NTNU
Norwegian University of
Science and Technology

Test-Driven Conceptual Modelling

evaluation through a case study

Isaac Bernat-Casi

Master of Science in Informatics
Submission date: December 2011
Supervisor: John Krogstie, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The purpose of this project is to showcase the work cycle and feasibility of Test-Driven Conceptual Modelling (TDCM) on a real-sized system. TDCM is a novel methodology to develop conceptual schemas which can be understood as belonging to the wider eXtreme Programming (XP) family called Test-Driven Development (TDD). Its aim is to iteratively develop conceptual schemas through automated testing at the conceptual level.

In order to achieve its goal, this project is built upon the state of the art of conceptual schema testing, as seen in recent publications. It has been carried out in accordance with the design-science research model to ensure both rigour and relevance.

We contribute to TDCM experimentation by applying it to a case study. In particular, this project focus on the development, by reverse engineering, of the conceptual schema of Remember the Milk (RTM), a popular system that supports the management of tasks. As a complementary goal, some suggestions to improve the RTM system will be presented thanks to the knowledge gathered during the process. This document collects the result of this experience.

Our findings confirm that this methodology is promising, because the resultant conceptual's schema validation and high semantic quality paid off its relatively small additional development efforts.

Preface

This previous semester was my last one of regular classes. I chose two more optative subjects than I was required according to my needed number of ECTS credits, because I was conservative and did not want to risk failing a subject and not attaining the required credits before my Erasmus scholarship. I initially planned to start all the topics and just leave the ones that were going to be more difficult or time-consuming on the second week. One of those subjects was Requirements Engineering (RE), whose main professor was Antoni Olivé with the assistance of the PhD candidate Albert Tort, with the occasional appearance of Maria-Ribera Sancho. It certainly was one subject with a heavy weekly workload, but before I even considered leaving it, I was enthralled.

During the RE course I got the chance to know the TDCM approach and Conceptual Schema Testing Language (CSTL) processor personally. We worked on a small project with it, accomplishing very satisfactory results. I also learned about the Kano model and other facts and aspects I comment on along the project. After the course finished, Albert contacted me with an offer of a 30 ECTS credits project proposal to work on the Autumn semester. This project would mark the end of the 5-year long degree on Engineer on Informatics at Technical University of Catalonia (UPC). After a while, we finally managed to keep the project going, even if it was to be graded internationally at Norwegian University of Science and Technology (NTNU), thanks to Albert Tort, Antoni Olivé and John Krogstie to whom I am truly grateful.

Trondheim, December 2011

Isaac Bernat Casi

Contents

List of Figures

List of Algorithms

Part I.

Introduction and Background

1. Introduction

1.1. Overview

The main purpose of this project is the development of the conceptual schema of an existing information system by applying Test-Driven Conceptual Modelling (TDCM). In particular, we focus on the development, by reverse engineering, of the conceptual schema of Remember the Milk (RTM), a system that supports the management of tasks. This document reports the context, the application details and the results of this experience.

This chapter aims to describe the context, before properly starting the topic of the thesis. To start with, section 1.2 defines the terms of conceptual modelling, validation of conceptual schemas and reverse engineering, all of them paramount to understand the project. Next, at section 1.3 we show the notability of the work we are building upon. At section 1.4 there is a brief explanation of the motivation that made this project possible. On section 1.5 and section 1.6 the project objectives and its methodology to achieve them are defined. Finally, section 1.7 summarises the contents of the rest of the chapters on this document.

1.2. Context

1.2.1. Conceptual modelling

> A conceptual schema defines the general knowledge required by an information system to perform its functions.[16]

This means that, in order to develop any information system, software engineers and system designers must first define the conceptual schema of the system to-be. This practise is commonly known by the name of conceptual modelling.

1.2.2. Validation of conceptual schemas

The validation of a conceptual schema consists on the evaluation of its quality. According to the conceptual modelling quality framework proposed[13], a conceptual schema of an information system has semantic quality when it is valid and complete.

Validity means that the schema is correct and relevant. A conceptual schema is correct if the knowledge it defines is true for the domain, and it is relevant if the knowledge it defines is necessary for the system. Completeness means that the conceptual schema includes all relevant knowledge.

Actually there are two main challenges when it comes to validation of conceptual schemas.

1. Stakeholder involvement: Since stakeholders determine the requirements of an information system, the validation of the corresponding conceptual schema its to involve them. Moreover, as end-users are usually not comfortable with formalisms, they might find the conceptual schema difficult to understand. Hence, they may not really be able to validate that it represents precisely what they expect.

2. Rigorous validation: Many techniques that aim to validate requirement specifications are essentially manual. Validation is the "confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled" [10] . In software development, "validation concerns the process of examining a product to determine conformity with user needs" [10] . Not taking into account human errors, the main issue here is the fact that it is not possible to decide when the validation has ended at conceptual modelling stage. This brings us to two possible undesirable scenarios:

   a) Too many human resources are devoted to validation. In this case, resources are wasted, as the schema is being validated and evaluated when it is already correct.

   b) Not enough resources are put into validation. In this case the resultant conceptual schema is wrong. Correcting the errors when detected during further steps (e.g. design, implementation, maintenance, etc.) can make it orders of magnitude (i.e. tens or hundreds of times) more costly to repair than at this stage. Furthermore, it will incur on delays to provide the finished working product, and potentially dangerous side-effects if it goes unnoticed long enough.

