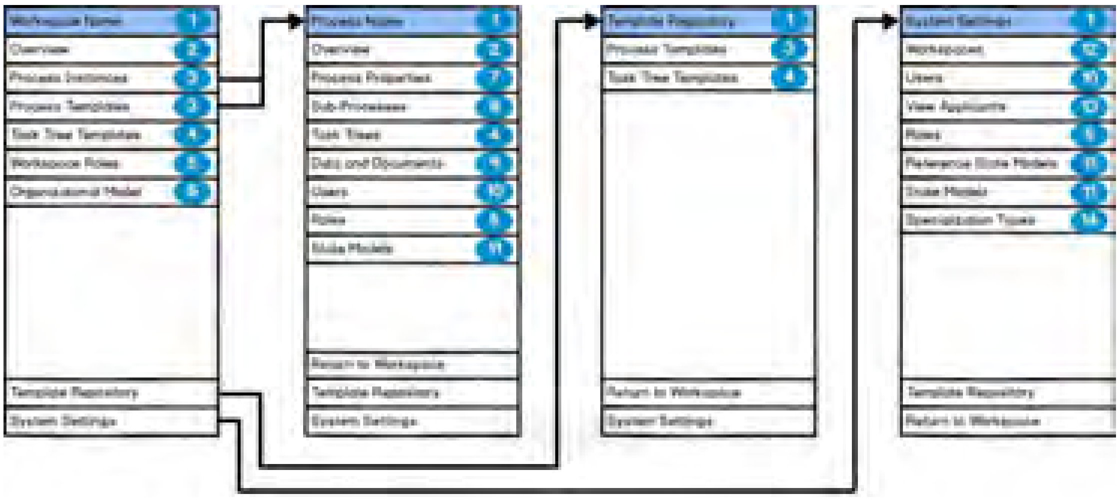


Figure 4.2: Overview of the Navigation Menus and their Interrelationship

## 4.2 Workspace

1. At the top of the navigation, there is always a label informing the user about the context, in which he is currently working.
2. The overview can be used to let knowledge workers know about the latest activities of their co-workers as well as the progress of the different processes.

3. In the view for process instances or process templates view, all the different process instances or templates are listed, allowing knowledge workers to browse and manage them (see Section 4.2.2).
4. The task tree view provides an easy way to manage the different task trees (see Section 4.2.4).
5. In the role view, all the available roles are listed. Knowledge workers may update or delete them as well as they may add new ones (see Section 4.4.1).
6. The organizational model view provides an overview of the organizations and the different organizational roles (see Section 4.2.7).
7. The process properties view allows knowledge workers to get a quick overview of the different properties of a process as well as the ability to update them.
8. The sub-processes view lists all the sub-processes of a process (see Section 4.2.2).
9. The data and documents view presents all the data and documents related to the currently selected process.
10. The users view provides a list of all the users assigned to the selected process.
11. The state models view provides an overview of all the state models connected to the selected process.
12. The workspaces view lists all the available workspaces.
13. View applicants displays all the applicants (i.e., users who submitted a registration form) to the *proCollab* system, and allows for approving or rejecting their applications (see Section 4.4.2).
14. The specialization types view allows knowledge workers to create new process, task tree and task types.

The menu entries at the bottom of the navigation are used to let knowledge workers change into the template repository or system settings view, or to return to the current workspace.

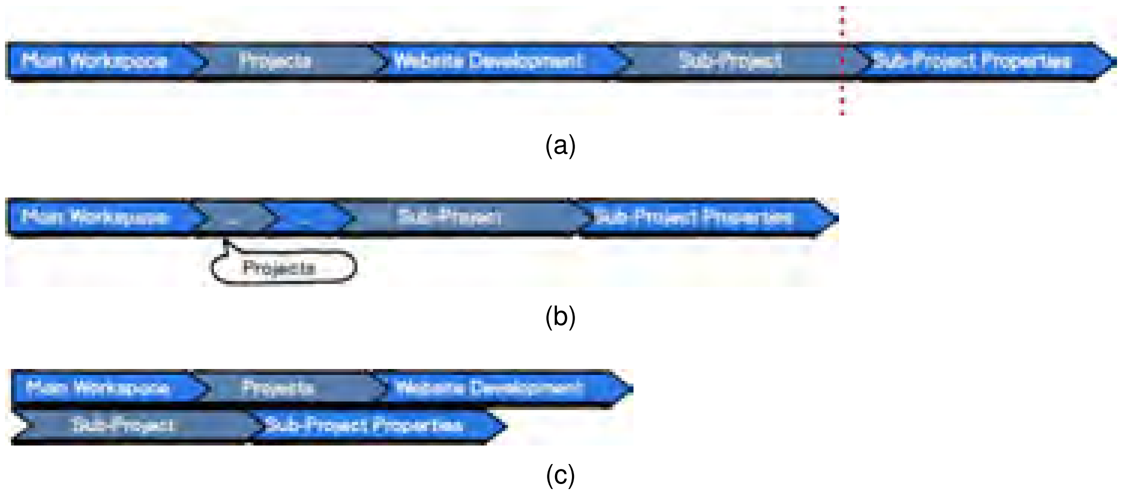


Figure 4.3: Breadcrumbs and Alternative Approaches to Deal with Insufficient Space

### 4.2.1 Breadcrumbs Trail

To provide the user with a clear sense of where he is in the client and to allow him to quickly jump back to previously visited views, a breadcrumbs trail can be used (see Figure 4.3a). However, as more breadcrumbs are added to the breadcrumbs trail, the available space become insufficient (shown as the red dotted line). One way to solve this problem is to replace the labels of the breadcrumbs with ellipsis in order to save space (see Figure 4.3b). If the user hovers over a breadcrumb, with an ellipsis the content will be displayed using a tool-tip providing the information hidden. An alternative way to deal with the problem of insufficient space would be to make it possible for the breadcrumbs trail to span multiple lines (see Figure 4.3c). This yields the advantage that the entries information is always visible. However, to display breadcrumbs spanning multiple lines additional vertical space is needed.

To provide an adaptive navigation (see FR13) breadcrumbs using ellipsis to save space were chosen. This approach was chosen, because it provides a consistent user interface, taking up little space.

The workspace navigation menu (see Figure 4.4) allows knowledge workers to quickly select a process they want to use. Furthermore, it offers access to the process templates and allows for the management of the different roles and organizational models linked to the workspace.

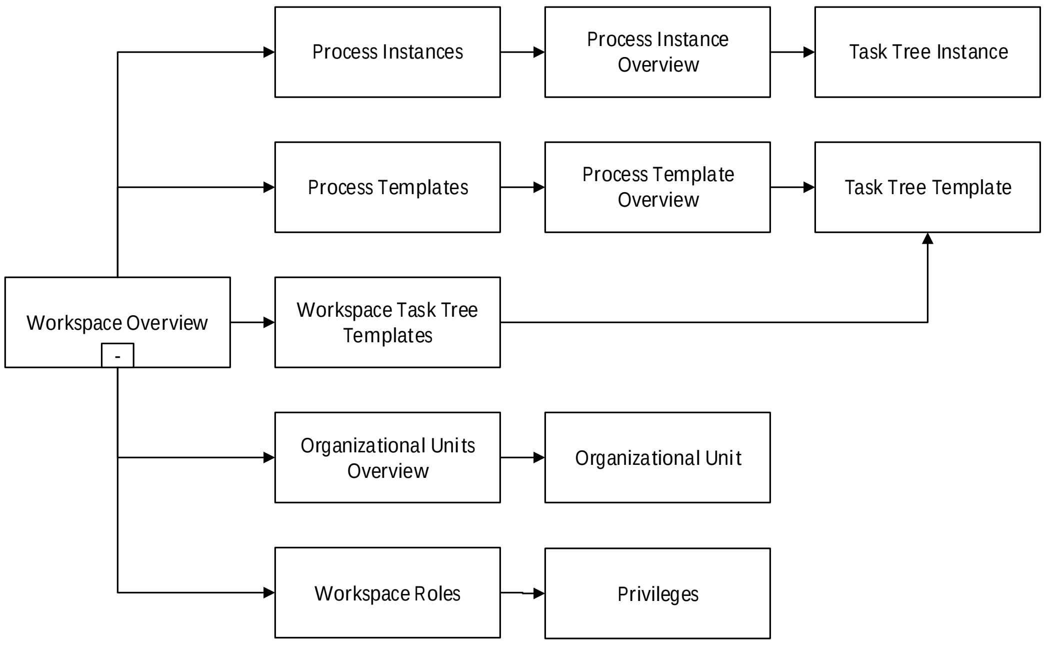


Figure 4.4: Overview of the Views Linked to the Workspace Navigation Menu

### 4.2.2 Process Overview

The process overview will be used for both process instances and process templates. This reuse ensures a uniform look and feel to the client as well as it promotes learnability. The goal of the process overview is to give knowledge workers a quick overview of all the processes linked to the current workspace (see FR1). One way to achieve this is given by a simple table. Tables have the advantage that information can be presented in a very condensed form, allowing a large number of processes to be shown at once (see Figure 4.5).

Alternatively, separate containers could be used for each process. This prevents the application from appearing too cluttered and helps knowledge workers to gain a quick

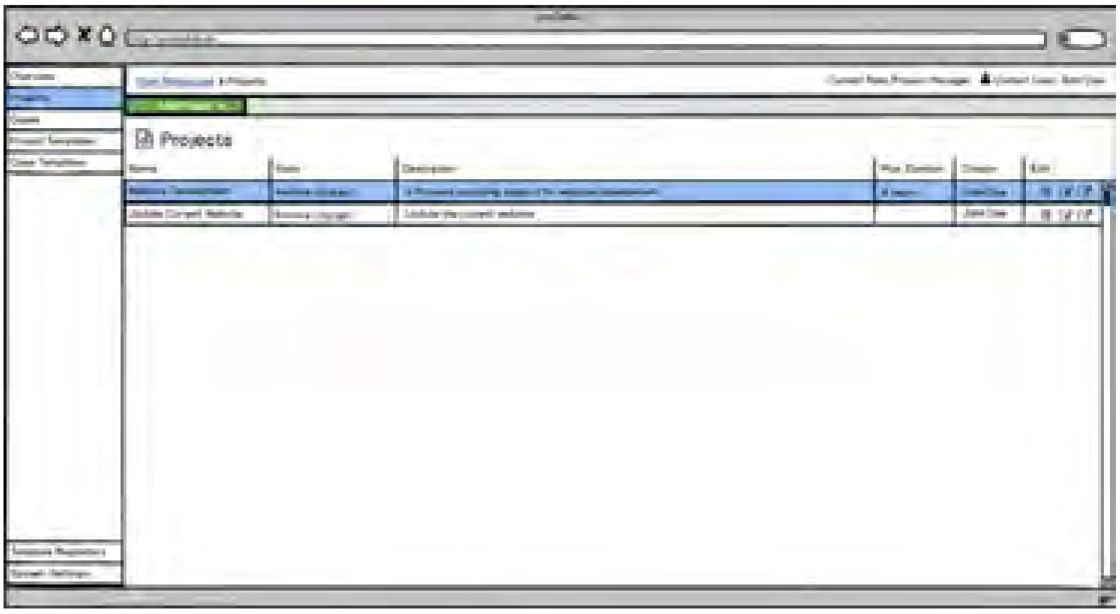


Figure 4.5: Overview of the Different Processes in a Workspace Using a Table

overview. However, this approach needs more space to display the individual processes. Therefore, compared to the first approach, only a fraction of the processes can be displayed at once (see Figure 4.6).

The process overview should also allow knowledge workers to update the different processes and add new ones (see FR1). This can be achieved by enabling inline editing after the update button was pressed (see Figure 4.7). This approach increases learnability by providing a preview of the end result while the user updates the form.

An alternative way would be to open a modal window, which covers the current page and allows the user to update the selected process.

As KiPs can be found in many different domains, the requirements for the user interface may differ from one KiP to another. Therefore, an adaptable user interface is needed. Thus, both approaches for displaying processes should be implemented allowing knowledge workers to switch views as needed. Because this approach makes inline editing impractical, updating processes should be done using a modal window.

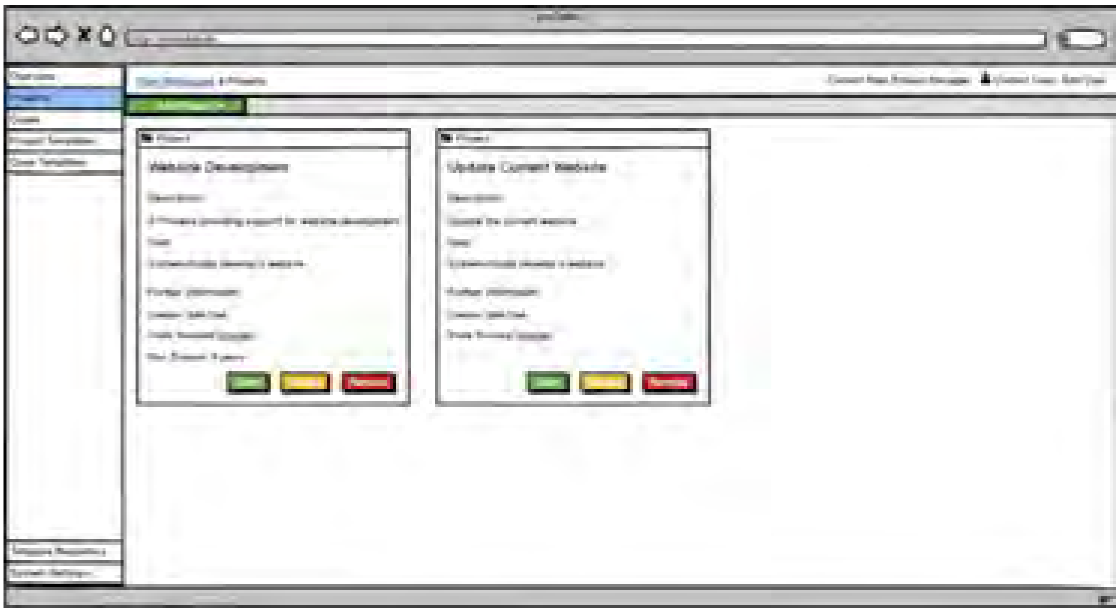
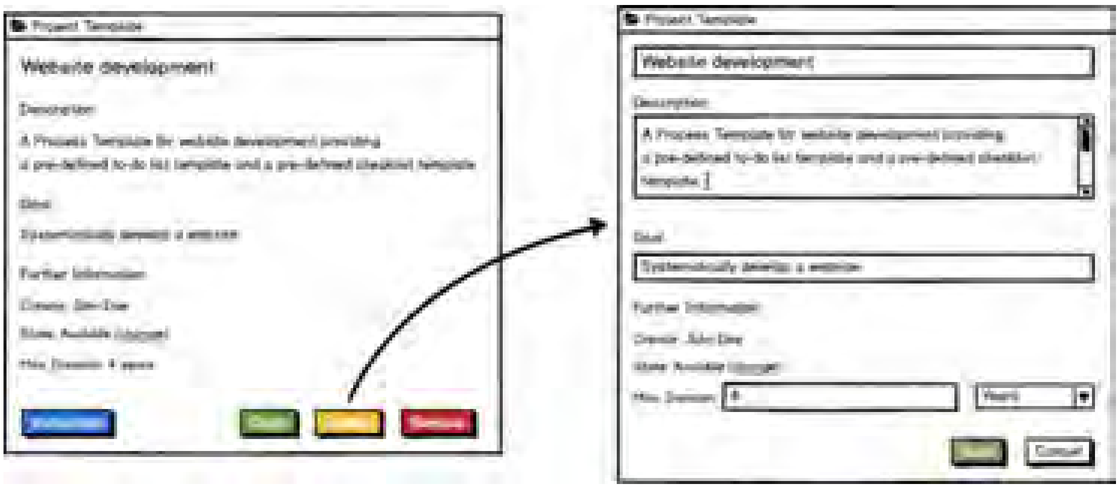


Figure 4.6: Overview of the Different Processes in a Workspace Using Containers


Project Template

Website development

Description

A Process Template for website development providing a pre-defined to-do list template and a pre-defined checklist template

Goal

Systematically develop a website

Further Information

Creator: John Doe

Project Template

Website development

Description

A Process Template for website development providing a pre-defined to-do list template and a pre-defined checklist template

Goal

Systematically develop a website

Further Information

Creator: John Doe

Figure 4.7: Inline Editing of a Process

### 4.2.3 Process Assessment

After a knowledge worker has chosen the process, he wants to work with (see Section 4.2.2), an overview of this process should be presented to him. The overview should contain all available information regarding the process in a compact and clear way (see FR2). In particular, this information should include available sub-processes, the task trees linked to the process, relevant data and documents, and a feed of recent changes. To save space, the different task tree types could be displayed using different tabs (see Figure 4.8).