An automatic test-based validation system would not suffer from these two issues. In first place, because even if the schema is written using a formal language, the tests can be thought as stories that stakeholders could understand and propose. Those stories could later very easily be translated into test cases using a formal language close to the story description. Regarding the revision, apart from diminishing the possibility and incidence of human errors, it also offers us a method to determine when the validation has ended, therefore avoiding the two possible inconvenient scenarios derived from not knowing when the phase has finished.

1.2.3. Reverse engineering

Reverse engineering can be understood as the process of analysing a system in order to identify its components and inner relationships. In later step, a representation of the system is then created in another form or even in a higher level of abstraction.

The main objective of the reverse engineering of an information system, is to provide a deeper understanding of it. This knowledge can be aimed towards correcting it, building documentation, making it the basis for enhancements or even for a complete redesign.

1.2.4. Test-Driven Conceptual Modelling

TDCM is an innovative conceptual modelling method based in Test-Driven Development (TDD), which is popular in programming field. Its aim is to drive the elicitation and the definition of the conceptual schema of an information system iteratively. TDCM uses test cases to drive the conceptual modelling activity. In TDCM, conceptual schemas are incrementally defined and continuously validated.[21]

1.3. Project context

This project is based on the work about conceptual schema testing [20] and Test-Driven Conceptual Modelling [21] published by Albert Tort, Antoni Olivé and Maria-Ribera Sancho, which are members of the GMC[1] whose researchers mainly belong to ESSI[2] at FIB[3] - UPC BarcelonaTech[4]. Within this group, and especially in the scope of Albert Tort's doctoral thesis, there are some advancements and work done in the field of Test-Driven Development (TDD), concretely in tools for the validation of software at the conceptual modelling stage.

These tools, which will be the core for the Test-Driven Conceptual Modelling (TDCM) application are :

Conceptual Schema Testing Language (CSTL): A language specifically designed to write automated conceptual schema tests.

CSTL Processor: A Java-based prototype that makes the execution of automated conceptual schema tests possible.

[1] Conceptual Modelling of Information Systems research group. Webpage at http://guifre.lsi.upc.edu/index.html
[2] Services and Information Systems Engineering Department. Webpage at http://www.essi.upc.edu/
[3] Barcelona School of Informatics. Webpage at http://fib.upc.edu/
[4] Technical University of Catalonia. Webpage at http://www.upc.edu/

1.4. Motivation

Besides the personal motivation already mentioned in the preface, I think the development of a conceptual schema of a real system using TDCM and CSTL is a good opportunity to refine this contributions by experimentation and evaluation of its feasibility and its advantages and drawbacks.

Currently it is not widely used because the tools we have seen in section 1.3 are still too recent and have not yet been established in the community. Therefore not many conceptual schemas have been developed using them and their principles. Furthermore, the processor is a prototype, and that means that is constantly being upgraded and still has some room for improvements.

1.5. Objectives

The main purpose of the project is the development of an executable conceptual schema of a popular task management system. The development will use Test-Driven Conceptual Modelling (TDCM), a novel method for the incremental definition of conceptual schemas which uses conceptual test cases in order to drive the development. The conceptual schema will include both the structural and the behavioural aspects (events) of the system and it will be specified in Unified Modelling Language (UML) class diagrams plus Object Constraint Language (OCL) constraints.

More specifically, the project has two subgoals. These objectives are TDCM experimentation and improvement proposals to the system, and are going to be briefly explained in the following lines.

TDCM experimentation

In the first place, we intend to demonstrate the feasibility of TDCM by using it for a real-sized information system. Moreover, since the modelled system is already implemented, in this project we put in practise a reverse engineering process aimed to define the conceptual schema of the system by using TDCM.

TDCM fosters the development of correct and complete conceptual schemas according to stakeholders' needs and expectations. Additionally to the conceptual schema, by applying TDCM we also obtain a set of validation stories formalised as test cases that have been used to drive the knowledge to be included in the schema.

In TDCM, test cases are collected and used for regression testing in each iteration. The advantage of regression testing is the ability to steadily advance in the development of the conceptual model with courage about the quality of the (partial) schema achieved in each iteration. This is done by executing the collection of test

cases at each iteration, making sure that for every knowledge update that all previously tested functionalities continue to work as expected. This way, the developer can also make sure that each part of the schema is free of collateral and undesired changes, contributing to improve and guarantee a certain level of conceptual schema quality during the development.

We will keep track of the time spent in each TDCM activity in order to analyse patterns in the use of TDCM and comparing the conclusions with other existing applications of TDCM. Moreover, we will propose improvements to the methodology and the supporting tool according to our experience.

Improvement proposals to the system

Since we model an existing system, the application of conceptual modelling principles and an accurate development of the conceptual schema by testing may reveal improvement proposals to the system which may be detected at the conceptual modelling stage. We will define the conceptual schema according to the quality properties pursued by TDCM and we will keep track of the differences between the schema and the real system in order to propose improvements.

1.6. Methodology

The definition of the final conceptual schema, has followed different consecutive steps. These steps, would ideally be non-overlapping, but when the need arose some minor transgressions where made. All in all, the order, which is coherent with the Test-Driven Conceptual Modelling (TDCM) philosophy, could be resumed in this way:

1. Acquisition of knowledge of the system (Remember the Milk): this would include experimenting with it as a user, as well as consulting all the available information and documentation. The main goal in this first step is to discover all of its capabilities, and what the clients of the system could do and what could not. We approached this step with a reverse engineering attitude, not only filling the different fields with expected and unexpected data, but also trying exhaustively as many functionalities and scenarios as the system offered.
2. Familiarisation with the tools and TDCM: here I would prepare my working environment to make it a suitable place to run the CSTL processor prototype. Furthermore, I would look at available examples of CSTL code and read the available papers as well, to better comprehend the next steps I would be applying, according to the TDCM methodology.
3. Formalisation of the general use cases: after having gathered enough knowledge from the system I proceeded to formalise all its functionalities into use cases.