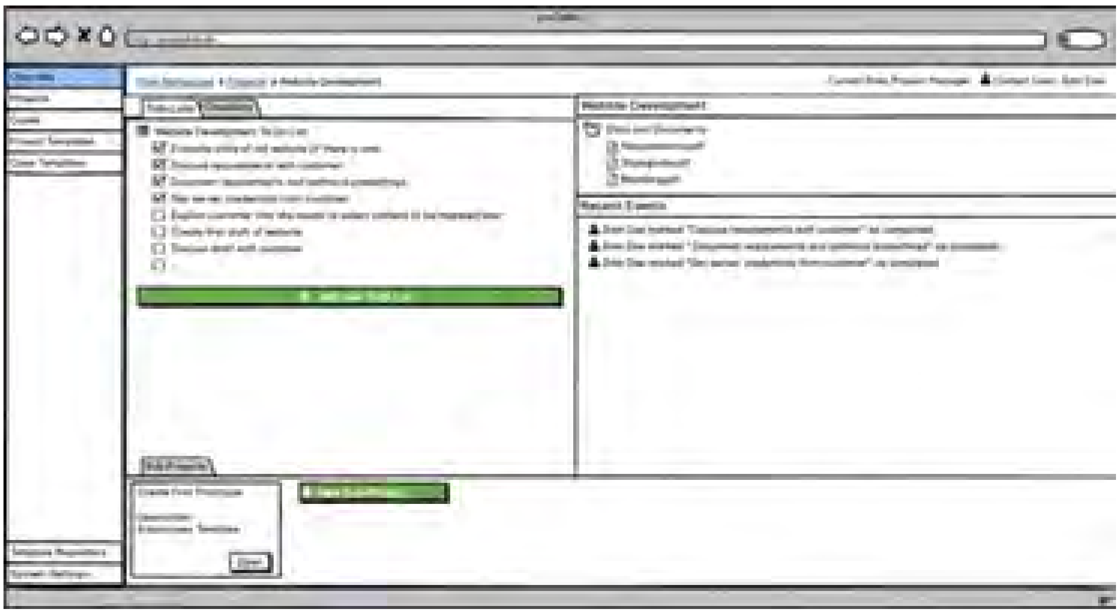


Figure 4.8: One Option for the Process Overview

The proposed approach grants the task trees to take more space and, thereby, makes it possible to display more information regarding the different tasks at once. However, switching between different task tree types could become cumbersome, especially if work has to be done on more than one task tree type at once. For example, if a to-do list is used to expedite a process, while a checklist has to be used to provide quality assurance. In general, this problem could be solved with an approach that presents all the different task tree types at once. Obviously, this means each task tree has to be displayed using less space. Nevertheless, it could provide a clearer overview of the current state with all the different task trees involved (see Figure 4.9). In this approach, prospective task

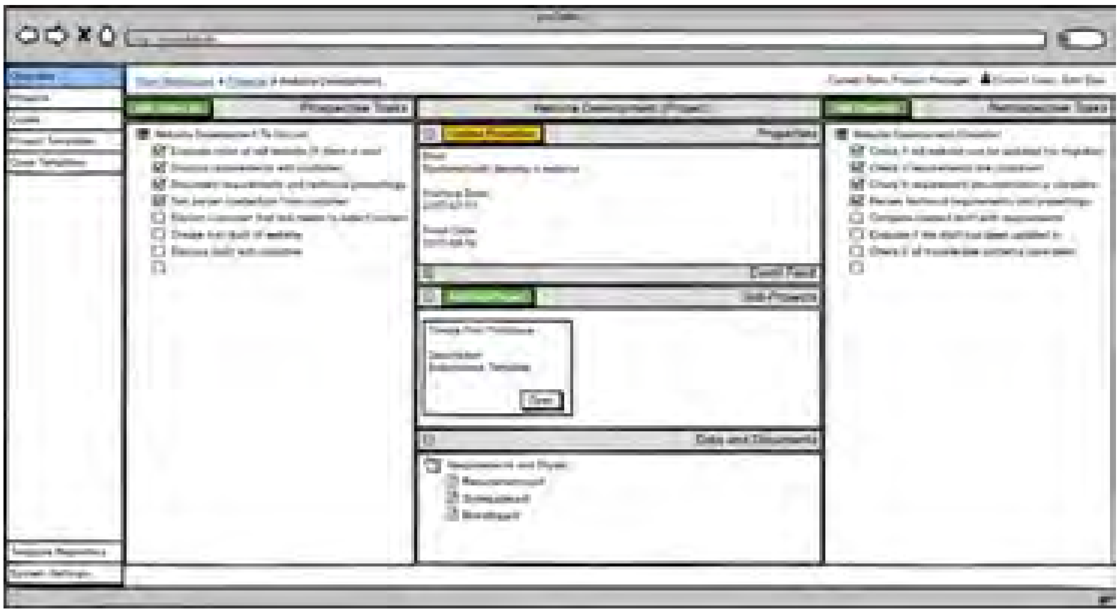


Figure 4.9: Another Option for the Process Overview

trees (e.g. to-do lists) are displayed in the left-hand side column whereas retrospective tasks (e.g. checklists) are displayed in the right-hand side column. The middle column is used to display the other relevant data (e.g., the process properties, an event feed, sub-projects, linked data and documents) using containers with a caption describing the content. A minus or, respectively, plus button next to the caption allows knowledge workers to show or hide the containers content.

In addition, the process overview should provide an easy way to add new task trees and sub-processes to the process. To achieve this, buttons could be added to the bottom of the respective containers (see Figure 4.8). However, placing the buttons at the bottom would lead to the need for scrolling to the bottom of the container, each time a new task tree or sub-process is to be added. To avoid this scrolling, the button could alternatively be placed next to the caption of the respective containers (see Figure 4.9). After clicking one of these buttons, a modal window may be presented to the user, containing a form that allows him to create a new task tree or sub-process, accordingly.

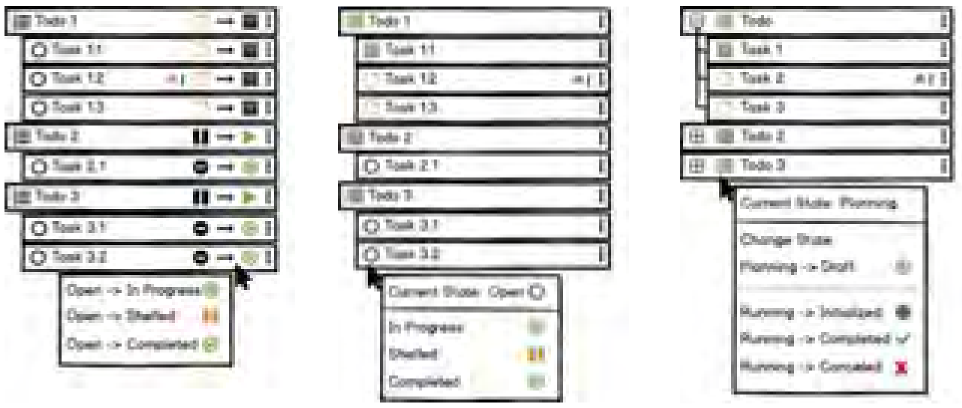
The second approach was chosen for implementation (see Figure 4.9), as it allows knowledge workers to work with multiple task trees of different types concurrently. This approach yields the additional advantage that the buttons to add new sub-processes and task trees are always visible, increasing learnability and efficiency.

### 4.2.4 Management of Task Trees

As shown in Section 4.2.3 *proCollab* heavily relies on the use of task trees in the shape of, e.g., to-do lists, to support knowledge workers. Thus, it is crucial to provide a clean way of displaying all their information, such as their name and current state as well as the subordinate task trees and tasks they comprise. Furthermore, adding new tasks and task trees as well as editing them must be possible to support emergent KiPs and to react to changes (see FR2, FR3).

#### Displaying Task Trees

As task trees can be used to replace the prevalent task tree lists, a hierarchical list is an intuitive choice to display the task trees. Figure 4.10 shows three possibilities that were considered closely in this work. In Figure 4.10a the icons, to the left denotes what type of entity the user is viewing. For example, it is denoted whether the user is viewing a to-do list or a to-do list task. On the right-hand side is a context menu, represented by vertical ellipsis (⋮), which offers functionality, such as updating and deleting task tree elements. Next to the context menu is the state selection that allows the user to quickly change the state of the task tree element. An icon on the left side exposes the current state whereas the icon on the right denotes a following state. By clicking the icon on the right side, a user can switch to the corresponding state. After clicking on the arrow between the two state icons, the user is presented with a drop-down menu, allowing him to select states other than the suggested one. The described approach has the disadvantage that a suggested state is needed for each transition, requiring additional adjustments for each new state model.



(a) (b) (c)

Figure 4.10: Different Task Tree Designs and Approaches for Changing the State of Task Tree Elements

In Figures 4.10b and 4.10c, in turn, only one icon is used to indicate both the entity type and the current state. This approach saves space and makes the user interface less cluttered. Through clicking on the icon, a drop-down menu of all the states, which he may select next, is presented to the user. The state selection drop-down menu in Figure 4.10b is less cluttered than the one shown in Figure 4.10c, but reaches its limits if a refined state (see Section 2.3.6) has to be displayed. Then it is not clear whether the transition is part of the refining state model or not. The drop-down menu shown in Figure 4.10c seeks to eliminate this problem by providing a clearer indication of the current state and its context. Figure 4.10c also introduces the concept of collapsible task trees and tasks to help knowledge workers focus on specific tasks. As a consequence of these considerations, the last approach (see Figure 4.10c) was chosen for implementation. The chosen approach offers knowledge workers the best overview of a task tree element, and does not require the presence of a suggested next state.

### Adding a New Task Tree or Task

Knowledge workers need the ability to easily and quickly add new task tree elements to an existing task tree. Buttons can be used to indicate the position at which the new element shall be inserted (see Figure 4.11). A knowledge worker also has to select what type of element he wants to add e.g., a to-do list or a to-do list item.

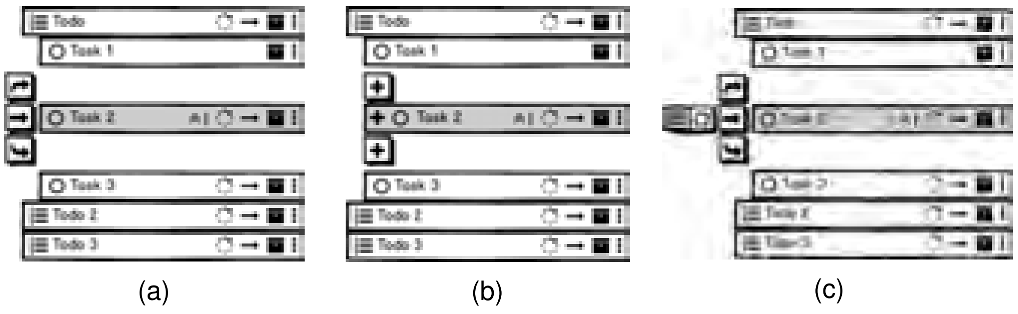


Figure 4.11: Different Approaches to Choosing the New Position of a Task or Task Tree with the Help of Buttons

This could be done by either providing two radio buttons, allowing knowledge workers to select the desired type (see Figure 4.11c) or, as an alternative, in a modal window (see Figure 4.12). Displaying buttons next to the task tree elements has the advantage that a visual preview of where the new element will be added is provided to the user. However, a drawback of this approach is that every time an element is selected the whole task tree has to move to make room to display the relevant buttons. This problem could be solved through the usage of a context menu on the right side. The latter may then also be used, to add new task tree elements (see Figure 4.12). Three vertically stacked radio buttons could be used to represent the different positions at which the new task tree element will be added. In particular, whether it should be added above, below or inserted into the selected task tree element.

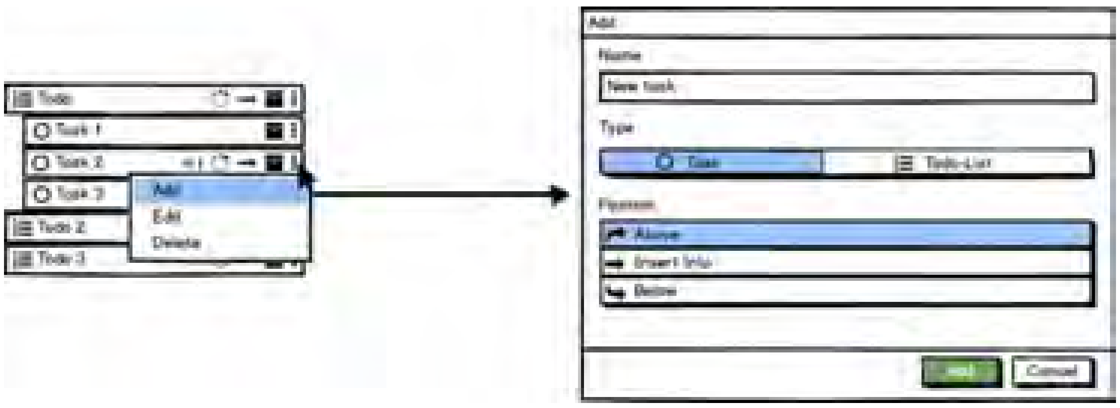
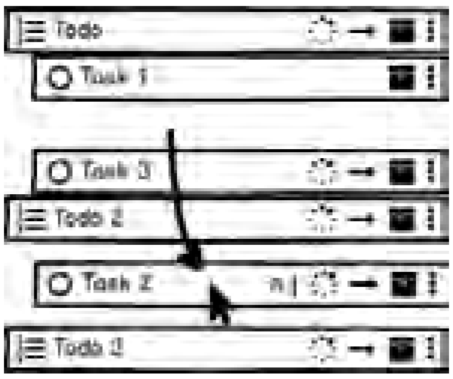
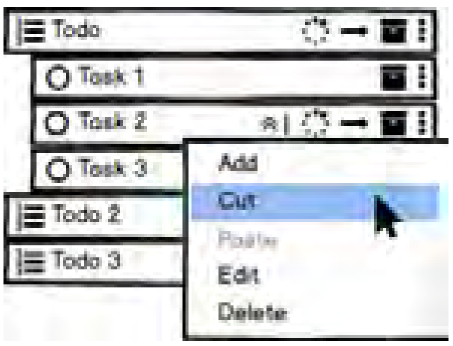


Figure 4.12: Choosing the Position of a New Task Tree Element Using a Modal



(a)



(b)

Figure 4.13: Different Ways to Move a Task

The approach, using a context menu combined with a modal window was chosen to be implemented, as it provides a more uniform user interface, supporting efficiency and learnability.

### Moving Task Trees or Tasks

Knowledge workers should be able to move task tree elements (see FR3). To accomplish this, a drag and drop approach could be used (see Figure 4.13a). The latter has the advantage of instant visual feedback as to where the element is being moved. There is however the drawback that elements could easily be moved by accident. An alternative approach would be to use a context menu to provide cut and paste functionality (see Figure 4.13b). Based on this approach the visual feedback is missing but it is easier to move tasks and task trees between different task trees, since the source and target task tree do not have to be on screen at the same time.

The first approach was chosen for implementation because the visual feedback it provides. However cut and paste functionality would still be useful and may be part of further implementations in the future.

### 4.2.5 Task Tree Configuration

In specific situations (e.g., a case of emergency) only selected parts of a task tree may be needed or additional task tree elements shall be included (see FR8, FR11). The *proCollab* prototype provides the ability to create and manage *task tree configurations* (see Section 2.3.5) to support the configuration of task tree templates before their instantiation.

To enable knowledge workers to add configuration parameters, a corresponding form must be provided allowing them to enter the parameters' name and type. Configuration parameters are then used in an additional form that allows the user to create a new contextual situation (see FR9). In this form, the user may enter a name for the contextual situation and define a regular expression, using the configuration parameters (see Figure 4.14). This regular expression is used when the task tree template is instantiated to determine which contextual situations to use. An intuitive way to display the individual configuration parameters would be a list of check boxes, that showing all available configuration parameters (See Figure 4.15).

A configuration simulation is needed to provide knowledge workers with a preview of a task tree after a contextual situation has been triggered (see FR10). This preview can be provided by showing the additional tasks and task trees which were, created by configuration specifications, grayed out and with a dashed border (see Figure 4.16a). However, this has the disadvantage that the resulting task tree significantly expands and thereby, it may become distracting. An alternative option would be to display contextual situations only if the knowledge worker selected it in the context menu of a task tree (see Figure 4.16b). Furthermore, the name of the currently shown contextual situation is then displayed next to the name of the task tree. Additionally, an icon is added, allowing the knowledge worker to cancel the preview of the contextual situation again. The advantage of this approach would be that contextual situations are only displayed as a knowledge worker explicitly requests it. As a result, space can be saved and the user interface is kept clean. Therefore, the second approach was chosen for the implementation.

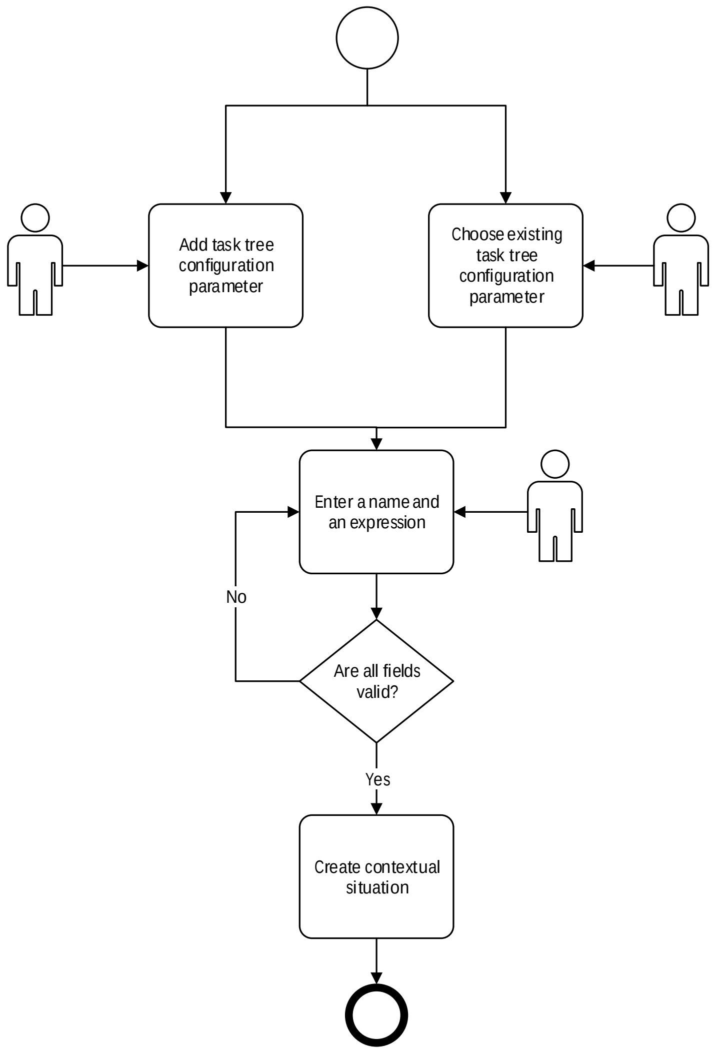
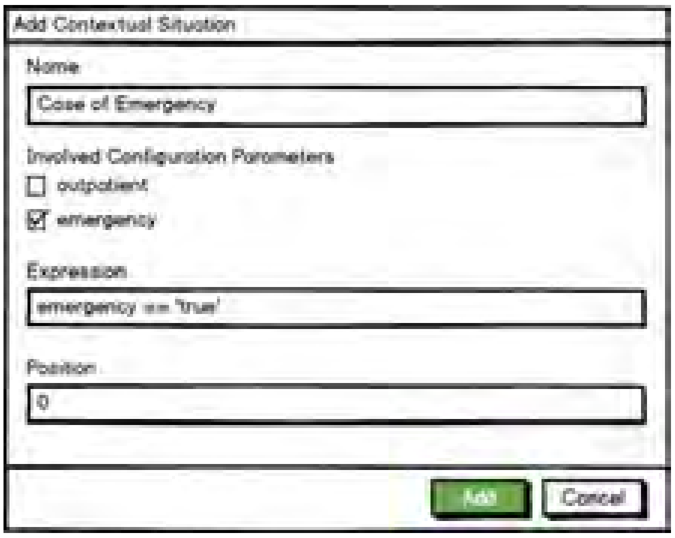

Add task tree configuration parameter
Choose existing task tree configuration parameter
Enter a name and an expression
No
Are all fields valid?
Yes
Create contextual situation

Figure 4.14: Flow Chart for Adding a Contextual Situation



Add Contextual Situation

Name

Case of Emergency

Involved Configuration Parameters

☐ outpatient

☑ emergency

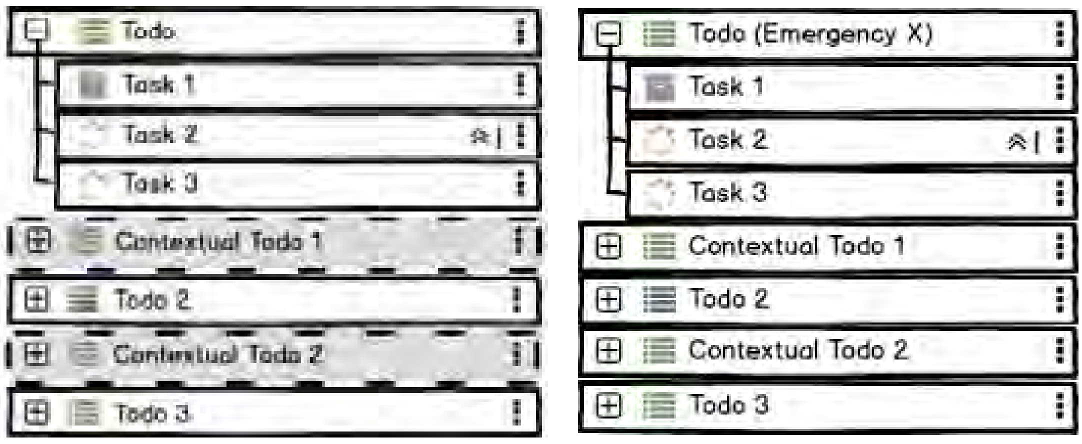
Expression

emergency == 'true'

Position

0

Figure 4.15: Adding a New Contextual Situation



(a) (b)

Figure 4.16: Different Ways of Displaying Contextual Situations

### 4.2.6 Task Tree Overview

A dedicated view is required to, allow knowledge workers to work on a specific task tree (see FR6). A first approach would be to display the selected task tree, together with a container, to allow updating the selected task tree element (see Figure 4.17). As a result, task trees can be updated more quickly.