The process was executed in no particular order, as at this point I lacked a basis solid enough to decide which use cases were more valuable than others.

4. Application of TDCM: the information gathered from the use cases allowed the definition of a test strategy in order to maximise the overall value at any point in the iterative process, according to the Kano model[5]. This phase includes the following activities :

   a) Writing up the test stories: in this first step, the most important stories not yet tested, are written. They are meant to cover every aspect of the use case from the point of view of all the stakeholders. They will be used as a generalisation which their satisfaction would warrant the correctness and completeness of the whole use case. Therefore, exhaustivity is especially important at this point.

   b) Encode the CSTL tests: for each selected use case from the previous step, every test story is chosen and translated into the CSTL language.

   c) Construction and refinement of the schema: A further smaller scale iterative process is started for each use case.

      i. Execution of the tests. If they are all correct, continue outside the loop.

      ii. Modification of the conceptual schema. According to the messages add relationships between classes, attributes, add invariants, modify postconditions, etc. via the USE language.

      iii. Modification of the methods file. Update the actions the use case is expected to do to leave the knowledge base in a consistent state.

      iv. Repetition. Return to the step i.

   d) Conceptual schema refactoring: once this iteration is finished, the conceptual schema is analysed to see if there are some modifications that could improve its quality. If so, the schema is rectified accordingly. To ensure there is no unexpected behaviour derived from the modifications and the schema is still valid, the execution continues from step i once again. The schema is opened using the USEx tool, which permits its visualisation straight from the code version.

5. Utilisation of the schema: once the schema is finished and completed it can be utilised in further steps on the software engineering process. In our case however, it is only published in this document.

[5]More information in section 6.4

1.7. Document structure

A short summary of the different chapters that constitute this document is presented below.

Part I: Introduction and Background

Chapter 2 Background: The background presents and expands some of the concepts that are used as a basis to build this project on.

Chapter 3 Research approach: This chapter describes the research approach, making emphasis on why this project is both relevant and rigorous.

Part II: Case study

Chapter 4 Use cases: The use cases correspond to all the functionalities that make up the system, written in a general technology-independent form, as user system interactions specified in natural language.

Chapter 5 Test stories: The test stories represent the expected scenarios that should be feasible if the conceptual schema specifies the correct and relevant knowledge.

Chapter 6 Testing strategy: This chapter argues why a testing strategy is important in the context of TDCM and which model we are using to prioritise the tests. It also shows the dependencies between use cases, the approach we take at prioritisation and the final subsets of use cases and their testing order.

Chapter 7 Application of TDCM: The application of the TDCM includes the progressive translation of the test stories in natural language to Conceptual Schema Testing Language (CSTL) and its execution according to the testing strategy and the TDCM method. The writing of the UML conceptual schema, its restrictions in OCL and the methods to implement the functions necessary to pass the tests.

Part III: Results

Chapter 8 Conceptual schema: In this chapter we present the resulting conceptual schema of the Remember the Milk (RTM) system, which has been obtained from the application of TDCM.

Chapter 9 Lessons learnt: In this chapter we report the pros and cons of TDCM learnt from the practical application and RTM's perceived errors and possible enhancements that may be suggested to its community.

Chapter 10 Conclusions and further work: The project objectives are revisited, the results are summarised and further work is proposed to continue researching along the lines of this work.

Part IV: Appendixes and Bibliography

2. Background

2.1. Overview

The aim of this chapter is to deepen the notions that were already hinted in sec. 1.2 . The most relevant terms and facts about conceptual modelling are reviewed. This includes what is understood as conceptual schema quality and how automatic testings can contribute to its validation. There we also justify the *necessity principle of conceptual schemas[16]* that requires the definition of a conceptual schema for any possible information system. At sec. 2.3 we define what a task management system is and why we chose the Remember the Milk (RTM) system over all other candidates.

2.2. Conceptual modelling

2.2.1. Information systems

In the definition of conceptual modelling at sec. 1.2 we talked about the development of information systems.

The term is defined as follows[17] :

> Information systems are implemented within an organization for the purpose of improving the effectiveness and efficiency of that organization. Capabilities of the information system and characteristics of the organization, its work systems, its people, and its development and implementation methodologies together determine the extent to which that purpose is achieved.

The above definition, talks about the reason to be of an information system. It talks about the capabilities influencing the degree of achievement of its goals, which might be varied according to the necessities it has to fulfill. Nevertheless, in reference to its characteristics, we can identify at least three main common functions that all information systems must typically implement[15], which are:

Memory: To maintain a consistent representation of the state of a domain.

Informative: To provide information about the state of a domain.

Active: To perform actions that change the state of a domain.

An information system not only stores but also interprets the information (as inputs) it deals with and updates (as outputs). This information corresponds with the knowledge of the specific domain the system is suited for.

2.2.1.1. Domain

The domain is the world the information system knows and where all its reality takes place. It is a set of real objects (not a representation of them) with all their relationships. To most information systems, the domain is simply the organization that holds them, but it could be almost everything, ranging from a vehicle and its environment to something as abstract as a chess game.

2.2.1.2. Information base

The concept of domain and information base is quite close, but should not be confused. The main difference between them is that the information base is the representation of the entities of a specific domain, the relationships between these entities and their type classification. Being a representation, means that the information base does not exist physically. Therefore it is only an abstract description that illustrates a concrete domain state.