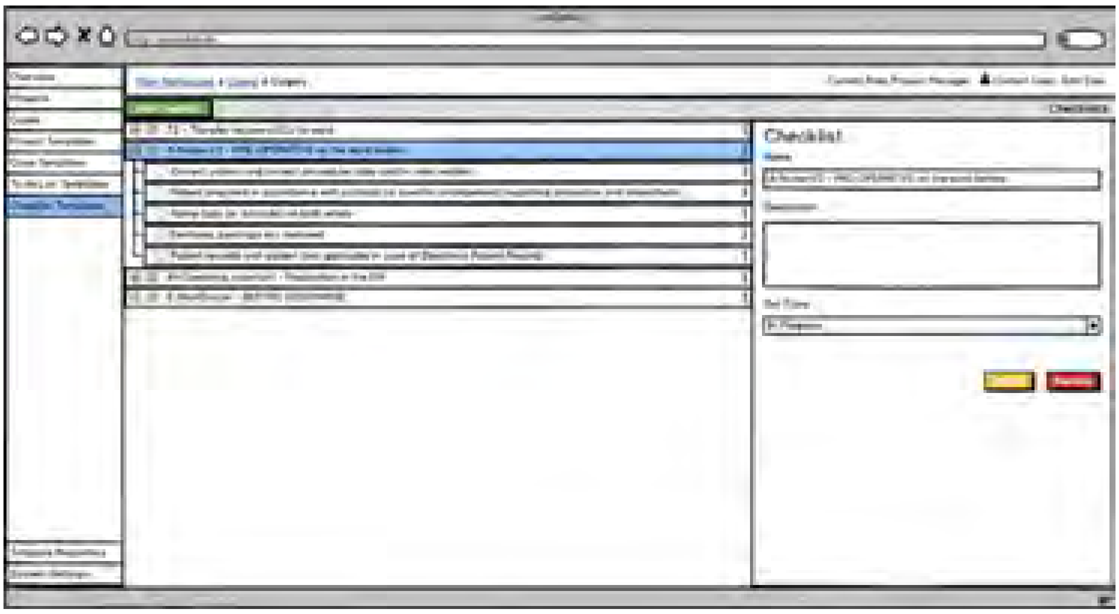


Figure 4.17: One Option for the Task Tree Overview

A second approach would be to display two columns. The first column only displays task trees and the number of tasks they contain. The second column, displays all the task tree elements of the currently selected task tree (see Figure 4.18). This approach allows users to quickly jump from one task tree to another, providing a more efficient user interface. Consequently, the second approach was chosen for the implementation.

### 4.2.7 Management of Organizational Units

Organizational units are, like task trees hierarchical. For example, an organization may be headed by a dean, presiding over multiple departments. These departments, in turn,

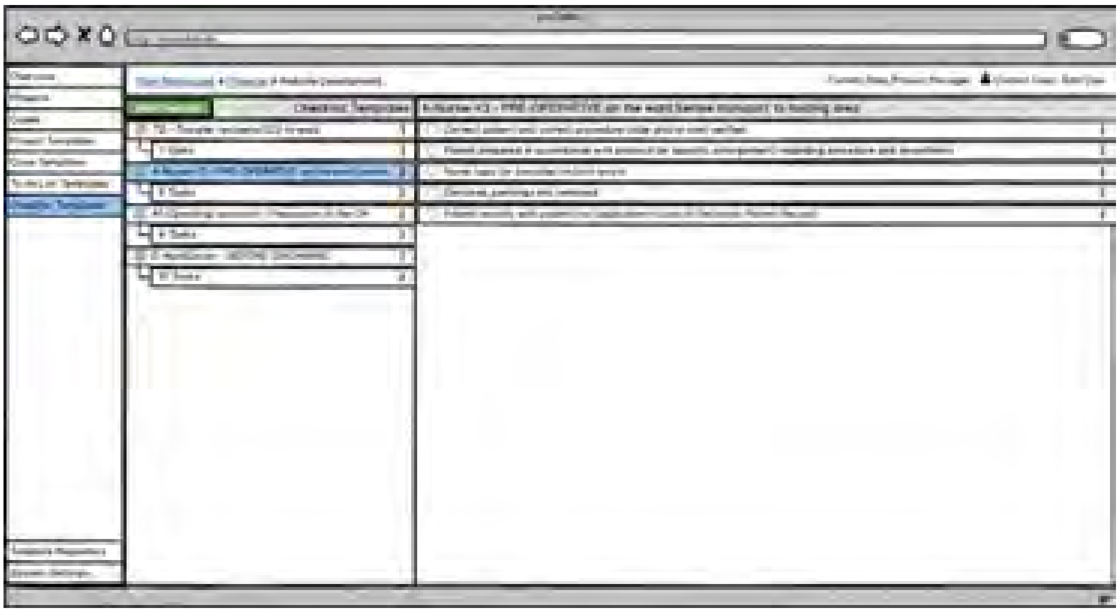


Figure 4.18: Task Tree Overview Using Two Columns

have department heads, presiding over them. As a result, organizational units can be displayed in a similar manner as the task trees discussed in Section 4.2.3 (see Figure 4.19). The reuse of design principles aims to establish a coherent user experience and to improve maintainability. Besides the design, knowledge workers need to be able to add, edit, and delete organizational units as well as organizational roles (see FR26).

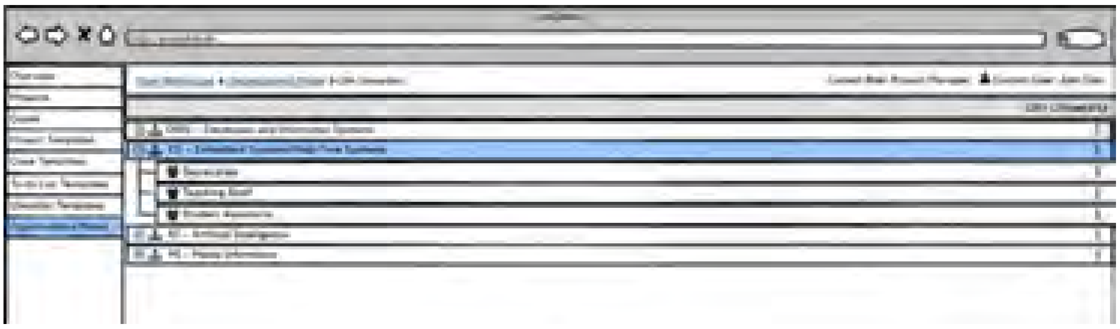


Figure 4.19: Organizational Model Overview

## 4.3 Template Repository

The template repository navigation menu (see Figure 4.20 enables knowledge workers to browse the templates available in the template repository (see Section 2.3.4). The

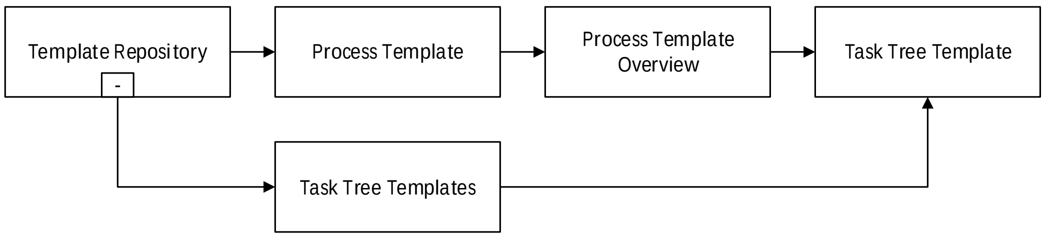


Figure 4.20: Overview of the Views Linked to the Template Repository Navigation Menu

template repository allows sharing of templates across multiple workspaces in order to provide templates that follow best practices (see FR7). It should comprise a process templates overview (see Section 4.2.2), a view to assess process templates (see Section 4.2.3) and a task tree view (see Section 4.2.6). However, the template repository has not been in the focus of this thesis and, therefore, only basic functionality has been added.

## 4.4 System Settings

The system settings menu (see Figure 4.21) enables a user to reach various pages to perform administrative tasks, such as creating and updating state models as well as managing users and user applicants.

### 4.4.1 Management of Roles and Privileges

To take advantage of the ORAC approach discussed in Section 2.3.8, a respective user interface should provide a clear and easy way to add new roles and to link them to the appropriate privileges to pre-selected roles (see FR17, FR18, FR19).

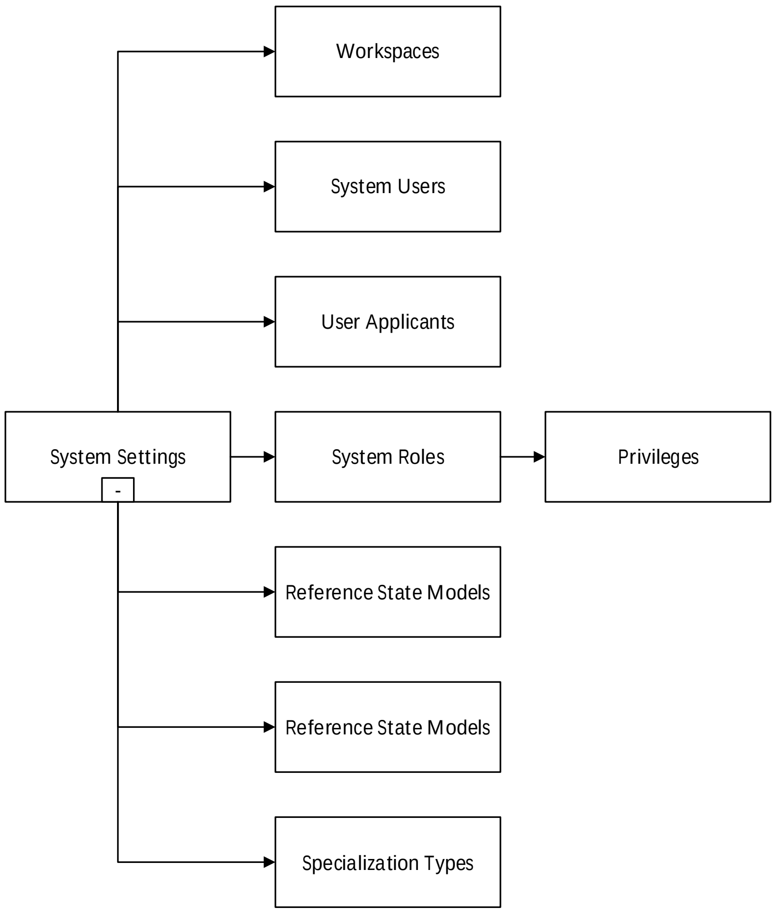


Figure 4.21: Overview of the Views Linked to the System Settings Navigation Menu

Therefore, a user interface is required that lists all of the available roles and allows the creation of new ones. To increase learnability, this user interface can be designed in a similar fashion to the process overview discussed in section 4.2.2. The current *proCollab* web client already supports adding new roles (see Section 3.3). However, it is not possible to assign any privileges to them. As a consequence, a new user interface is needed displaying all the existing privileges in a clear and concise way. Therefore, a user interface with multiple columns was designed, which allows a user to select privileges based on their properties. The properties are in particular defined by the context, target type, action type as well as the name of the action protected by ORAC (see Figure 4.22).

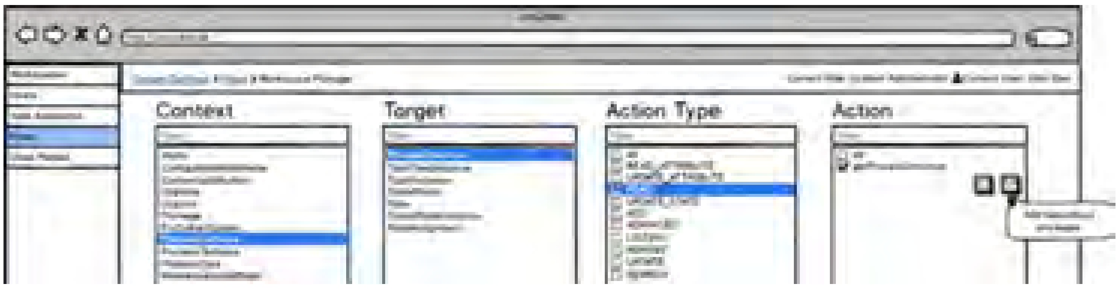


Figure 4.22: Assign Privileges to a Pre-Selected Role

## 4.4.2 User Applicants

Users, who have filled out the registration form to apply for access to *proCollab* are listed as user applicants. To allow administrators to accept or reject the user applicants, a user interface is needed.

One approach would be, to display the user applicants using containers, similar to the ones used for the process overview (see Section 4.2.2). A accept or reject button can be used to accept or reject a user application.

A second approach would be to simply list user applicants in a table with an accept and reject button. This provides system administrators with a simple way to decide whether a user should have access to *proCollab* or not.

As a table needs less space, than a separate container for each individual user application, a higher information density can be provided. Consequently, the second approach was chosen for implementation.

### 4.4.3 State Model Graph

To better visualize the states and transitions of a state model graph, a graphical representation is required. To better denote the properties of a state (e.g., whether it is the initial state or a final state), different styles can be used to display it. As a

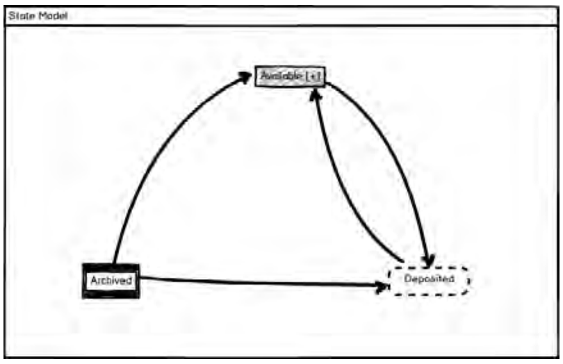
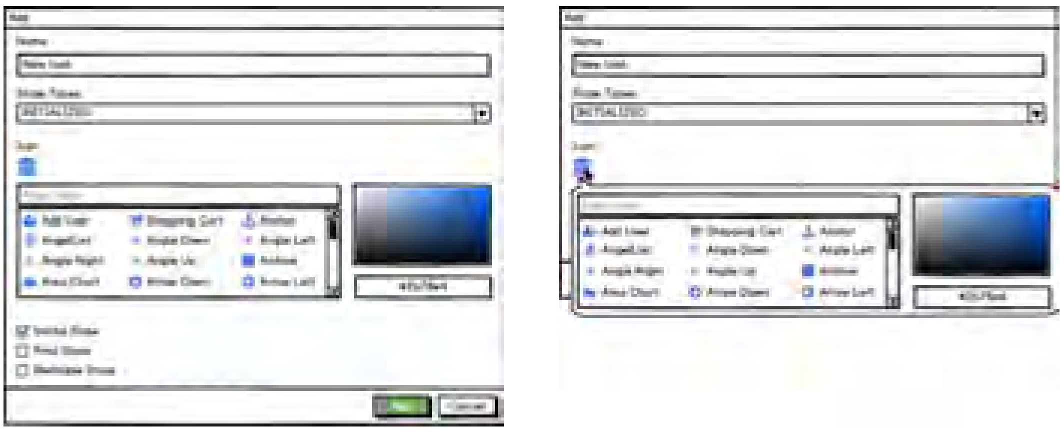


Figure 4.23: Displaying a Reference State Model as Graph

consequence, initial states will be displayed using dashed lines, final states will be displayed using a bold border, and refinable states will be displayed cross-striped. If a refinable state has been refined, a `[+]` behind its name indicates the possibility of viewing the refined state model in a new graph.

#### States

In order to allow knowledge workers to update state models according to their needs (see FR5) corresponding, new functionality in the client is needed. To add a new state to a state model, the user has to provide a name as well as an icon. This could be accomplished by a simple modal window (see Figure 4.24). To help a user to select a suitable icon, an icon picker can be provided giving preview of all available icons. As



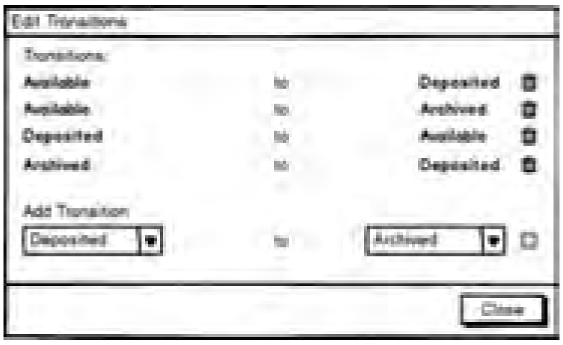
(a) (b)

Figure 4.24: Adding a New State

there are many possible icons, it would help the user to be able to filter them, e.g., using their names. The user should also be enabled to decide in what color the icon should be displayed. Since the color, represented by a hex code, is not immediately obvious, a color picker should allow the user to easily select his desired color. Such an icon and color picker can be displayed either inline (see Figure 4.24a) or in a pop-up after the user clicked on the icon he wants to change (see Figure 4.24b). Displaying it inline has the advantage of being easier to discover. However, the modal window will need more space, to fit all the elements required for the icon and color picker. Nevertheless, an inline icon and color picker was chosen as discoverability is an important factor. Furthermore, as the states are displayed in a graph form, existing states can be deleted through adding a "Delete State" button as soon as the user selects a state. After confirming his action based on a confirmation modal dialog (see Section 4.6.1), the state will be deleted.

### Adding New Transitions

A user needs the ability to add new transitions and delete old ones from a state transition graph. Such functionality is not only needed to connect new states to the graph, but also to edit existing states. A text-based approach, listing all the existing transitions can be used to accomplish this (see Figure 4.25). Next to each transition, a trashcan can be displayed to delete the transition. Two select drop-downs can be utilized to select the source and target of a new transition. After clicking the plus button next to the select drop-downs, the new transition should be added. If a graphical approach was chosen, new transitions could be added by simply drawing a line between two states (see Figure 4.26). Thereby, providing the user with instant feedback where the transition will be inserted. Additionally, the existing states and transitions can be displayed to provide an overview of the state model. Therefore, the graphical approach was selected for the implementation.



| | | |
|---|---|---|
| Available | to | Deposited |
| Available | to | Archived |
| Deposited | to | Available |
| Archived | to | Deposited |

Add Transition: Deposited to Archived

Figure 4.25: Text-Based Approach for Adding and Removing State Transitions

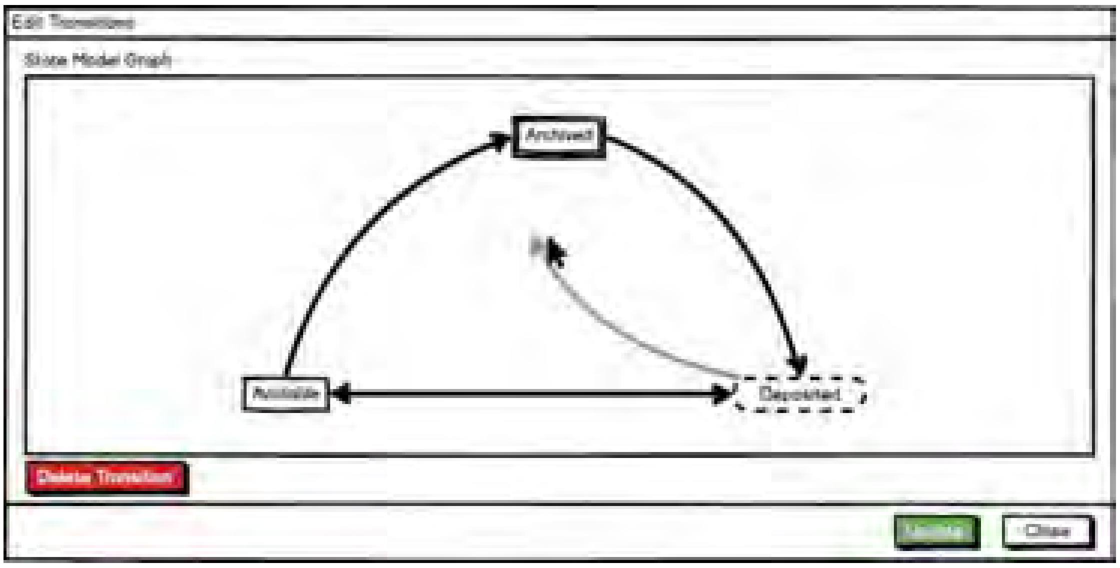


Figure 4.26: Graphical approach for Adding and Removing State Transitions

## 4.5 Account Management

The *proCollab* web client already allows users to log in and update their profiles. However, there is no way for a new user to register himself with the system. A password recovery mechanism should also be provided.

### 4.5.1 Password Recovery

If a user has forgotten his password, he will need an easy way to recover his account, without involving any administrator (see FR24). The *proCollab* server already provides functionality to send a password recovery e-mail to the e-mail address provided by the user. This functionality should be used, through the provision of a simple password recovery form in the web client. This form can be displayed in a modal window, requesting him to enter his e-mail address. Ensuing an e-mail is sent to him containing a link with a unique key. The latter enables a user to create a new password for his account (see Figure 4.27).

### 4.5.2 Registration

To enable users to request an account for *proCollab*, a registration process is needed (see FR22). To register a new account, the user shall be able to enter his data into a form first. After submitting this form, he should be included in the list of user applicants shown in the system settings (see Section 4.4.2). Furthermore, he should be informed that his account will be enabled as soon as his application has been successfully reviewed and approved (see Figure 4.28). After the successful activation of an account, a user will be informed with an e-mail about the activation.

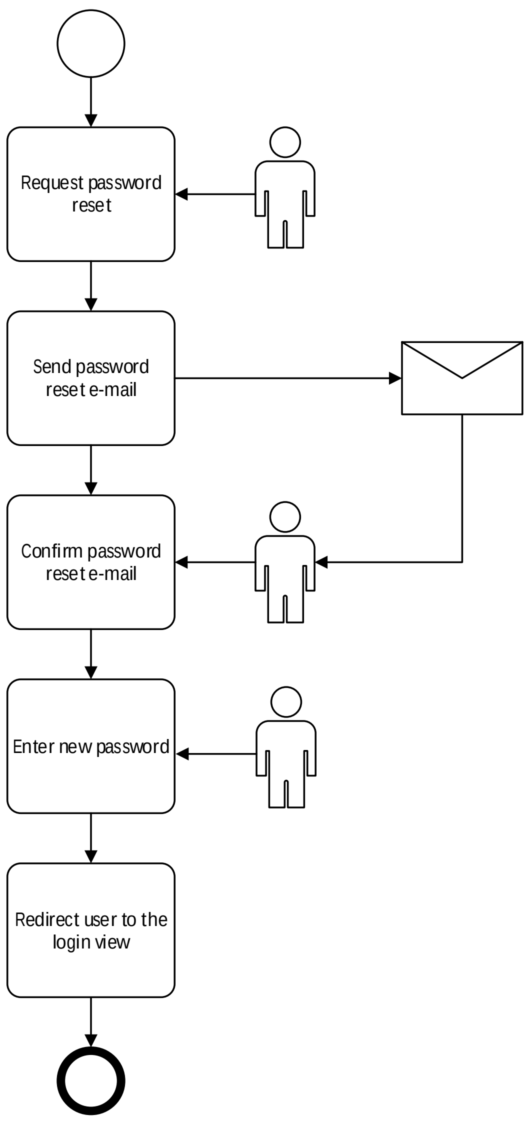

Request password reset
Send password reset e-mail
Confirm password reset e-mail
Enter new password
Redirect user to the login view

Figure 4.27: Flow Chart for the Password Recovery Procedure

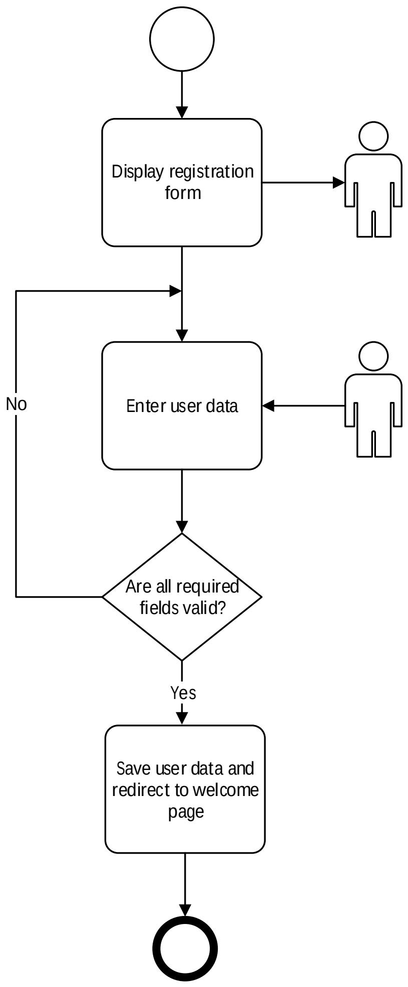

Display registration form
Enter user data
No
Are all required fields valid?
Yes
Save user data and redirect to welcome page

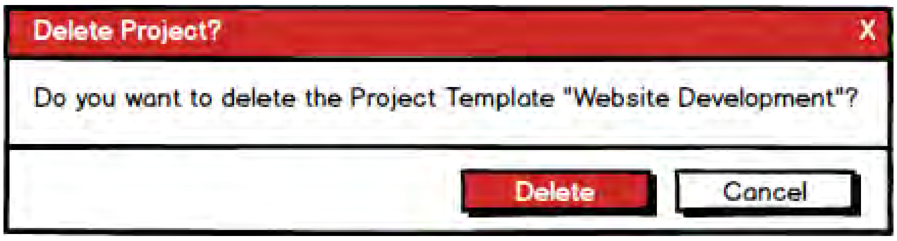
Figure 4.28: Flow Chart for the Registration Procedure

## 4.6 Cross-Cutting Components

To provide a uniform look and feel for the web client, shared components should have a consistent layout. The following sections discuss different approaches to display these cross-cutting components.

### 4.6.1 Confirmation Modal Dialog for Destructive Actions

As most destructive actions cannot easily be undone, an additional confirmation is required to avoid their accidental execution. A possibility is the use of a modal dialog that requests a confirmation, before a destructive action is performed (see Figure 4.29).



Do you want to delete the Project Template "Website Development"?

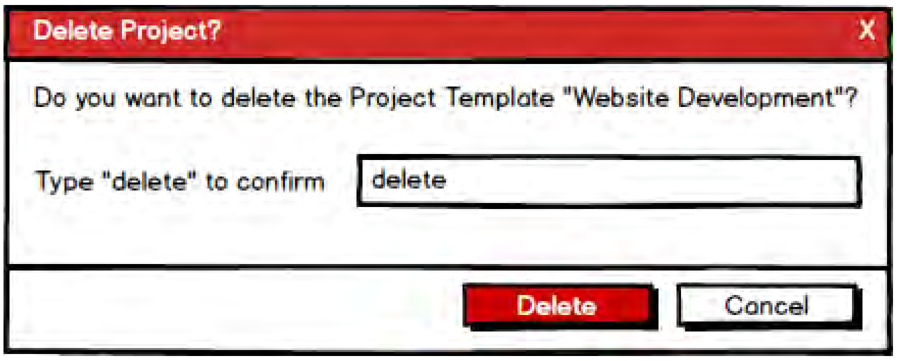
Figure 4.29: Confirmation Modal Dialog for Destructive Actions

A second approach would be, to additionally require the user to enter a string (e.g., "delete") before the destructive action can be performed (see Figure 4.30). As a result, the user needs to perform an additional step, giving him more time to reflect on his action.

The first approach was chosen for the implementation, as it provides sufficient protection from the accidental execution of destructive actions, without causing much inconvenience to the user.

### 4.6.2 Role Selection

An important part for the support of process optimization is to know in which role a knowledge worker wants to perform a specific action. It makes for example, a significant


Delete Project?

Do you want to delete the Project Template "Website Development"?

Type "delete" to confirm: delete

Figure 4.30: Confirmation Modal Dialog for Destructive Actions Requiring Additional Input

difference when a worker changes the state of a task tree as part of his "normal work" or if he changes it as part of an administrative duty. By default, knowledge workers should always operate with the default role for a certain context. For example, the default role linked to the workspace should be selected, when a knowledge worker adds a new process template to the workspace. Only after explicitly switching roles, knowledge workers may perform administrative duties. As a consequence, a place in the user interface is needed where knowledge workers may view the current role and, if needed, switch their current role.

A first approach to address this issue would be to include a radio button for each role (see Figure 4.31) in the navigation bar. This way the knowledge workers are able to quickly perceive all available roles as well as switch to another role. However, this approach becomes impractical, for a large number of available roles. As an alternative, a role

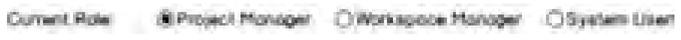

Current Role [✓ Project Manager] Workspace Manager System User

Figure 4.31: Displaying the Available Role Using Radio Buttons

selection drop-down menu could be included in the navigation bar. Another approach would be to include the role select drop-down menu in the navigation bar. This approach enables knowledge workers to quickly perceive, in which role they are currently acting. Furthermore, they may switch quickly between different roles without leaving the current page. The second approach, can display a large number of roles, without problems, while also providing a clean user interface. Therefore, the second approach was selected for implementation.

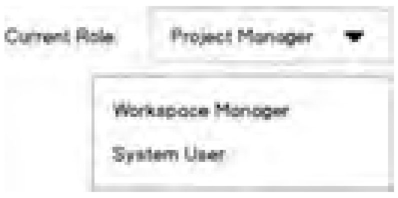


Figure 4.32

### 4.6.3 Caching

The introduction and usage of caching causes additional complexity in the architecture and logic of an application, but it also provides a much smoother and faster user experience. In general, some applications omit caching to avoid complexity and increase maintainability. However, this has the drawback that calls to the servers API may take a comparable long time if the user is accessing the application, while using a slow internet connection. A first simple solution is to cache the result of each API call in a map, using the called function and its parameters as key. The cache can be marked as stale after a pre-defined time period and the next call of the function will consequently refresh the cache. This solution has the advantage that it is easy to implement and does not add much complexity. A more sophisticated approach would be to cache the different entities using their unique ids. This adds additional complexity, but enhances the performance as only changed entities are updated. A big disadvantage, however, is that the user may see stale data, if the data on the server has changed since his cache was updated. To address this problem as well, a WebSocket can be used to inform the client of changes on the server (). This information can then be used to mark the pertaining entities in the cache as stale and to get the latest version from the server. Despite any additional complexity and implementation effort, this approach was chosen to guarantee fast and smooth user experience.

# 5 Implementation

This chapter presents selected aspects from the implementation of the previously discussed concepts in Chapter 4. In Section 5.1, the technologies used by the *proCollab* web client are described to provide a general overview. Section 5.2 discusses selected excerpts from the implementation.

## 5.1 Technologies

The *proCollab* web client was developed, based on the Angular 4 framework (see Section 5.1.4) and the typescript programming language (see Section 5.1.6). The *proCollab* web client communicates with the *proCollab* server on the base of a Representational State Transfer (REST) interface. Furthermore, a WebSocket connection is used to receive notifications from the server.

### 5.1.1 Representational State Transfer (REST)

The REST programming paradigm is used for the communication between client and server in distributed systems [Fie00]. Methods of the Hypertext Transfer Protocol (HTTP) (e.g., GET and POST) are used by the client to manipulate resources on the server side. REST supports create, read, update, and delete (CRUD) operations. An important advantage of REST is that it provides a lightweight solution compared to competing standards such as Simple Object Access Protocol (SOAP).

### 5.1.2 WebSocket

The WebSocket protocol provides bidirectional communication between a client and a server [FM11]. To establish such a connection, only a single TCP connection is needed, this avoids the overhead of other solutions like HTTP polling [FM11]. WebSockets are supported by all state of-the-art browsers.

### 5.1.3 JavaScript Object Notation

JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It can represent four primitive types (strings, numbers, Booleans, and null) and two structured types (objects and arrays) [Cro06]. JSON is human readable and produces, in comparison to alternative data interchange formats such as XML, a low overhead.

### 5.1.4 Angular 4

Angular 4 is a framework for building client applications in HTML and a programming language, compiled to JavaScript [Ang17a]. Its aim is to provide scalable web applications that offer good performance. Figure 5.1 shows an overview of the different components Angular 4 comprises. Angular 4 applications are written by composing HTML templates with Angular-specific markup. HTML templates are managed by *component classes*. The application logic for the component classes is provided by services. Services and components are organized, in modules. In the following the Angular 4 architecture is discussed in more detail.

#### Angular 4 Architecture

Angular 4 uses modules to organize an application into cohesive blocks of functionality [Ang17b]. Components are used to control the different views through an API of properties and methods. Angular 4 creates, updates and destroys components as

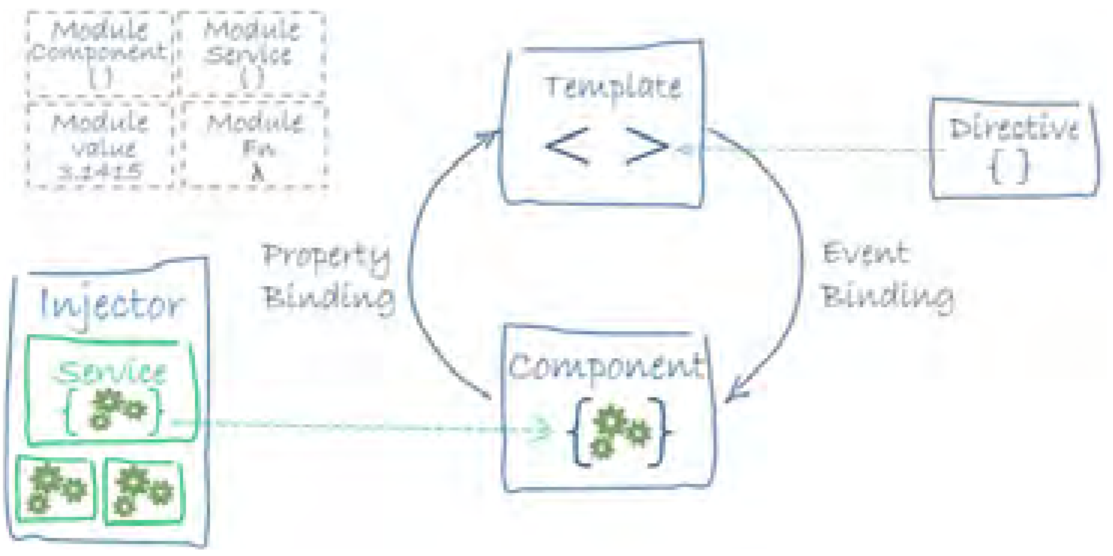


Figure 5.1: Angular 4 Overview [Ang17a]

the user moves through the application. The component's view is defined using its *companion template*. Templates are used by Angular to render the component. The templates use regular HTML syntax with some Angular specific additions. These additions are used to transform the Document Object Model (DOM) according to instructions, provided by directives. Finally, the application logic is realized by services. The services category encompasses any value, function or feature that an application needs. This modularity greatly improves the maintainability of Angular 4 applications because it provides a clear separation of the views and the application logic.

### 5.1.5 RxJS

Angular 4 includes RxJS, which enables the usage of asynchronous data streams. In RxJS, asynchronous data streams are represented using observables. Observers can subscribe to these observables and react to an item or a sequence of items emitted by the Observable [Rea17].

### 5.1.6 Typescript

Despite its prevalence, JavaScript remains a rather poor language for developing and maintaining large applications. This is partly caused by the untyped nature of JavaScritpt. The primary goal of TypeScript is to give a statically typed experience to JavaScript development improving development and maintainability [BAT14]. Statically typed languages, for example, allow a large number of errors to be detected, early in the development process. In particular, state-of-the-art development tools can check typescript code for type mismatches.

### 5.1.7 SASS

Cascading Style Sheets (CSS) are the prevalent way to style web-based applications. Sass is an extension of CSS that adds useful features to CSS while maintaining full CSS-compatible syntax [Sas17]. It allows, for example, for the usage of variables, nested rules, and inline imports. The ability to use variables makes it possible to define the color scheme for the *proCollab* web client (see Figure 5.2) centrally thereby, a uniform look and feel of the application can be well maintained.

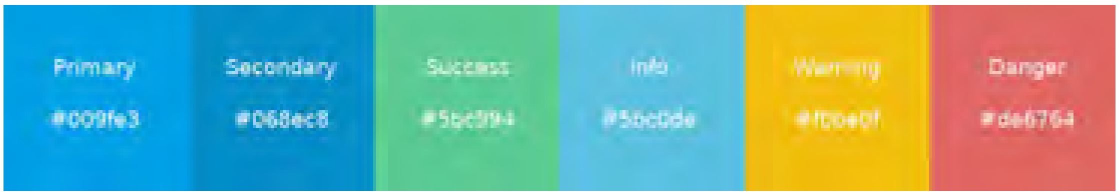


Figure 5.2: Color Scheme

### 5.1.8 Cytoscape.js

Cytosacpe.js provides a fully featured graph library written in pure JavaScript [Cyt17]. Cytoscape.js provides layouts to automatically place the nodes of graphs. Furthermore, it supports stylesheets to separate presentation from data. It is highly optimized and compatible with all modern browsers.