2.2.1.3. Conceptual schema

One definition of the term could be[16] :

> A conceptual schema defines the general knowledge required by an information system to perform its functions.

Hence, a conceptual schema could be understood as a metadomain. It describes the concepts utilised to describe a concrete domain. Those concepts are the result of the creation of a human mind (the group of engineers that envision the system) and allow the classification of the objects and their real relationships. This means that for a specific domain there might be different valid conceptual schemas that represent their knowledge. In the next subsection, when we talk about conceptual schema quality, we will see this in more detail.

In the scope of a concrete information system, its conceptual schema describes and defines only the knowledge subset of concepts (and their constraints) from its domain that are relevant to its functions and which are represented within that information system.

2.2.1.4. The conceptual model

What we understand as a conceptual model is how a domain is modelled (conceptualised). The assumption that this conceptualisation of a domain is formed by

entities and their relationships, is one possible way to approach it. Another valid way for the same domain, might be to conceive and model it as a set of facts and logical propositions.

The term conceptual model is used quite flexibly -sometimes not as consistently as one would desire- as a synonym of conceptual schema. Another common usage of it is to refer to the sum of the conceptual schema plus the information base.

In order to express a conceptual schema, a formal modelling language needs to be defined. The Unified Modelling Language (UML) from OMG[1]first developed in 1997 has been chosen for this task in this project, but because it is so extensive, only the subset concerning class diagrams and the Object Constraint Language (OCL) will be used.

The decision to choose UML was taken for its many advantages. In the first place, it has long been the industry standard. It is actively used and supported by many of the current leading software companies. Furthermore, because it is a language specifically designed for this task, it covers all the different stages from stakeholder interaction with the system, to the final class layout design, obviously going through the conceptual schema which it is perfectly suited to define. This means that if the resulting schema was to be implemented, it would be readily available to be used in the following steps, like design.

2.2.2. Quality in conceptual schemas

Revisiting the definition given at sec. 1.2.2, we saw that according to the conceptual modelling quality framework proposed[13], a conceptual schema of an information system has semantic quality when it is valid and complete, meaning validity that the schema is correct and relevant.

An expanded vision of the quality of a conceptual schema, would be the degree in which it has some specific properties that let it accomplish its functions effectively[13]. These characteristics can be described as :

Completeness: a conceptual schema must know all the aspects of the domain it describes that are relevant for the information system. As long as there is some knowledge not described in the conceptual schema, the corresponding information system will not be able to function properly.

Correctness: the knowledge a conceptual schema defines must be true and relevant to the functions the system has to carry out.

Relevance : the knowledge a conceptual schema defines is necessary for the system.

[1]Object Management Group, a consortium founded in 1989, originally aimed at setting standards for distributed object-oriented systems, and now focused on modelling (programs, systems and business processes) and model-based standards.

Syntactic correctness: a conceptual schema must not violate any of the rules of the language used to represent it.

Understandability: all the actors affected by a conceptual schema, must be able to correctly understand the part of it that is relevant for them.

Simplicity: this characteristic is linked in a way with understandability. Those conceptual schemas that express the same knowledge using less constructions or simpler ones are preferable, because they tend to be more comprehensible.

Conceptualisation principle[7]: a conceptual schema should only include conceptually relevant aspects, both static and dynamic, of the universe of discourse (UoD[2]), thus excluding all aspects of (external or internal) data representation, physical data organisation and access as well as aspects of particular external user representation such as message formats, data structures, etc."

The specification of an executable conceptual schema, makes the validation of syntactic correctness possible. Thus, giving the capability of execution to a conceptual schema is not only a necessary step towards the application of the TDCM, but it can also be seen as a goal by itself, as it helps achieving an overall conceptual schema quality. On the other hand, the execution of automatic tests in the context of TDCM seeks the validation of the completeness and correctness properties. These and other aspects will be discussed further at chapter 7.

2.2.3. Principle of necessity

This principle is succinctly announced as [16]:

> To develop an information system it is necessary to define its conceptual schema.

Small systems might not strictly require its conceptual schema to be formally defined, but in any case it must exist, even if it is only on the mind of the developer. The non-existence of the conceptual schema is not conceivable, because any operation an information system performs has to be defined at least in a way that determines which is the outcome of it, and what it has to do with its parameters.

An analogy of this principle with that of other engineering or technical fields, would be the fact that any building or piece of machinery needs a blueprint or some form of schematics. Even if they are not always are written down, engineers must envision the end result and its parts in order to proceed, so in a way, the schematics still exist, even if not physically but in the mind of them.

Having reached the conclusion that every system needs a conceptual schema, the most reasonable next step would be to write it down, preferably formalised. Making the schema explicit forces the engineer/s to analyse it more thoroughly and therefore

[2] i.e. domain

increase the confidence that the representation is solid and what was sought. It is also desirable because it can be used in the context of Requirements Engineering (RE) to ensure all stakeholders have their needs met. Future users need to know what the system will offer them and transmit the developers their expectations so it can be negotiated the features the system will present in a clear and concise manner. Furthermore, it avoids potential misunderstandings, being every relationship fixed, and it keeps on sight all the interactions between the main entities.