## 5.2 Implementation Excerpts

This section discusses selected excerpts of the implementation of the concepts introduced in Chapter 4. For illustrative purposes, screenshots and code excerpts are used to underline important aspects of the implementation.

### 5.2.1 Deployment Configurability

Variables, which are used to configure the *proCollab* web client, are stored in configuration files. The latter make it easier to change and maintain these variables. Overall, there are multiple configuration files: one for the base configuration and additional ones to override variables set in the base configuration. When deploying a new instance of the *proCollab* web client, parameters can be provided to select one of the additional configuration files. This enables the deployment of the application without the need to make any changes i.e., a client deployed on a server may use a different server address than a client running locally for development purposes. The base variables are defined in the `base.ts` file shown in Listing 5.1.

```
const BaseConfig: EnvConfig = {
  API: 'http://localhost:8080/procollab/rest/',
  SOCKET: 'ws://localhost:8080/procollab/websocket/',
  DATE_FORMAT: 'short', // MM/dd/yyyy HH:mm
  ICONS: {
    WORKSPACE: 'fa-th-large',
    ...
    CONFIRM: 'fa-check',
    PRIORITY: {
      HIGH: 'fa-angle-double-up',
      MEDIUM: '',
      LOW: 'fa-angle-down',
    },
    TASK_DUE: 'fa-clock-o'
  },
  // Time until Task is due (SOON, NOW, OVERDUE) in milliseconds
  TASK_DUE: {
      SOON: 259200000,
      NOW: 86400000,
      OVERDUE: 0
    }
};
```

Listing 5.1: Base Variables of the Client Configuration

The `base.ts` file provides the addresses of the REST API and the WebSocket. Furthermore, it allows configuring different aspects of the *proCollab* web client, such as the format used for displaying dates and the different icons used throughout the client. Additionally, it is used to determine the intervals any icons are displayed, before a task reaches its due date.

If the client is built for deployment, the `prod.ts` configuration file (see Listing 5.2) is used to override variables defined in the `base.ts` file to support the servers' environment for example, the addresses of the REST API and WebSocket have to be changed.

```
const ProdConfig: EnvConfig = {
  API: 'https://dev.procollab.dbis.info/api/',
  SOCKET: 'wss://dev.procollab.dbis.info/api/websocket/'
};
```

Listing 5.2: Adjusting the Variables for Deployment on a Server

### 5.2.2 Caching and Updating

To provide a good workflow and increase user acceptance, it is important that the user is always aware of latest data from the server without long loading times (see Section 4.6.3). To achieve this, services for storing and updating objects from the server have been implemented. Any component may use these services to get data from the server. The services, in turn, return RxJS (see Section 5.1.5) observable objects, to which the components may then subscribe. Once the call to the server has been completed, or if the result of the call has been already cached, the observable emits the requested data and the components may process it. To avoid stale data in the cache, a WebSocket provided by the server is used. This WebSocket informs the web clients of changes to objects, through providing their id, information about the kind of change (e.g., whether the object has been deleted or updated) and contextual entities of the object (e.g., the process to which a task tree is linked to). The cache service then checks whether the object is used in any previously created view (i.e., whether its parent object has already been cached). If that is the case, the service requests the object from the server and updates the corresponding observable. After the update, all the components that are subscribed to the observable responsible for the object are informed about the change and may react accordingly.

The caching implementation makes collaboration easier because all knowledge workers may instantly see changes to objects, which they are working with. Thereby, the problems associated with a stale cache can be avoided.

Listing 5.3 shows an example of data emitted by the WebSocket after the state of a task tree instance has been changed. This data includes the information that the state model instance with the id 942 has been updated. Furthermore, it contains information pertaining the context of this update. According to this data, the task tree instance with the id 842, which is part of a process instance with the id 840 in the workspace with the id 803, has to be updated. Based on this information, the caching service makes the appropriate REST calls to update the task tree instance. Subsequently, all subscribers are provided with the latest version of the task tree instance, which contains the updated state model instance.

```
guardedEntityClazz:
  "info.dbis.procollab.system.datamodel.statemgmt.StateModelInstance",
guardedEntityId: 942,
persistenceLogEventType: "UPDATED",
contextualGuardedEntities: [
0: [
  {
    guardedEntityId: 842,
    guardedEntityClazz:
      "class info.dbis.procollab.system.datamodel.TaskTreeInstance"
  },
  {
    guardedEntityId: 840,
    guardedEntityClazz:
      "class info.dbis.procollab.system.datamodel.ProcessInstance"
  },
  {
    guardedEntityId: 803,
    guardedEntityClazz:
      "class info.dbis.procollab.system.datamodel.Workspace"
  },
  {
    guardedEntityId: 102,
    guardedEntityClazz:
      "class info.dbis.procollab.system.datamodel.ProCollabSystem"
  }
 ]
]
```

Listing 5.3: An Example of Data Emitted by the WebSocket

Listing 5.4 shows the function that is used to add or update a task tree element in the cache, using the data provided by a REST call. This function uses a *ReplaySubject* as observable. The function takes one variable called `item`. This variable contains the updated object that should be added to the cache. The `statefulEntityObservables` object contains all the observables used for caching stateful entities. The ids of the individual entities are used as keys. In Line 2, it is first checked whether an observable for this id is already defined. If that is not the case, a new observable ReplaySubject is created for this id. Subsequently, the obsevable emits the updated object using its `next()` function. In doing so, all the observers are provided with the updated object.

```
1  set(item:TaskTreeTemplate | TaskTreeInstance | TaskInstance | TaskTemplate) {
2    if(!this.statefulEntityObservables[item.id]) {
3      this.statefulEntityObservables[item.id] = new ReplaySubject(1);
4    }
5    this.statefulEntityObservables[item.id].next(item);
6    }
7  }
```

Listing 5.4: Add an Item in the Cache

To observe an observable provided by the cache, its `subscribe()` function is used. Listing 5.5 shows a simplified example for this, where the observable emits a task instance.

```
taskObservable.subscribe((task: TaskInstance) => {
  doSomething(task);
}
```

Listing 5.5: Observing an Observable

Each time the `taskObservable` emits a new value using its `next()` function, the `doSomething()` function is called with the new task object as argument. This can be used, for example, to keep the view up to date.

### 5.2.3 Navigation Structure

As discussed in Section 4.1, a dynamic sidebar approach combined with a breadcrumbs trail was selected as preferred navigation structure. As a result, the user can quickly perceive where he is looking at in the *proCollab* web client any given time. The different views of the sidebar can be seen in Figure 5.3 and follow the structure proposed in Section 4.1. To provide this functionality, a location service was implemented generating

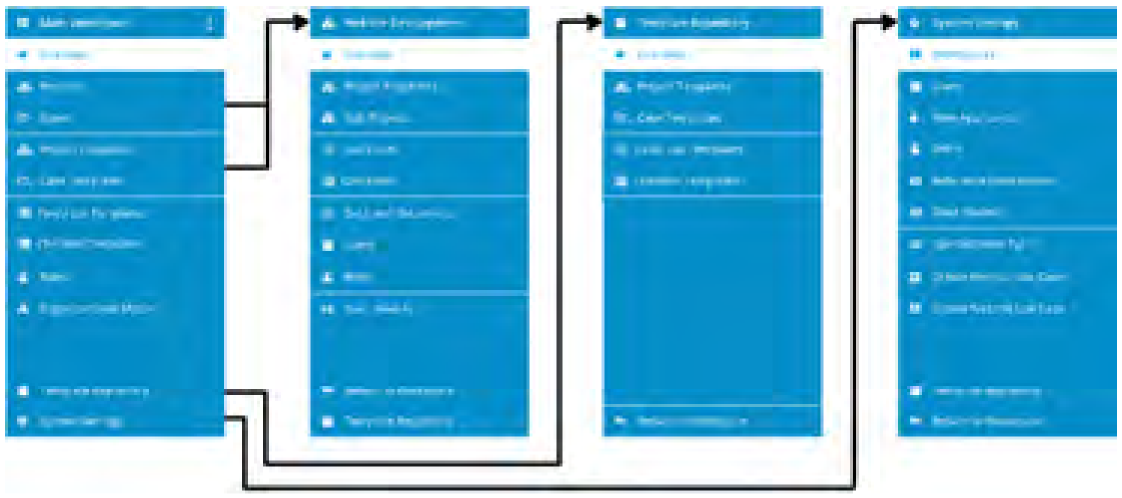


Figure 5.3: Overview of the Different Navigation Menus

the sidebar and breadcrumbs trail according to the parameters provided by the URL.

### Breadcrumbs Trail

Figure 5.4 shows an example of a breadcrumbs trail used in *proCollab*. This example uses the URL:

```
.../workspace/803/templates/processType/202/template/805;process=804
```

This URL can be read as:

- Workspace: 803 (Main Workspace)
- Process type: 202 (Project Template)
- Process template: 805 (Requirements Engineering)
- Process template: 804 (Website Development)

It has to be noted that the parent processes and parent task trees are added to the end of the URL using matrix parameters. If there is more than one parent, a - (dash) will be used as delimiter.

Based on this information, a breadcrumbs trail can be built enabling the user to quickly jump to any of the views he has likely visited previously. He can also easily share the URL with his co-workers and they will see the same breadcrumb trail, providing a quick overview and all parental views.

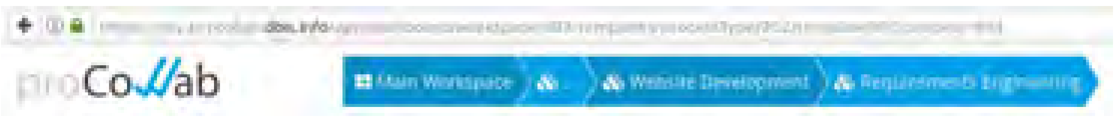


Figure 5.4: Breadcrumbs Trail

### Adding Ellipsis to the Breadcrumbs Trail

The breadcrumbs trail seen in Figure 5.4 shows the usage of ellipsis to save space. Listing 5.6 shows the code to determine how many ellipses have to be used in the breadcrumbs trail.

```
 1  // calculate space left for breadcrumbs
 2  let crumbsContainer = all - navLeft - navRight;
 3  // get size of each breadcrumb
 4  let crumbsSizes:number[] = [];
 5  for (let i = 0; i < document.querySelectorAll('.crumb').length; i++) {
 6    crumbsSizes.push(document.querySelectorAll('.crumb')[i].clientWidth);
 7  }
 8  // size of all breadcrumbs
 9  let allCrumbsSize = crumbsSizes.reduce((a, b) => a + b, 0);
10
11  // add ellipsis if needed
12  if(crumbsContainer < allCrumbsSize) {
13    for (let i = 0; i < crumbsSizes.length
14      && crumbsContainer < allCrumbsSize; i++) {
15      this.ellipsisIndex++;
16      allCrumbsSize = (allCrumbsSize - crumbsSizes[i]) + 55;
17    }
18  }
```

Listing 5.6: Adding Ellipsis to the Breadcrumbs Trail

First, the space available for the breadcrumbs trail is calculated. Subsequently, the for-loop in Line 5 adds the individual sizes of all the breadcrumbs to an array. Next, all the values of this array are accumulated, to determine the size of all breadcrumbs in total. If this size is smaller than the available space, no special procedure is needed. If the space is insufficient, a for-loop is executed. The size of a breadcrumb containing an icon and ellipsis is 55px. In the for-loop, this value is used to calculate how many ellipses have to be added to fit the breadcrumbs trail into the available space. For this purpose, the length of the breadcrumb at the current index is subtracted from the combined size of the breadcrumbs trail and 55px are added. Additionally, the ellipsisIndex is incremented. The ellipsisIndex is then used in the template to render the breadcrumbs trail with the corresponding number of ellipsis.

### 5.2.4 Confirmation Modal Dialog for Destructive Actions

Angular 4 fosters the reuse of a component in different views (see Section 5.1.4). An example for such a reusable component is the confirmation modal dialog, as it is used in various contexts to make sure a user really wants to perform a given action (e.g., deleting a process instance). For better maintainability and a uniform appearance, this component is used for all confirmation modal dialogs. Listing 5.7 shows how the confirmation modal dialog can be applied to any view.

```
...

<confirmation-modal *ngIf="showConfirmationModal" [modalData]="modalData"
   (modalClosed)="modalClosed($event)"></confirmation-modal>

...
```

Listing 5.7: Integration of a Confirmation Modal Dialog Into a View

In this example, the showConfirmationModal variable is just a Boolean that is set to true when the modal dialog should be displayed. Further, modalClosed is an event emitter that triggers the modalClosed function of the confirmation modal component as soon as the modal dialog shall be closed. The modalClosed function then sets showConfirmationModal to false, thus hiding the confirmation modal dialog again. Finally, modalData comprises the actual data used by the modal component. This data is provided using a model built for this purpose. The different variables, which this model contains, are shown in Listing 5.8.

```
/**
*  color: 'success' | 'warning' | 'danger' | 'info' | 'primary' | 'default';
*    The color of the modal header and confirmation button
*  title:string; Title of the modal
*  text:string; Text of the modal
*  cancelBtn:string; Text of the cancel button
*  confirmBtn:string; Text of the confirmation button
*  fnc:Funtion; function to be called if confirmBtn is clicked
*/

export class ConfirmationModalData {
  color: 'success' | 'warning' | 'danger' | 'info' | 'primary' | 'default';
  title:string;
  text:string;
  cancelBtn:string;
  confirmBtn:string;
  fnc:Function;

  ...
```

Listing 5.8: Model of the Confirmation Modal Dialog

The `color` variable controls the color of the modal header and of the confirmation button. Using typescript (see Section 5.1.6), it can be ensured that only valid strings are accepted (i.e., success, warning, danger, info, primary or default). The `title` contains a string used as the title of the modal dialog. Furthermore, `text` contains the text, which the modal dialog shall display. Next, `cancelBtn` and `confirmBtn` contain the strings used as label for the cancel and confirm button, respectively. Finally, `fnc` contains the function that shall be called after the user clicks the accept button. Figure 5.5 shows an example of the confirmation modal dialog asking the user whether he really wants to delete a process instance.

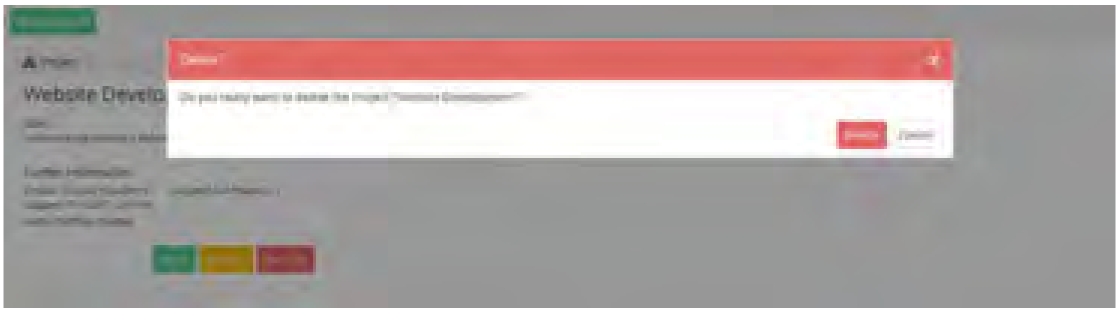


Figure 5.5: A Confirmation Modal Dialog for Destructive or Important Actions

### 5.2.5 Process Overview

To give the user an overview of all the PTs and PIs available to him, a process overview was implemented. As discussed in Section 4.2.2, two different views have been implemented. One relying on containers (see Figure 5.6) and a second one using a table to provide an overview of the processes (see Figure 5.7). The view can be switched by clicking the button in the bottom right corner.

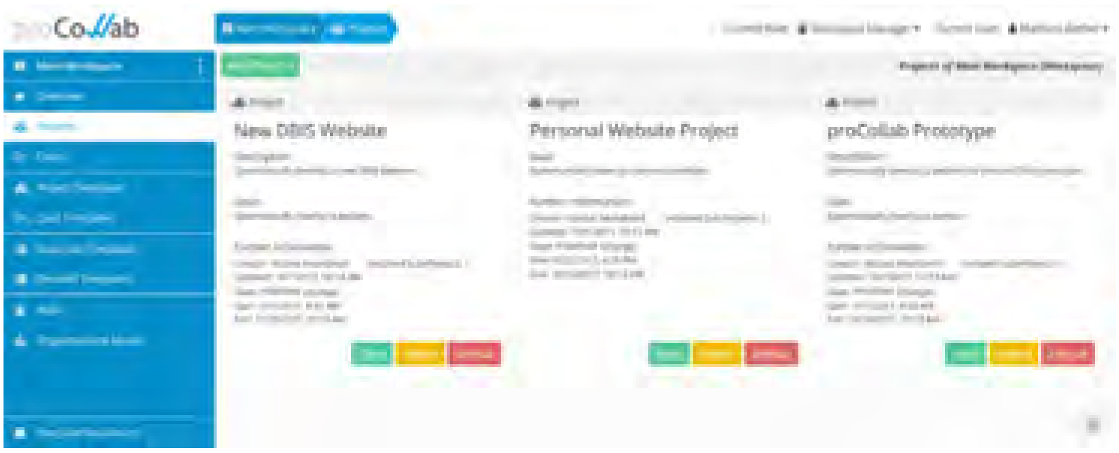


Figure 5.6: Overview of Available Process Instances Using Containers

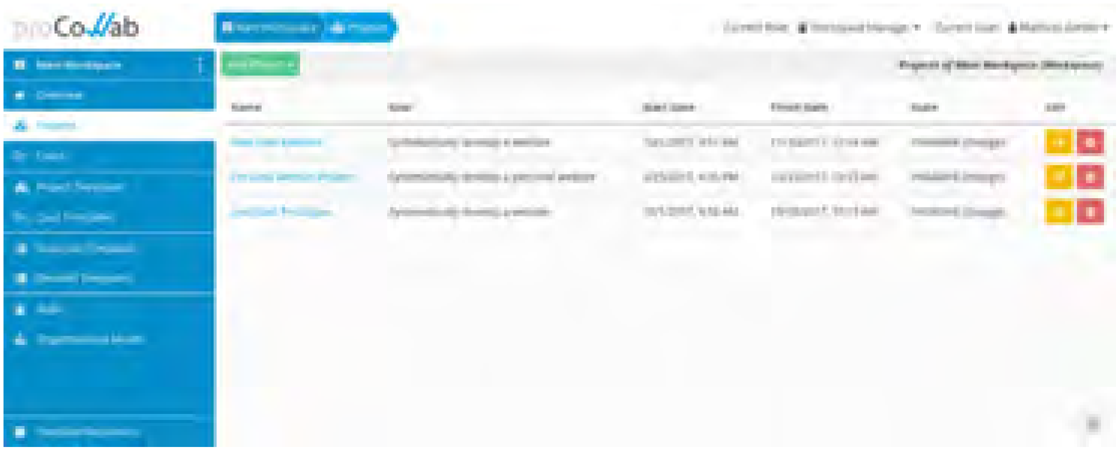
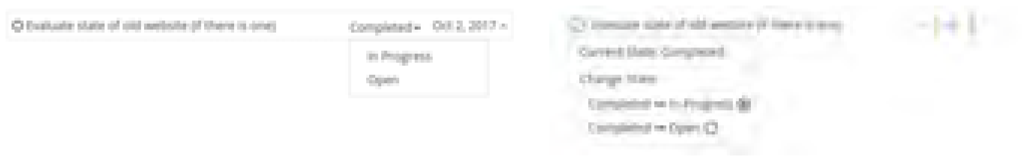


Figure 5.7: Overview of Available Process Instances Using a Table

The processes shown in the respective views are updated without the need to reload the view. This is accomplished by using observables provided by the caching service (see Section 5.2.2). The process overview also allows knowledge workers to create new process templates or instances, depending on whether he views process templates or process instances, respectively. When creating a process instance, it can also be chosen whether a blank PI should be created or if a PT should be instantiated to speed up the process creation. It the user chooses the latter, he is redirected to the process templates overview. The new process instance and template dialogue is realized based on modal windows for fast and easy access. Updating the processes is also done using a modal window in which a knowledge worker may make updates and view the state model graph of the selected process (see Section 5.2.12). If a knowledge worker wants to work with a process or just view its content, he may press the open button referring him to the process overview view (see Section 5.2.9).

### 5.2.6 Task Tree

To provide as much information as possible on the screen, the design of the individual task tree element views provided by the *proCollab* web client were made more compact. Figure 5.8 shows a comparison between the previously used design of task tree elements (see Figure 5.8a) and the new design (see Figure 5.8b).



(a) (b)

Figure 5.8: Comparison of the Old Task Design and the New One

The current state and type of task tree element is now denoted by an icon only saving much of space. The due date is no longer displayed directly in the task tree element. Instead, the user has to hover over the blue information icon to see any details regarding the task tree element. To warn knowledge workers of task tree elements that are due any time soon, a clock icon in different colors is displayed. The different times before the due date at which the icon appears and changes its color, can be configured using the configuration file discussed in Section 5.2.1. Finally, the priority of a process can also be displayed using an icon defined in the configuration file.

Figure 5.9 show the different components a task tree comprises:

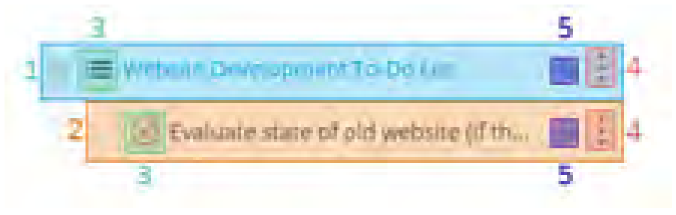

3
5
1
Website Development To-Do List
4
2
Evaluate state of old website (if th...
4
3
5

Figure 5.9: Different Components a TaskTreeViewComponent Comprises

1. `TaskTreeViewComponent`
2. `TaskViewComponent`
3. `SetStateComponent`
4. `EditDropdownComponent`
5. `InformationDropdownComponent`

The `TaskTreeViewComponent` can contain further `TaskTreeViewComponents` to display subordinate task trees as well as `TaskViewComponents` to display tasks. The `TaskViewComponent`, in turn, can also contain further `TaskViewComponents` to display subordinate tasks. To improve maintainability, `TaskTreeViewComponents` and `TaskViewComponents` share all components they comprise (i.e., `SetStateComponent`, `EditDropdownComponent`, and `InformationDropdownComponent`). The `SetStateCompone` is used to display the current state as well as to change the state. Furthermore, the `InformationDropdownComponent` is used to display information regarding the task tree element (e.g., its due date, priority, or current state). Finally, the `EditDropdownComponent` is used to manage the task tree element (see Section 5.2.7).

Listing 5.9 shows an example of a task tree instance object represented in JSON. The JSON object contains basic information, such as the name of the creator and the name of the task tree instance. Additionally, the name and the id of the template that the task tree instance was derived from, is provided. Furthermore, it contains the two arrays `subTaskTreeIds` and `taskIds`. The `subTaskTreeIds` array contains the ids of the subordinate task trees, whereas `taskIds` contains all the ids of the tasks, which the task tree comprises. However, these arrays do not provide any information regarding the order in which the task tree elements need to be displayed. The order is provided by the individual objects comprised by the `taskTreeLists` object. More specifically, each of the individual objects contains an `elements` array, containing all the ids of the task tree elements in the order that they shall be displayed. The keys for the individual objects in the `taskTreeLists` object are the ids of their parent task tree elements.

```
{
  "id":842,
  "version":1,
  "dateCreated":1506350141000,
  "dateUpdated":1506350142000,
  "creatorId":2,
  "creatorName":"Matthias Gerber",
  "grantedPrivileges":[...],
  "name":"Website Development To-Do List",
  "description":"",
  "taskTreeTypeId":204,
  "parentTaskTreeIds":[],
  "subTaskTreeIds":[],
  "taskIds":[
    845,865,848,860,861,844,858,854,864,850,853,856,859,843,852,863,849,
      846,855,847,862,857,851
      ],
  "taskTreeLists":{
    "842":{
      "id":2706,
      "version":0,
      "dateCreated":1506350142000,
      "dateUpdated":1506350142000,
      "elements":[843,847,863,850,853,846,865,860,852,844,859,856,849,845]
    },
    ...
  },
  "parentOfTaskTreeLists":{
    "2706":842,
    "2707":846,
    "2708":844
  },
  "processInstancesIds":[840],
  "taskTreeTemplateId":806,
  "taskTreeTemplateName":"Website Development To-Do List"
}
```

Listing 5.9: JSON Representation of a Task Tree Instance

Listing 5.10 shows an example how different components are used in the `TaskTreeViewComponent` template. Additionally, this example shows the use of two core Angular 4 functionalities, `*ngFor` and `*ngIf`. `*ngFor` allows to iterate over any iterable object (i.e., an array), while `*ngIf` can be used for conditional statements. The `taskTree` variable contains a TTI (see Listing 5.9) or TTT. The different components are added to the template of the view through selectors. The `SetStateComponent`, for example, is included by using `<set-state ...></set-state>` in the template.

```
<div>
  <!-- Expand/Collapse Button -->
  <div class="btn-toggle" (click)="toggleTree()">
    ...
  </div>
  <div class="set-state-btn">
    <set-state [item]="taskTree" ...></set-state>
  </div>
  <!-- Task Tree Content -->
  <div class="item-content">
    <a ...>
      {{ taskTree?.name }}
    </a>
    ...
    <information-dropdown ... ></information-dropdown>
    <edit-dropdown ...></edit-dropdown>
  </div>
</div>
```

```
<span *ngFor="let itemId of taskTree?.taskTreeLists[taskTree?.id].elements">
  <!-- Insert taks -->
  <span *ngIf="taskTree?.taskIds.includes(itemId)">
    <task-view [task]="tasks[itemId]" ...></task-view>
  </span>
  <!-- Insert subTaskTrees -->
  <span *ngIf="taskTree?.subTaskTreeIds.includes(itemId)">
    <task-tree-view [taskTree]="taskTrees[itemId]" ...></task-tree-view>
  </span>
</span>
```

Listing 5.10: Excerpt of the TaskTreeView Implementation

### Context menu

The context menu of each task tree element can be triggered by clicking the vertical ellipsis on its right-hand side. This opens a drop-down menu with the different actions available to the knowledge worker to manage the task tree element (see Figure 5.10).

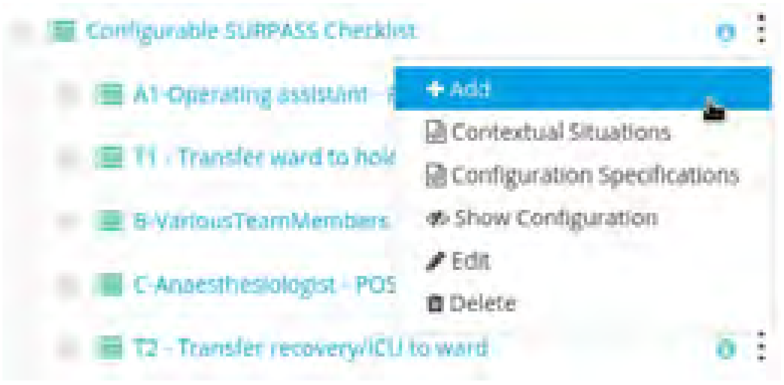

Configurable SURPASS Checklist
A1-Operating assistant
T1 - Transfer ward to hold
B-VariousTeamMembers
C-Anaesthesiologist - POS
T2 - Transfer recovery/ICU to ward
+ Add
Contextual Situations
Configuration Specifications
Show Configuration
Edit
Delete

Figure 5.10: Context Menu of a Task Tree Template

In particular, the individual entries allow knowledge workers to:

- Add new tasks or task trees (see Section 5.2.7).
- Edit the task tree element using a modal window.

- Delete the task tree element after confirming the action in a confirmation modal dialog (see Section 5.2.4).

If a TTT is selected, task tree configurations are additionally managed and displayed (see Section 5.2.4).

- Contextual Situations: Management of configuration parameters and contextual situations.
- Configuration Specifications: Listing of all the available configuration specifications.
- Show Configuration: Preview, of different configurations.

### 5.2.7 Management of Task Trees

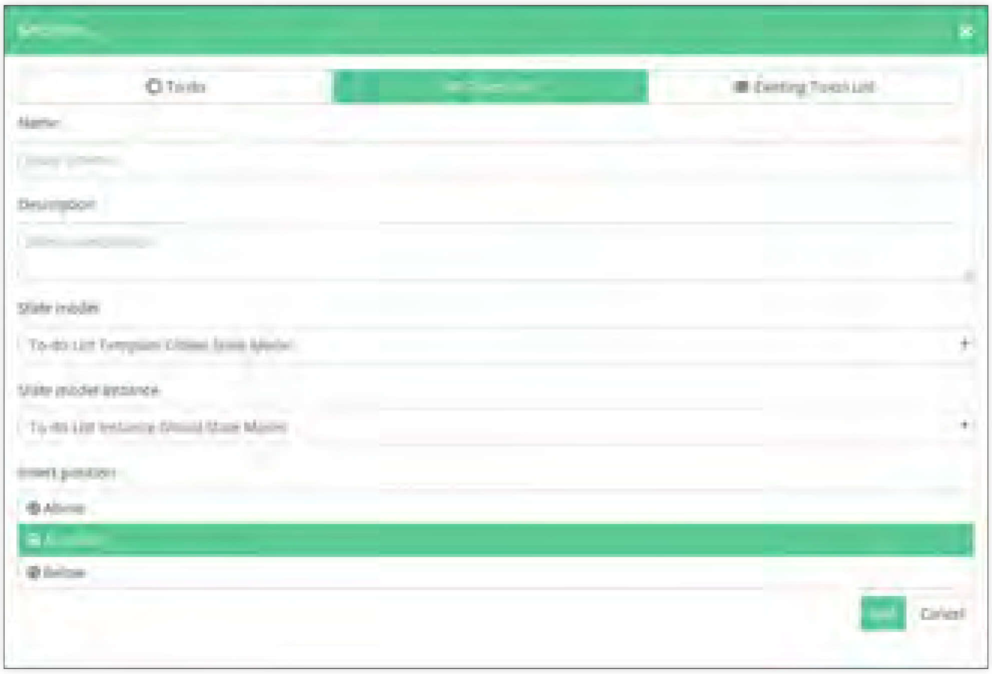
To support the task-centric *proCollab* approach, basic task tree management functionality such as adding, deleting and updating task tree elements is needed. Additionally, more advanced features such as the movement of task tree elements are needed to increase efficiency.

Figure 5.11 shows the modal window used for adding new task tree elements. This modal window allows knowledge workers to select what type of task tree element they want to add and provide a name and a description. Furthermore, they may select the state model that the task tree element should be based on, as well as the position at which the task tree element should be inserted into the task tree.

Additionally, the functionalities to smoothly move task tree elements was implemented based on the drag and drop approach discussed in Section 4.2.4. Figure 5.12 shows an example of a task being moved from one position to another. The task that is to be moved is shown with less opacity to provide a preview of the final task tree.

### 5.2.8 Task Tree Configuration

Task Tree Configurations were implemented according to the approach suggested in Section 4.2.5, enabling knowledge workers to manage, use and preview task tree configurations.



To do | [illegible] | Existing To-do List

Name:

Description:

State model:

To-do List Template (State Model)

State model instance:

To-do List Instance (State Model)

Insert position:

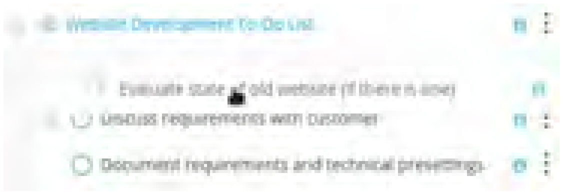
Above

[illegible]

Below

Add Cancel

Figure 5.11: Adding a New Task Tree or Task



Website Development To-Do List

Evaluate state of old website (if there is one)

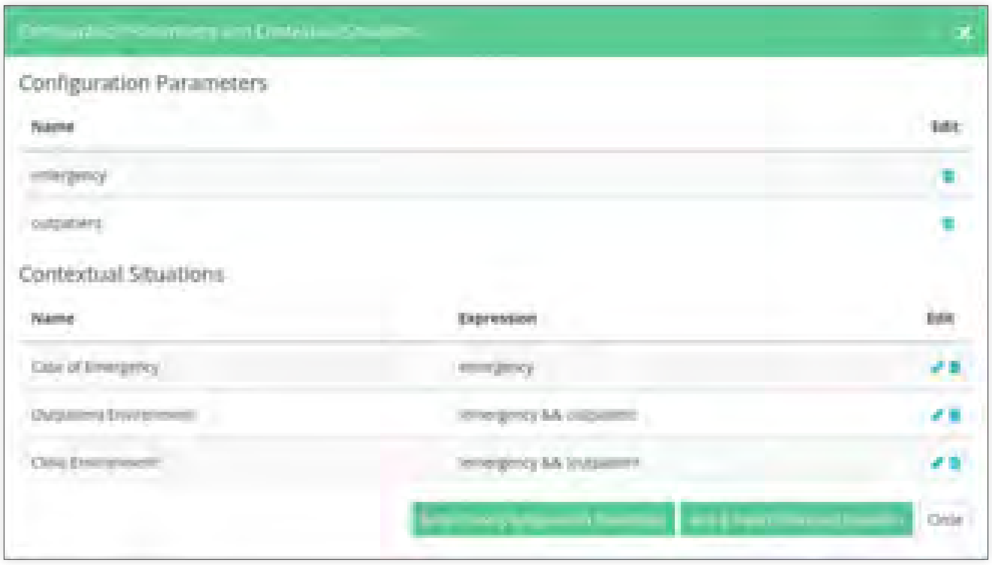
Discuss requirements with customer

Document requirements and technical presettings

Figure 5.12: Moving a Task Tree Element

### Configuration Parameters

Figure 5.13 shows the modal window that was realized to display and manage configuration parameters and contextual situations. A pencil and trashcan icon next to the individual entries enables users to update and delete them, respectively.


[illegible]

Configuration Parameters

| Name | Edit |
|---|---|
| emergency | 🗑 |
| outpatient | 🗑 |

Contextual Situations

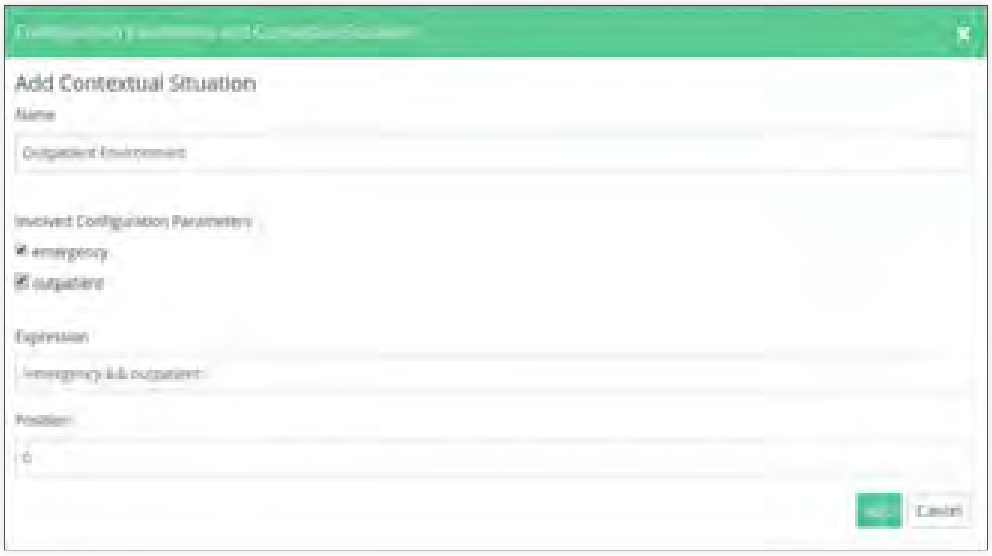
| Name | Expression | Edit |
|---|---|---|
| Case of Emergency | emergency | ✎ 🗑 |
| Outpatient Environment | !emergency && outpatient | ✎ 🗑 |
| Clinic Environment | !emergency && !outpatient | ✎ 🗑 |

Figure 5.13: Configuration Parameters and Contextual Situations Modal Window

Through the buttons on the bottom of the modal window, new configuration parameters and contextual situations can be added. A new contextual situation is created by using a modal window implemented according to the concept discussed in Section 4.2.5 (see Figure 5.14). With the modal window, users can define a name, the desired position and the expression that will trigger the contextual situation.