2.3. Task management systems

2.3.1. Characteristics

Current task management systems, can be understood as evolved and more complex versions of time management systems, which at the same time are successors of the simple pen and paper to-do lists and agendas. Their aim, like its physical counterparts, is to help the user better organize his/her tasks, in a way no one is forgotten. It can also provide some further valuable information whose objective is to save time by having information handy like notes and contact details. Those systems however, typically go beyond the traditional functionalities, by letting the tasks be shared or assigned among contacts, be prioritized or even by sending customisable periodic reminders.

Their chronological evolution, along with the expanding number of typical functionalities was resumed as follows [6] :

First generation: reminders based on clocks and watches, but with computer implementation possible; can be used to alert a person when a task is to be done.

Second generation: planning and preparation based on calendar and appointment books; includes setting goals.

Third generation: planning, prioritising, controlling (using a personal organiser, other paper-based objects, or computer or PDA-based systems) activities on a daily basis. This approach implies spending some time in clarifying values and priorities.

Forth generation: being efficient and proactive using any of the above tools; places goals and roles as the controlling element of the system and favours importance over urgency.

2.3.2. Our case study: Remember the Milk

The task/time management system we have chosen to study for this project is RTM. After some years, it is still very relevant in its segment today and it is a decent standpoint to compare any competitors in the market.

Some of the reasons that led us into choosing it are:

Long experience in the field: the development of this product dates from August 2004 and it launched on October of 2005.

Proven relevance: as of November 2011 it has more than 3.1 million users.

Potential: it has shown a steady growth over time, from 2 million users as of April 2011 to 2.5 million as of July 2011 to 3.1 million as of November 2011.

Cloud-based: it is hosted on the web, accessible anywhere at any time, it does not need any installation and it brings all the other advantages typical from cloud-based software.

Rich feature set: even having a straightforward interface and simplistic nature, the system provides a wide enough set of features that can make up for a good amount of different use cases.

Available for free: meaning that everyone can try it out without the need to spend any money to verify the results of the project, or the functionalities of each use case.

Previous knowledge: we had previously worked with this system, and we knew it would be suitable for the project.

3. Research approach

3.1. Overview

This chapter aims to explain how the research was approached. In section 3.2 we talk about the specific methodology used, how it works, why it was chosen and how their goals are achieved in the project's context. At section 3.3 we summarise how the project was conceived and divided into stages from the very beginning.

3.2. Design science research

The Design Science Research[9](DSR) is the framework of the research that we will use along all the project.

3.2.1. Definition

The DSR, besides being proposed somewhat recently (2004), it is a widely used research approach spanning hundreds of papers and articles. The DSR is a problem-solving paradigm based on the creation and evaluation of artifacts intended to gain knowledge about a problem domain in order to propose a solution for some identified organisational problems. In other words, we could define its core mission as the development of general knowledge which can be used by professionals in the field in question to design solutions to their specific problems. Figure 3.1 illustrates it.

There are also seven guidelines proposed to drive Design Science Research[9] (listed alphabetically) :

Communication of research: Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Design as a search process: The search for an effective artifact requires utilising available means to reach desired ends while satisfying laws in the problem environment.

Design as an artifact: Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.

Design evaluation: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.

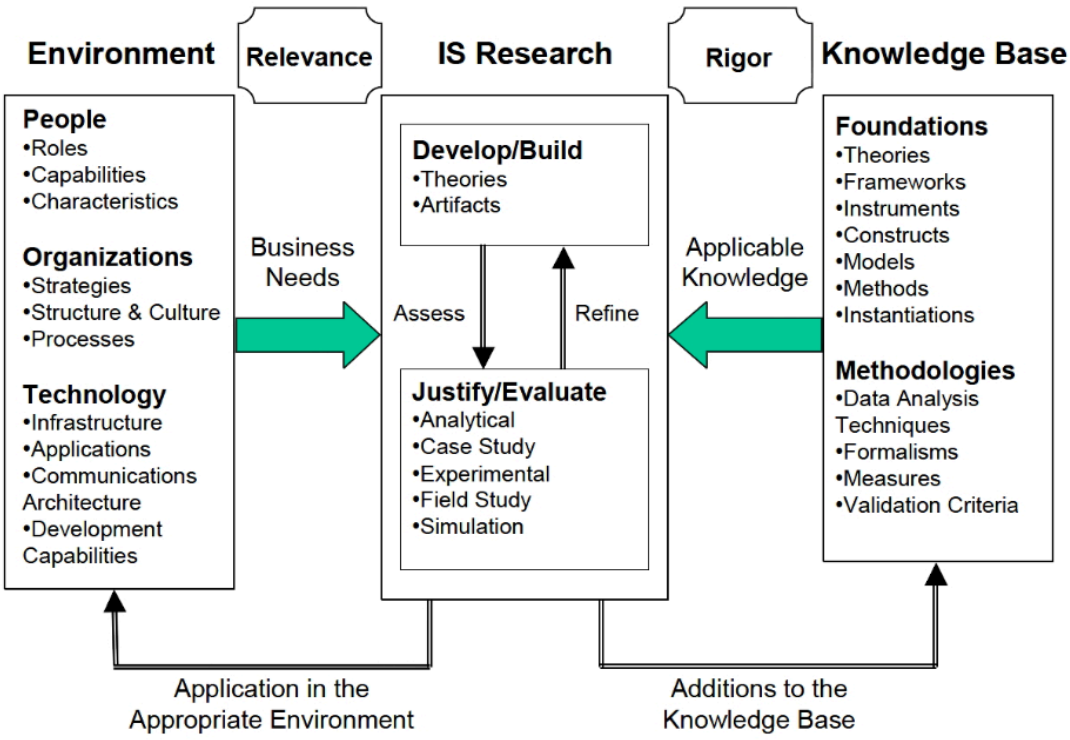

Figure 3.1.: Information Systems Research Framework [9]

Problem relevance: The objective of design-science research is to develop technology-based solutions to important and relevant business problems.