### Configuration Specifications

The configuration specifications modal window (see Figure 5.15) lists all the available task tree configurations including the name of the task tree elements, which are inserted by the configuration specifications. Furthermore, this table provides additional information to the user such as at which position and into which task tree the task tree element will


Configuration Parameters and Contextual Situation

Add Contextual Situation

Name

Outpatient Environment

Involved Configuration Parameters

☑ emergency

☑ outpatient

Expression

!emergency && outpatient

Position

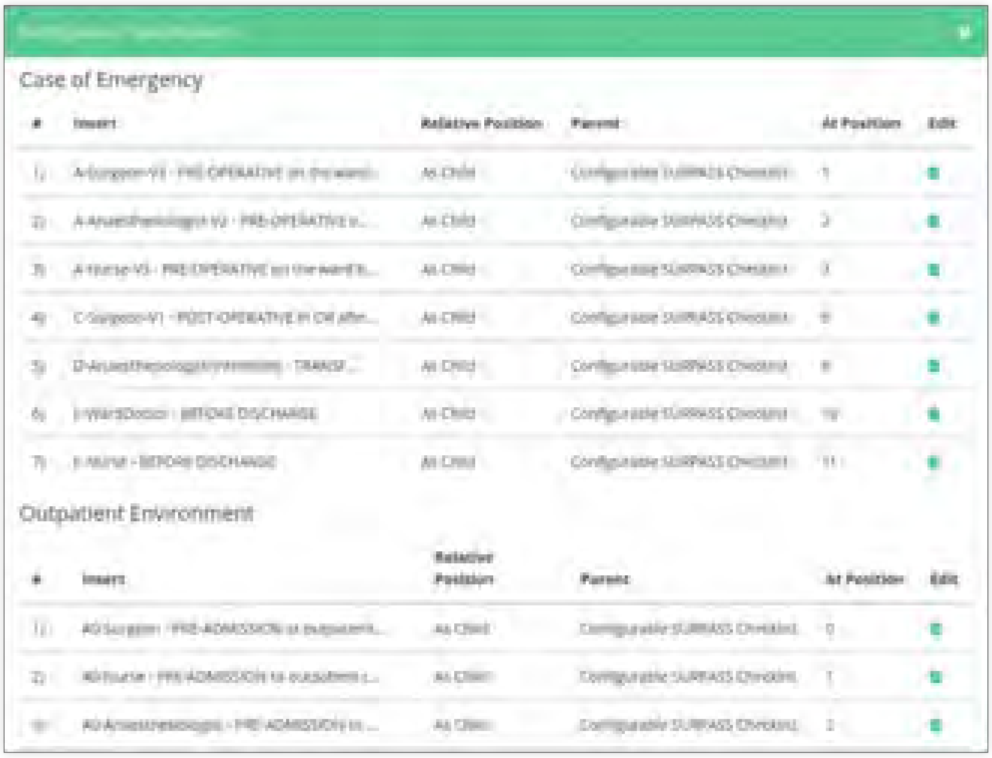
0

Figure 5.14: Adding a Contextual Situation

be inserted. Finally, the modal window also enables the user to remove individual task tree elements from the configuration specifications.

### Show Configuration

Section 4.2.6 discussed the need of previews for the configurations based on contextual situations. Figure 5.16 shows the workflow for previewing task tree configurations. First, the menu item "Show Configuration" has to be selected. This opens a modal window containing a selection element, with all the available contextual situations. After the user has selected the contextual situation, he wants to preview, the modal window is closed again and the task tree is updated to display the task tree for the selected contextual situation. The name of the contextual situation is displayed next to the name of the task tree. In addition, an icon that allows the user to end the preview of the contextual situation is shown.



Case of Emergency

| # | Insert | Relative Position | Parent | At Position | Edit |
|---|---|---|---|---|---|
| 1) | A-Surgeon-V1 - PRE-OPERATIVE on the ward ... | As Child | Configurable SURPASS Checklist | 1 | 🟩 |
| 2) | A-Anaesthesiologist-V2 - PRE-OPERATIVE o... | As Child | Configurable SURPASS Checklist | 2 | 🟩 |
| 3) | A-Nurse-V3 - PRE-OPERATIVE on the ward b... | As Child | Configurable SURPASS Checklist | 3 | 🟩 |
| 4) | C-Surgeon-V1 - POST-OPERATIVE in OR after... | As Child | Configurable SURPASS Checklist | 6 | 🟩 |
| 5) | D-Anaesthesiologist(mmmmm) - TRANSF... | As Child | Configurable SURPASS Checklist | 8 | 🟩 |
| 6) | E-WardDoctor - BEFORE DISCHARGE | As Child | Configurable SURPASS Checklist | 10 | 🟩 |
| 7) | E-Nurse - BEFORE DISCHARGE | As Child | Configurable SURPASS Checklist | 11 | 🟩 |

Outpatient Environment

| # | Insert | Relative Position | Parent | At Position | Edit |
|---|---|---|---|---|---|
| 1) | A0-Surgeon - PRE-ADMISSION at outpatient... | As Child | Configurable SURPASS Checklist | 0 | 🟩 |
| 2) | A0-Nurse - PRE-ADMISSION to outpatient c... | As Child | Configurable SURPASS Checklist | 1 | 🟩 |
| 3) | A0-Anaesthesiologist - PRE-ADMISSION to ... | As Child | Configurable SURPASS Checklist | 2 | 🟩 |

Figure 5.15: Configuration Specification Modal Window

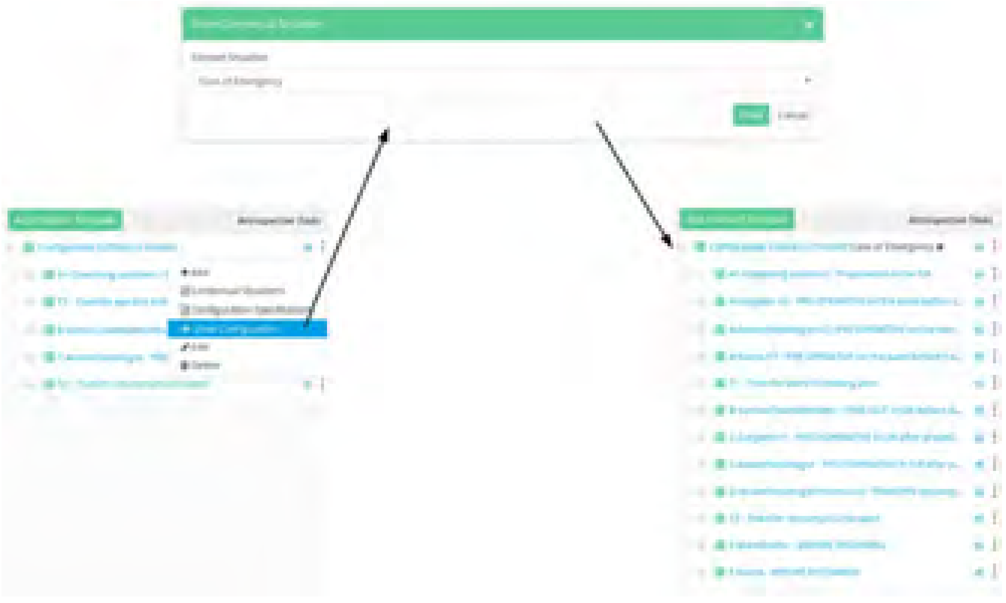


Figure 5.16: Displaying a Contextual Situation

### 5.2.9 Process Assessment

The process overview view aims to provide knowledge workers a quick overview of the current state of a process. Thus, the approach featuring separate columns for prospective and retrospective tasks was chosen as discussed in Section 4.2.3 (see Figure 5.17). The column in the middle provides information regarding the process. At the top of the two task tree columns, buttons allow knowledge workers to add new task trees to the process. In the middle column, one button is located to let knowledge workers update the information regarding the current process. Moreover, an additional button enables the adding of sub-processes. The middle column also contains an overview of all sub-processes of the selected process.

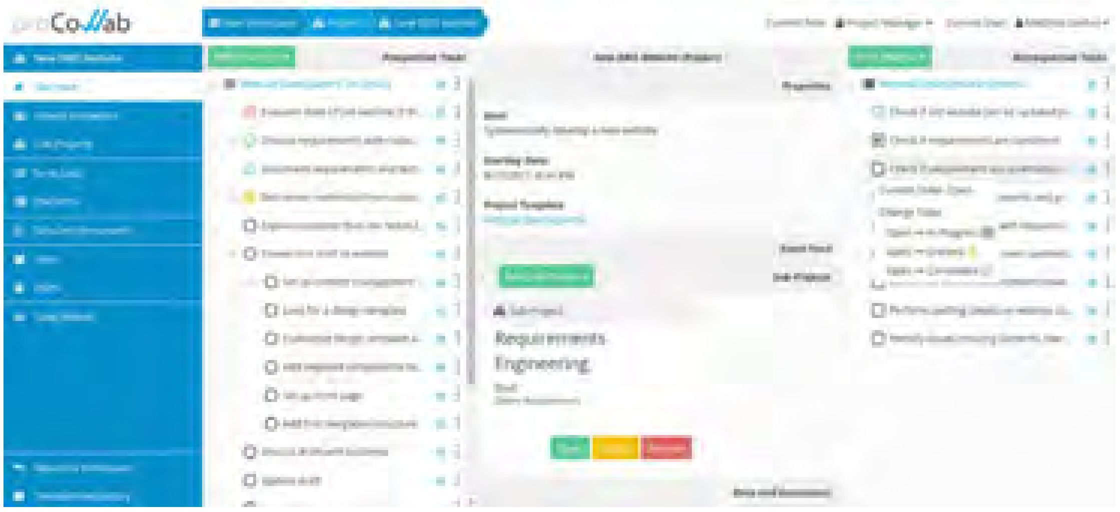


Figure 5.17: Overview of a Process Instance

### 5.2.10 Task Tree View

The concept for the task tree view was discussed in Section 4.2.6. The task tree view first provides knowledge worker a list of all task trees in the left column. A label informs the user, how many tasks a task tree comprises. The task tree elements of the selected task tree are shown in the right column in full detail. Additionally, new tasks can be added by clicking the button at the top, opening the appropriate modal window.

The task tree view allows knowledge workers to focus on a specific task tree as well as to quickly jump between task trees. Figure 5.18 shows an example of the task tree view. It was implemented using two `TaskTreeViewComponents` (see 5.2.6) placed next to each other. Variables passed to the `TaskTreeViewComponents` are used to configure their appearance (i.e., to not display tasks).

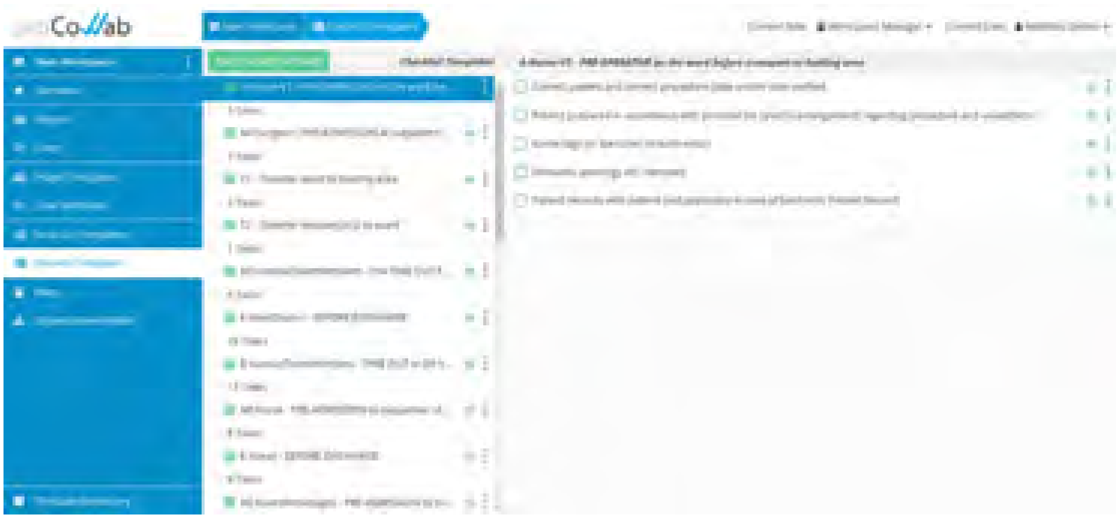


Figure 5.18: Task Tree Templates Overview

### 5.2.11 Assign Privileges

The *proColallab* web client already allowed the creation of roles in its previous version. It was, however, not possible to assign privileges to them. In Section 4.4.1, a view to select and assign privileges was presented and discussed. Figure 5.19 shows the implementation of this view. The user first has to narrow down the privileges by selecting the desired context type, target type and action type. He can then select the privileges using checkboxes and assign them by clicking the green button to the right. An input at the top of each column allows the user to filter the entries. The filter enables him to find the desired items more quickly.

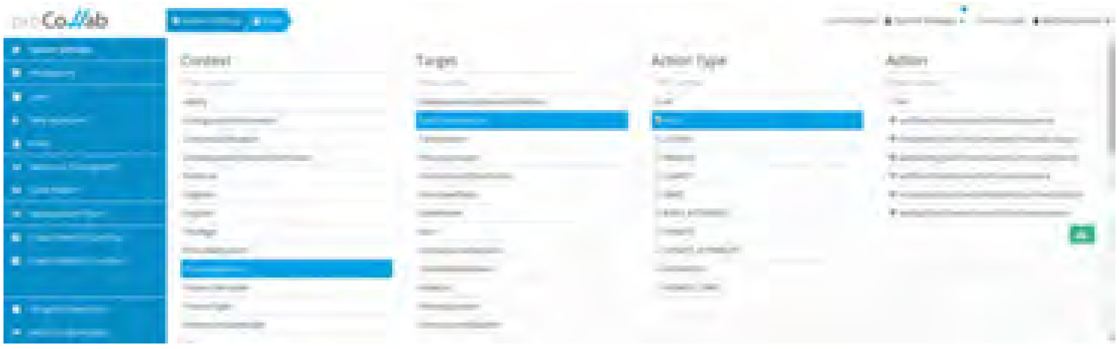


Figure 5.19: Assign Privileges to a Role

### 5.2.12 State Model Graph

This section discusses the implementation of the concepts for displaying and managing state models, discussed in Section 4.4.3. All the following state model functionality was implemented for reference state models as well as refining state models.

#### Displaying Refined State Models

The *proCollab* web client already enabled users to view state models using graphs in the previous version (see Section 5.1.8). However, it lacked the ability to display refined states. To address this issue, functionality was added allowing users to view the refining state model by double clicking on the refinable state (see Figure 5.20). Additionally, the design of the state model graph was changed providing a better way to distinguish the different properties of a state. For example to denote whether a state is the initial state or a final state and if the state is refinable. The new design follows the notation introduced in [MR17c].

#### Updating a State Model

The state model graph may also be used to update a reference state model (see Figure 5.21). An input at the top of the modal window allows users to change the name of a state model. Furthermore, an "Add State" button allows users to add a new state

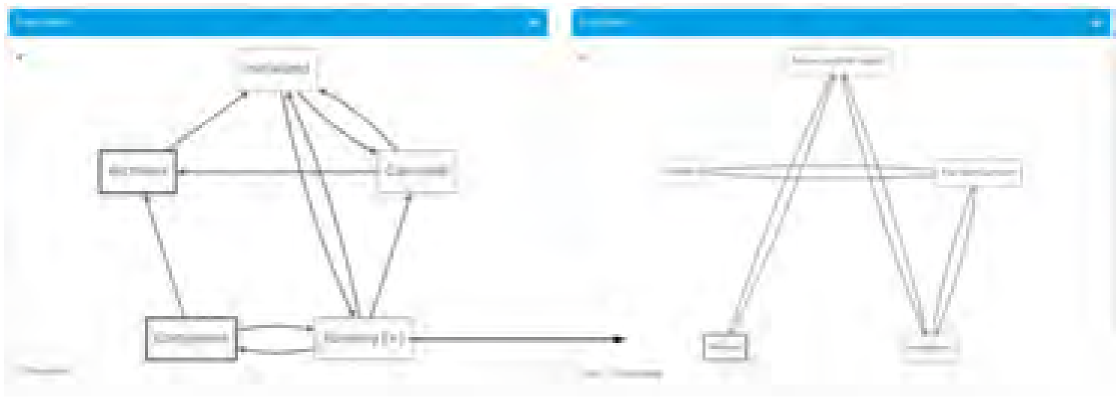


Figure 5.20: Displaying a Refined State Model

(see Section 5.2.12). After a state has been selected, an "Edit State" and a 'Remove State' button are displayed, enabling the user to update or delete the selected state. If a transition is selected these two buttons will be replaced by a single "Delete Transition" button allowing the user to remove the selected transition. New transitions can be added by drawing a line between two states. The line is shown as a light green arrow. To save the changes applied to the state model, the "Update" button in the bottom right needs to be clicked by the user.

### Adding a State

A modal window containing a graph can be used to add a state to a state model (see Figure 5.22). The user may enter a name for the state as well as select a state type and an icon. The icon is used to represent the state and can be selected, using a container, listing all the available icons. An input field on top of the container allows users to filter the icons based on their names. Additionally, a color picker at the right of the container may be used to select the desired color for the icon. Three checkboxes can be used to select whether the state is an initial state, final state or refinable state.

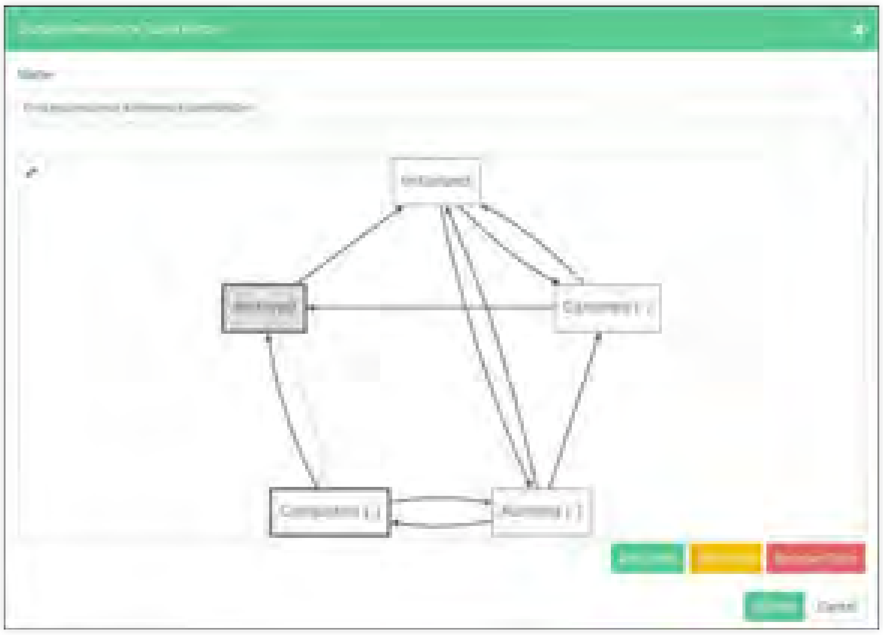


Figure 5.21: Updating a Reference State Model

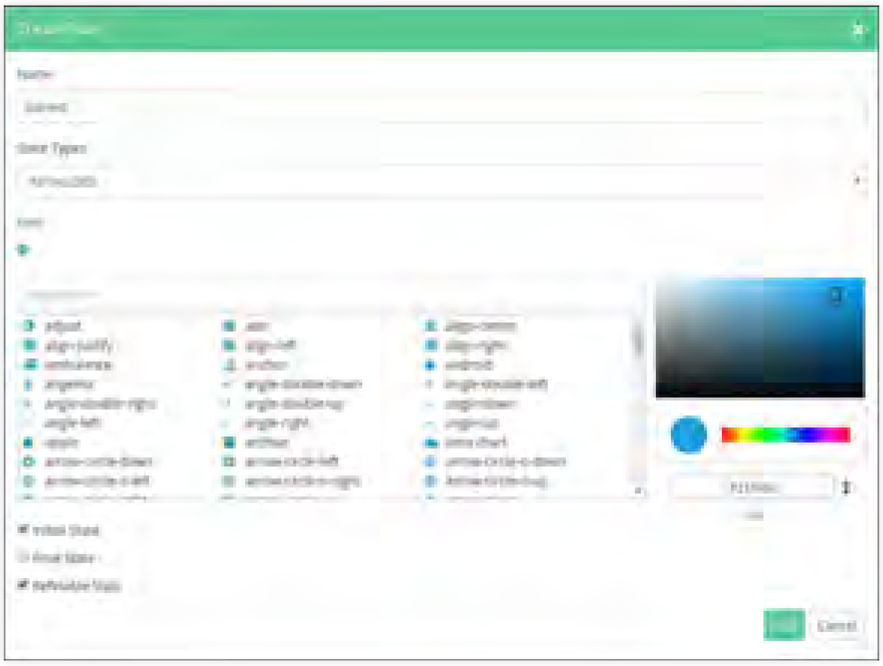


Figure 5.22: Adding a New State

### 5.2.13 Role Selection

In Section 4.6.2, the need for automatic and manual role selection was discussed. The automatic role selection is implemented using the `grantedPrivileges` array (see Listing 5.11). The latter is provided by every guarded object, for example, a workspace or process instance. The `grantedPrivileges` array contains objects providing all applicable roleAssignmentIds. Using these roleAssignmentIds, the corresponding roles are ascertained. Finally, these roles are compared, to find the role with the least privileges, which is then automatically selected.

```
grantedPrivileges: [
  0: {
    privilegesContextRestrictions:{}
    privilegesIds: [1024, 1033, ...]
    privilegesSpecializationRestrictions:{}
    privilegesStateRestrictions:{}
    roleAssignmentId:2114
    roleAssignmentVersion:0
  },
  1: { ... },
  2: { ... }
]
```

Listing 5.11: GrantedPrivileges Array

To also provide manual role selection, the available roles are ascertained the same way as for the automatic role selection. The available roles are then presented to the user via a drop-down menu (see Figure 5.23). The user may manually switch his role by selecting one of them. If a role is selected manually, it is denoted by a blue dot next to the currently selected role. This role stays selected until a guarded entity is selected to which the role is not assigned. The user can also manually switch back to the automatically selected role by clicking on the blue dot.

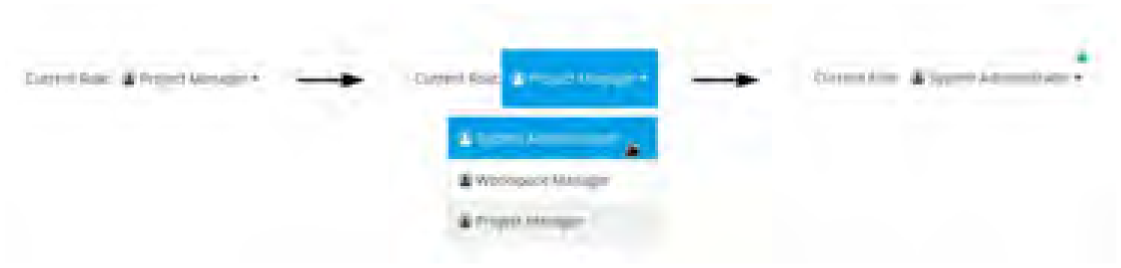

Current Role: Project Manager
Current Role: Project Manager
System Administrator
Workspace Manager
Project Manager
Current Role: System Administrator

Figure 5.23: Switching Roles Manually

### 5.2.14 Role Based User Interface

An entire role-based user interface was implemented for the sidebar navigation. Listing 5.12 shows an example of an entry being added to the sidebar navigation of a workspace. First, it is checked whether the user has the required privilege to perform the required action. For example if he wants to view the workspace overview, he needs the privilege required to get the workspace object from the server. To check whether the user has the needed privileges the `check()` function is called with the name of the action (i.e., `getWorkspace`) and the `grantedPrivileges` array of the current workspace as parameters (see Listing 5.11).

```
if(this.check('getWorkspace', workspace.grantedPrivileges)) {
  this.addNavItem('Overview', '/dashboard/workspace/'+ workspace.id, ...);
}
```

Listing 5.12: Permission Check

The `check()` function then evaluates whether the user—in his current role—has the permission to perform the action and returns `true` or `false`, respectively. When the `check()` function returned `true`, the entry is added to the sidebar and when not, it is omitted. Currently, this approach is only implemented for the sidebar, but the function can be easily used on comparable views or navigation structures as well.

### 5.2.15 Registration

To allow the registration of new account the solution discussed in Section 4.5.2 was implemented.

Administrators may accept or decline applications in the "View Applicants" view. In this view, all applicants are listed using a table with a button, allowing administrators to accept or reject the individual applications.

### 5.2.16 Password Reset

The password reset was realized by using a form in which the user has to enter his e-mail address. After submitting this form, an e-mail is dispatched from the *proCollab* server to this e-mail address containing a URL with an authorization token. This URL points to a view, enabling him to enter a new password. After he entered a new password, he is redirected to the login view, where he can log in to the *proCollab* web client using his new password.

# 6 Conclusion

Chapter 6 concludes, with a short summary in Section 6.1 and a prospect of future work in Section 6.2.

## 6.1 Summary

The focus of this thesis was to study the existing *proCollab* client and then to systematically improve it. To accomplish this, the fundamental concepts of the *proCollab* prototype were introduced in Chapter 2 and exemplified using an application case. This elucidated the necessity for a task list-based, collaborative approach to holistically support knowledge workers and their KiPs.

Subsequently, the requirements for the *proCollab* web client were collected and discussed in Chapter 3. Next, the current state of the *proCollab* web client was analyzed. The results were used to determine the requirements not yet met by the *proCollab* web client. Finally, comparable tools were analyzed to compare and evaluate concepts proposed by the individual tools.

Chapter 4 provided a discussion of different concepts to implement the requirements gathered in the previous chapter. The concepts were in particular enriched using mockups to give a clear picture of possible implementations. While doing so, the advantages and disadvantages of the different concepts were discussed and the concepts destined for implementation were selected.

Chapter 5 first introduced the technologies used for the implementation of the *proCollab* client. Subsequently, the implementation of the concepts, selected in the previous chapter, were described in part using screenshots and code excerpts.

Overall, the implementation, which was conducted in the scope of this work, covers a wide range of concepts developed for *proCollab*. In particular, this range comprises, for example, the sophisticated caching mechanisms, the dynamic, role-based navigation structures, the advanced task tree as well as the state management. For all these concepts, comprehensive implementations have been realized and closely integrated with each other. For illustrative purposes, however, only excerpts of these detailed implementations could be discussed in this work.

As of today, the *proCollab* web client now even more provides a feature-rich foundation for future enhancements and research conducted in the proCollab research project [MR17b].

## 6.2 Future Work

In the course of this work, the *proCollab* web client was enhanced with numerous important features to improve the support provided to knowledge workers and KiPs. However, there are still some features missing. For example, it is not yet possible to link data and documents to the different KiPs. This feature would further enhance the support provided for KiPs.

It is currently not possible to move task tree elements between processes. This could be accomplished by providing some sort of cut and paste functionality in addition to the implemented drag and drop approach.

Another possible enhancement would be the ability to assign tasks to a specific knowledge worker. This way knowledge workers would be able to better plan out their responsibilities, thus avoiding duplication of effort.

To support KiPs on an international scale, more languages should be supported. Thus, the *proCollab* web client could also be localized to reach a wider audience.

To keep knowledge workers better informed, an event feed could be implemented. This feed could be used to inform them about upcoming deadlines, work done by their co-workers and other information pertaining their KiPs.

Further, a workspace overview could be implemented showing open processes and their progress and pending tasks. This could be accomplished by using graphs and diagrams to provide a pleasant user experience.

The thus enhanced *proCollab* client could provide another step towards a more complete support for knowledge workers and KiPs. The future of this research project will show how well suited the *proCollab* approach is, to holistically support knowledge workers and their KiPs.

# List of Figures

2.1 PDSA Cycle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 KiP Lifecycle [MR14] . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 proCollab Key Components [MR17c] . . . . . . . . . . . . . . . . . . . . 10

2.4 *proCollab* State Management Entities [MR17c] . . . . . . . . . . . . . . . 14

2.5 Reference State Models [MR17c] . . . . . . . . . . . . . . . . . . . . . 16

2.6 Individual State Model for *proCollab* Process Instance [MR17c] . . . . . . 17

2.7 Specialization Entities [MR17c] . . . . . . . . . . . . . . . . . . . . . . 17

2.8 Overview of Object-Specific Role-Based Access Control [MR17d] . . . . . 19

3.1 Process Instance Overview . . . . . . . . . . . . . . . . . . . . . . . . 30

3.2 Checklist Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.3 View of the States and Transitions of a Stateful Entity . . . . . . . . . . . 32

3.4 Basecamp To-do List Overview . . . . . . . . . . . . . . . . . . . . . . 35

3.5 Basecamp Latest Activities . . . . . . . . . . . . . . . . . . . . . . . . 36

3.6 active.collab Task List View . . . . . . . . . . . . . . . . . . . . . . . 37

3.7 active.collab Latest Activities . . . . . . . . . . . . . . . . . . . . . . 37

3.8 Project Overview in Flow . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.9 Catch Up View of Flow . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4.1 Notation Used in the Flow Charts . . . . . . . . . . . . . . . . . . . . . 41

4.2 Overview of the Navigation Menus and their Interrelationship . . . . . . . 42

4.3 Breadcrumbs and Alternative Approaches to Deal with Insufficient Space 44

4.4 Overview of the Views Linked to the Workspace Navigation Menu . . . . . 45

4.5 Overview of the Different Processes in a Workspace Using a Table . . . . 46

4.6 Overview of the Different Processes in a Workspace Using Containers . . 47

4.7 Inline Editing of a Process . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.8 One Option for the Process Overview . . . . . . . . . . . . . . . . . . . 48

4.9 Another Option for the Process Overview . . . . . . . . . . . . . . . . . 49

4.10 Different Task Tree Designs and Approaches for Changing the State of Task Tree Elements . . . . 51
4.11 Different Approaches to Choosing the New Position of a Task or Task Tree with the Help of Buttons . . . . 52
4.12 Choosing the Position of a New Task Tree Element Using a Modal . . . . 52
4.13 Different Ways to Move a Task . . . . 53
4.14 Flow Chart for Adding a Contextual Situation . . . . 55
4.15 Adding a New Contextual Situation . . . . 56
4.16 Different Ways of Displaying Contextual Situations . . . . 56
4.17 One Option for the Task Tree Overview . . . . 57
4.18 Task Tree Overview Using Two Columns . . . . 58
4.19 Organizational Model Overview . . . . 58
4.20 Overview of the Views Linked to the Template Repository Navigation Menu 59
4.21 Overview of the Views Linked to the System Settings Navigation Menu . . 60
4.22 Assign Privileges to a Pre-Selected Role . . . . 61
4.23 Displaying a Reference State Model as Graph . . . . 62
4.24 Adding a New State . . . . 63
4.25 Text-Based Approach for Adding and Removing State Transitions . . . . 64
4.26 Graphical approach for Adding and Removing State Transitions . . . . 64
4.27 Flow Chart for the Password Recovery Procedure . . . . 66
4.28 Flow Chart for the Registration Procedure . . . . 67
4.29 Confirmation Modal Dialog for Destructive Actions . . . . 68
4.30 Confirmation Modal Dialog for Destructive Actions Requiring Additional Input . . . . 69
4.31 Displaying the Available Role Using Radio Buttons . . . . 69
4.32 . . . . 70

5.1 Angular 4 Overview [Ang17a] . . . . 73
5.2 Color Scheme . . . . 74
5.3 Overview of the Different Navigation Menus . . . . 79
5.4 Breadcrumbs Trail . . . . 80
5.5 A Confirmation Modal Dialog for Destructive or Important Actions . . . . 83

5.6 Overview of Available Process Instances Using Containers . . . . . . . . 84
5.7 Overview of Available Process Instances Using a Table . . . . . . . . . . 84
5.8 Comparison of the Old Task Design and the New One . . . . . . . . . . . 85
5.9 Different Components a TaskTreeViewComponent Comprises . . . . . . . 86
5.10 Context Menu of a Task Tree Template . . . . . . . . . . . . . . . . . . . . 89
5.11 Adding a New Task Tree or Task . . . . . . . . . . . . . . . . . . . . . . . 91
5.12 Moving a Task Tree Element . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.13 Configuration Parameters and Contextual Situations Modal Window . . . 92
5.14 Adding a Contextual Situation . . . . . . . . . . . . . . . . . . . . . . . . 93
5.15 Configuration Specification Modal Window . . . . . . . . . . . . . . . . . 94
5.16 Displaying a Contextual Situation . . . . . . . . . . . . . . . . . . . . . . 94
5.17 Overview of a Process Instance . . . . . . . . . . . . . . . . . . . . . . . 95
5.18 Task Tree Templates Overview . . . . . . . . . . . . . . . . . . . . . . . . 96
5.19 Assign Privileges to a Role . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.20 Displaying a Refined State Model . . . . . . . . . . . . . . . . . . . . . . 98
5.21 Updating a Reference State Model . . . . . . . . . . . . . . . . . . . . . . 99
5.22 Adding a New State . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
5.23 Switching Roles Manually . . . . . . . . . . . . . . . . . . . . . . . . . . 101

# Bibliography

[Act17] Active Collab. *Active Collab*. 2017. URL: `https://activecollab.com/` (last visited July 9, 2017).

[Ang17a] Angular Contributors. *Angular - Architecture Overview*. 2017. URL: `https://angular.io/guide/architecture` (last visited Oct. 1, 2017).

[Ang17b] Angular Contributors. *Angular - NgModules*. 2017. URL: `https://angular.io/guide/ngmodule` (last visited Oct. 1, 2017).

[Ang17c] Angular Contributors. *Angular - Style Guide*. 2017. URL: `https://angular.io/guide/styleguide` (last visited Oct. 1, 2017).

[Bas17] Basecamp. *Basecamp*. 2017. URL: `https://basecamp.com` (last visited July 6, 2017).

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. “Understanding TypeScript”. In: *Object-Oriented Programming: 28th European Conference (ECOOP 2014)*. 2014.

[Bus12] David Bustard. “Beyond mainstream adoption: From agile software development to agile organizational change”. In: *19th IEEE Int’l Conference and Workshops on Engineering of Computer Based Systems (ECBS 2012)*. 2012, pp. 90–97.

[CL05] Christy Cheung and Matthew Lee. “The Asymmetric Effect of Website Attribute Performance on Satisfaction: An Empirical Study”. In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* C (2005), pp. 1–10.

[Cro06] Douglas Crockford. *RFC 4627 - The application/json media type for JavaScript Object Notation*. 2006. URL: `https://tools.ietf.org/html/rfc4627` (last visited Oct. 2, 2017).

[Cyt17] Cytosacape Contributors. *Cytoscape.js*. 2017. URL: `http://js.cytoscape.org/` (last visited Aug. 15, 2017).

[DJB95] Kevin Dooley, Timothy Johnson, and David Bush. “TQM, chaos and complexity”. In: *Human systems management* 14.4 (1995), pp. 287–302.

[Fie00] Roy Fielding. “Architectural styles and the design of network-based software architectures”. PhD thesis. University of California, Irvine, 2000.

[Flo17] Flow. *Flow*. 2017. URL: `https://www.getflow.com` (last visited July 16, 2017).

[FM11] Ian Fette and Alexey Melnikov. “RFC 6455 - The WebSockets Protocol”. In: (2011). URL: `https://tools.ietf.org/html/rfc6455` (last visited Oct. 2, 2017).

[Kow11] Julia Kowalewski. “Specialization and employment development in Germany: An analysis at the regional level”. In: *Papers in Regional Science* 90.4 (2011), pp. 789–811.

[Kru14] Steve Krug. *Don’t make me think!: a common sense approach to Web usability*. New Riders Publishing, Indianapolis, 2014.

[LR07] Richard Lenz and Manfred Reichert. “IT support for healthcare processes - premises, challenges, perspectives”. In: *Data and Knowledge Engineering* 61.1 (2007), pp. 39–58.

[MKR12] Nicolas Mundbrod, Jens Kolb, and Manfred Reichert. “Towards a System Support of Collaborative Knowledge Work”. In: *1st Int’l Workshop on Adaptive Case Management (ACM’12), BPM’12 Workshops*. LNBIP 132. Springer, 2012, pp. 31–42.

[MR14] Nicolas Mundbrod and Manfred Reichert. “Process-Aware Task Management Support for Knowledge-Intensive Business Processes: Findings, Challenges, Requirements”. In: *18th IEEE Int’l Distributed Object Computing Conference - Workshops and Demonstrations (EDOCW 2014)*. 2014, pp. 116–125.

[MR17a] Nicolas Mundbrod and Manfred Reichert. “Configurable and Executable Task Structures Supporting Knowledge-intensive Processes”. In: *36th International Conference on Conceptual Modelling (ER 2017)*. 2017.

[MR17b] Nicolas Mundbrod and Manfred Reichert. “Demonstrating Flexible Support for Knowledge-Intensive Processes with proCollab”. In: *Demo Track of the 15th International Conference on Business Process Management (BPM 2017)*. 2017.

[MR17c] Nicolas Mundbrod and Manfred Reichert. “Flexible Task Management Support for Knowledge-Intensive Processes”. In: *19th IEEE Int’l Enterprise Distributed Object Computing Conference (EDOC 2015)*. 2017.

[MR17d] Nicolas Mundbrod and Manfred Reichert. “Object-Specific Role-Based Access Control”. (Internal Paper in Review). Ulm University. 2017.

[Pro17] ProCollab Team. *proCollab*. 2017. URL: `http://procollab.de` (last visited Sept. 23, 2017).

[Pry+14] Rüdiger Pryss et al. “Supporting medical ward rounds through mobile task and process management”. In: *Information Systems and e-Business Management* 13.1 (2014), pp. 107–146.

[Rea17] ReactiveX Contributors. *ReactiveX - Observable*. 2017. URL: `http://reactivex.io/documentation/observable.html` (last visited Oct. 1, 2017).

[Sas17] Sass Contributors. *Sass Documentation*. 2017. URL: `http://sass-lang.com/documentation/` (last visited Sept. 10, 2017).

[Sch04] Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004.

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 2010.

[Tie10] Michael Tiemann. “Wissensintensive Berufe”. In: *Wissenschaftliche Diskussionspapiere des Bundesinstitut für Berufsbildung* 114 (2010), pp. 1–70.

[TRH13] Julian Tiedeken, Manfred Reichert, and Joachim Herbst. “On the Integration of Electrical/Electronic Product Data in the Automotive Domain”. In: *Datenbank Spektrum* 13.3 (2013), pp. 189–199.

[Vac+11] Roman Vaculín et al. “Declarative business artifact centric modeling of decision and knowledge intensive business processes”. In: *15th IEEE Int’l Enterprise Distributed Object Computing Conference (EDOC 2011)*. 2011, pp. 151–160.

Name: Matthias Gerber Matrikelnummer: 726161

**Erklärung**

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ..........................................................................

Matthias Gerber