Research contributions: Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.

Research rigour: Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.

3.2.2. Application

Having described what the DSR consists on, we are ready to state the contributions of this project in its context. In order to do so, we will use the recommended guidelines to support and explain to which degree they are fulfilled. Because the order is irrelevant, they are dealt alphabetically.

Communication of research: The language of this project has been English because traditionally it is by far the most popular language for the research topics on Computer Science and Software Engineering. This project will be made publicly available free of charge to anyone according to the guidelines

and policies of the Barcelona Faculty of Informatics (FIB) at Biblioteca Rector Gabriel Ferraté (BRGF) library. It will also be given to the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) where they will be able to made it publicly available to any of their many libraries. Furthermore, I will fully grant my authorship rights to Albert Tort so he may publish it in his web portal about Test-Driven Conceptual Modelling (TDCM) along the other related works.

Design as a search process: The TDCM architecture is iterative, so in order to build the main artifact (Remember the Milk's (RTM) conceptual schema), we incrementally built it (chapter 6 and chapter 7) and analysed its results each step to obtain the necessary feedback to drive the research process and verify its progress.

Design as an artifact: This project contributes to the revision and enhancement of TDCM and the Conceptual Schema Testing Language (CSTL) Processor prototype (chapter 10). Complementary artifacts would be the RTM conceptual schema (chapter 8) formally presented as a Unified Modelling Language (UML) + Object Constraint Language (OCL) class diagram and its representation, its functions and tests written in CSTL (chapter 7).

Design evaluation: The work is evaluated with the resulting artifacts. The conceptual schema (chapter 8) is proven to be functionally equivalent to that of the RTM system, thus proving the TDCM a feasible methodology.

Problem relevance: Conceptual schema validation (the original problem that TDCM addresses) in Information Systems development is a common present-day problem.

Research contributions: Some preliminary experiences have been conducted by applying conceptual schema testing and TDCM. These experiences need to be extended with more experimentation in real-sized systems. This project contributes to TDCM experimentation with the development of the conceptual schema of a well known task management system.

Research rigour: The contributions of this project are based both on long proven approaches (e.g. the Kano model sec. 6.4) and on the state of the art related to conceptual schema validation by testing (chapter 7).

3.3. Project management plan

Along the duration of the project, weekly and bimonthly meetings were arranged between the author and the directors in order to keep track of the progress and to make some calendar adjustments when required. Prior to the thesis' initial phase, some milestones were discussed and agreed upon between both parties.

The resulting stages, were very similar to those seen in section 1.6, plus two consecutive more, which are presented here.

Analysis of results

This stage consisted on the elaboration of conclusions upon the results obtained through TDCM experimentation. Another task was to think of further rigorous and relevant work which could continue from where this project left.

Documentation

Writing and editing this document to its final form, while drawing figures and looking for appropriate bibliography, was the main concern of this stage. However, it was discussed that if there was to be an oral defence or presentation, it could have been interesting for at least one of each type of scenario to use video with actual footage from RTM. These visual aids would showcase the real system's use cases steps being "executed" as a support for the CSTL test code.

Part II.

Case study

4. Use cases

4.1. Overview

The Remember the Milk (RTM) system is by far too complex to be analysed as a single entity. One sensible solution to this problem that naturally arises, is to split it into smaller independent parts. Ideally, these reduced set of parts which constitute the system, each provide a coherent set of functionalities in a controlled environment. This is the context where different actors (stakeholders) interact with them, usually to achieve some specific business goals. Each of these functionalities is what a use case represents, and this is why we are going to use them in the context of this project.

This chapter aims to show the 65 use cases that form the RTM system. In section 4.2 we talk about what a use case is and which elements constitute it. In section 7.4 we talk about the possible differences between the real life RTM's system use cases and the ones written in this project. We also delve into the motivation behind this differences. At section 4.4 we offer a sample use case definition template where we describe its elements and parts. Finally, at section 4.5, we define the system's use cases with the previously shown template.

4.2. Definition

A short description of the term would be [16]:

> A use case is a set of actions performed by a system which yields an observable result that is typically of value for one or more actors of the system.

Now that we know what a use case it is, we need to justify why they are needed. Again, we can see that use cases address a real problem through this concise definition [16]:

> The functionality provided by an information system is too large to be analysed as a single unit. We need a means of partitioning that functionality into smaller, more manageable pieces. The concept of a use case is very useful for this purpose.

Its purpose, which could be inferred from the above definition, would be [16]:

> Use cases define the functionality provided by an information system.

4.2.1. Elements

Now that we know the meaning of a use case, we are ready to discuss its necessary elements. The bare minimum are actors and name, and all use cases must have at least these. The other elements may be present or not, but a lot of them are common and certainly useful. The following list is not meant to be extensive, but to cover the most habitual ones.

4.2.1.1. Actors

Resorting again to Olivé's definition[16] :

> In the field of information systems, an actor is a role played by a physical entity that interacts with an information system. The physical entity may be a person, an organisation, or another system. A single physical entity may play any number of different roles in the same system and, conversely, a given actor can be played by several different entities.

Furthermore, roles can be conceived into belonging to hierarchies. This would mean that some of them could be understood as more specialised or more generic versions of others. The implications this would have is that different more specific roles could still represent its "parent" role in a use case, thus avoiding the redundancy (a known anti-pattern) of having to create some virtually exact use cases in many different situations.

In the more specific context of use cases, an actor is some entity that has a direct involvement in the execution of that use case. Each use case must have at least one of what is known as a primary actor. What makes this actor special from the others, is that this one has at least a goal which can be satisfied by its execution. The primary actor is also usually the one who initiates the interaction with the system.

4.2.1.2. Name

Every use case must have a unique name that identifies it. Ideally, this name should be a description of what are the actions that take place during the said use case and/or what is its aim and what is it going to be achieved in the point of view of its primary actor. It is also preferable, in order to have a good legibility, to choose short use case names over long ones.

4.2.1.3. Preconditions

Preconditions are statements made that are assumed true at the beginning of the use case. No use case can be executed if their preconditions are not met, as it could not

guarantee that its behaviour would be the one described. In any case, preconditions should respond to situations that would not make sense in any use case scenario. Other situations that could lead to error but their occurrence is plausible, should be taken into account, for example with addition of use case extensions.

4.2.1.4. Trigger

Its function is defined as [16]:

> The trigger section describes the starting condition that causes the initiation of the use case.

4.2.1.5. Main success scenario

According to [16]:

> The main success scenario describes the basic flow of a successful scenario. It is written as a sequence of action steps, but the steps can be executed in parallel or in a different order or can even be repeated. It is recommended that the steps should be numbered. An action step may be an interaction between two actors; a validation, performed usually by the system; or an internal state change.

In our case study, however, we are somewhat more strict in its properties, to keep the things simple for the reader. Each of the steps of our main scenarios are meant to be executed sequentially, in the written order and without any repetitions. They are also numbered, as recommended.

4.2.1.6. Extensions

From [16]:

> The extensions section defines alternate flows when some condition is satisfied. An extension is related to an action step of the main success scenario, and has two parts: a condition and a sequence of action steps. An extension can be seen as a miniature use case that may be triggered when the main success scenario is at the step indicated and the condition is satisfied. At the end of the extension, by default the scenario merges back with the main scenario, but it can end with the failure of the whole use case.

4.2.2. Interactions

Basically, any use case can interact with any other number of use cases in three different ways. Those ways have the names of inclusions, extensions and specialisa-

tion/generalisation. In this project we will only be using inclusions and this is why we will not explain the other ones.

They have been defined as[16]:

> An include relationship from a base use case to an inclusion use case means that the behaviour defined in the inclusion use case is included in the behaviour of the base use case. This is useful for extracting common behaviours from several use cases into a single description.

4.3. Limitations

4.3.1. Scope

One of the main limitations on this project related to use cases has been its scope. Many of the 65 RTM's use cases design decisions could be explained in a high degree of detail, because there has been a lot of thought and effort put into them. Unfortunately, this would make for a much more lengthy document, certainly more than it would be reasonable in the context of a 30 ECTS credits project. This is why most of the use cases' specification will be presented in Appendix A without any additional explanation besides themselves.

4.3.2. Existing system

As has been discussed earlier, the starting point for the conceptual schema is a system that already exists. This means that some decisions, if we are to proceed with a reverse engineering approach, whether we like it or not, have been taken beforehand. This decisions, in some cases, may somehow limit the liberty of action of the author of this text and led to suboptimal solutions in order to make the specification of the system compliant with the one it represents (RTM).

One such example would be, in the context of contacts and groups section A.5, the logical possibility of updating them, following the standard Create, Read, Update and Delete (CRUD) philosophy. For example, an end-user might want to add (and consequently update) some information regarding one of his/her groups of contacts besides its members or related to its tasks. One such piece of useful information would be the name of the mailing list for that group. But because in the RTM system this functionality is not taken into account, we can not include it in the specification as it would mean modifying too significantly the offered functionalities.

On other cases, however, when the missing information or behaviour would lead to an erroneous state and/or does not imply modifying substantially, corrections have been made. One such example would be introduced data validation, which is not applied in some needed cases in the RTM system. Again this fact is more visible and further discussed in chapter 5 and chapter 7 .

4.3.3. Technology independence

In the context of this project, we focus on the specification of essential use cases.

> Essential use cases are technology-free and implementation-independent, keeping the interface out and focusing on the actor's intent. [16]

The main reason for this, is because we do not want to bind the system to any specific technology before building it. This would be unnecessarily limiting its reengineering potential and preventing some of the most important benefits of reverse engineering. This fact may make some concrete use cases look as if they did not correspond exactly with the RTM functionality they represent. In those cases, is because the use case specified here is more generic, and hence it represents the RTM way of accomplishing together some possible additional ones.

4.4. Template

In this section, we are showing how does the use case template we will be seeing in the next section looks like.

Use case name

Scope: the scope of the use case, in this project the RTM system.

Primary actor: the use case's primary actor, in most cases a end-user.

Preconditions: here preconditions necessary for use case execution will be stated, together with information to other sources when needed.

Includes: here the other included use cases will be linked. In most cases there will not be any.

Trigger: the action that marks the beginning of the use case.

Main success scenario: an ordered list of the use case steps is shown for the most usual scenario where the primary actor goal is achieved.

Extensions: in case they exist, they mark alternative paths for the main success scenario. They are written in a way that their step number is the same as the one from the main success scenario they replace. At this point, the alternative scenario is executed because the conditions are suitable for it and not for the main scenario. The main scenario stops executing at that numbered instruction and instead the first instruction from the alternate scenario is executed. The next instruction, in case it exists, is the consecutive one from the following scenario, and its subsequent until the alternate scenario is over. Once it is over, if it is not made explicit the end of the use case or the return to a specific position from the main scenario, by default the execution will continue from the place on the main scenario it jumped to the alternate one.

4.5. RTM use case specification

In this section, we use an example of the full specification of a relevant use case of RTM. The template used is the one seen in sec. 4.4 . In this project we specified 65 use cases whose specification is available in Appendix A.

4.5.1. Update priority from a task

4.5.1.1. Use case specification

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to change the priority from an existing task.

Main success scenario:

1. The actor reads a task[1].
2. The actor indicates the system that he/she wants to modify the priority of task he/she has read in the previous step.
3. The system provides the actor with a representation (e.g. a list or a colour schema) which contains all possible priorities, modifications and the current value.
4. The actor selects one priority from the representation.
5. The system temporarily sets the task's priority to the chosen level.
6. The system validates the introduced data.
7. The system sets a priority for the task according to the temporary value.
8. The system notifies the actor that the task's priority has been successfully updated.

Extensions:

4a. The actor wants to increase the current priority of the task:

1. The actor indicates the system that he/she wants to increase the priority of the task.
2. The system temporarily increments the task's priority by one level.

[1]See sec. A.1.2

3. The execution returns to item 6 of the main success scenario.

4b. The actor wants to decrease the current priority of the task:

1. The actor indicates the system that he/she wants to decrease the priority of the task.
2. The system temporarily decrements the task's priority by one level.
3. The execution returns to item 6 of the main success scenario.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

4.5.1.2. Additional notes

One detail distinctive of this use case is its two extension points on step 4. This is because the update of a priority, in the context of RTM could really be understood as three different use cases. Those would be Increase priority, Decrease priority and Replace priority, whose names are quite self-explanatory. But because conceptually they are all very close (they are updating the priority) and we want to avoid unnecessary code repetition and use case cluttering, we decided to combine them together in one single more general use case.

Differences in the user interface or in the technology for this very same use case can be better seen in subsection 5.5.1, but here they are already hinted. For example, in item 3, we can see how we do not address priority level in one specific magnitude. At this stage we leave it open to different implementations, which might seem obvious once seen, but they tend not always to be considered. In this situation, one priority level could be represented simply as numbers (RTM choice), but also other valid representations could be colours, or even categories like "High", "Medium" or "Low" (probably implemented as an enumeration type) and so on.

We also see that it includes one other use case case. This means that its functionality is already partially implemented in another use case, and that its completion must precede the implementation of this one, as there is a dependance relationship. This way, in the long term, we will avoid code repetition which in turn will diminish the effort put into code maintenance, both good practices

Another dependance, even if it is not present as an inclusion, would be the sec. A.2.2 log in use case. This is because the precondition for this use case execution is to have the main actor logged in. This means that this other use case must be implemented as well before this one.

5. Test stories

5.1. Overview

The aim of this chapter is to describe what a test story is and which ones we decided to create. In section 5.2 we explain what a test story is about, which includes its purpose and its context. In section 5.3 we talk about the limitations the test story approach has and how they affect the Remember the Milk's (RTM) system use cases. In section 5.4 we analyse which are the dependencies of each use case the RTM system has, and what this has to do with the test stories. Finally, in section 5.5, we define the test stories that we are using to test the use cases defined in Appendix A.

5.2. Definition

In our context, test stories can be seen as representations of specific use case scenarios, which would represent normal real-life usage of the system. The particularity about them, is that they are written in a way to accomplish at least one single test objective. These objectives, are the main purpose of test stories. They typically are the complete execution of a use case scenario from its beginning to its end. This could be either the main success scenario or any of its possible extensions

One characteristic of these test objectives is that, ideally, they are non-overlapping with the goals from other test stories. This is because, if we are confident that a test story faithfully represents its objective, it would just be redundant, and hence undesirable and inefficient to test it again with another story. Because there can be dependencies over use cases, as seen in sec. 5.4, some test stories simply need that other ones are verified before, as they build upon. This fact makes necessary a test strategy (chapter 6), to have a balanced workload regarding the number of written tests, their value and the effort put.

5.3. Limitations

Some of the limitations of this section may not be dissimilar to those found in sec. 7.4. After all, the project scope and the system's conceptual schema to be elaborated are the same, and the test-story stage is strongly linked to that of use case specification. We refer them in the next subsections.

5.3.1. Scope

Similarly to what happened with use case specification, one of the main limitations of this project in relation with test stories has been its scope. We said before that the RTM system has many use cases (65). This is even more true with the test stories, as each use case typically makes for at least two of them. They could all be analysed in detail, and explain the decisions that make each of them suitable for their objectives, but unfortunately, this would make for a much more lengthy document, surpassing what would be reasonable for a single semester project. This is why most of the test stories are presented in Appendix B without any additional explanation besides themselves, even if they have been carefully crafted as to cover all the reasonable scenarios.

5.3.2. Interface

These test-stories are meant to be representations of real life usage by end-users. As such, some form of interface must be provided or assumed, as end-users would need one in order to interact with the final system. In some cases, this interface is not somehow compatible with the more general technology independent approach made in the previous chapter. When this situation occurs, the chosen interface will be the one closest to the original RTM system, because after all we are modelling the schema after it.

5.3.3. Completeness

There are some extensions belonging to specified use cases that are rather open-ended. The most obvious example for this, are those extensions that have to deal with errors and validation of input. It is clear that one single test story can not possibly test all the different errors. It is also not reasonable for one single test story to test all of the different types of errors either. In this situation, probably a test story should be written representative for each validation type error. Because we are somehow already limited in the length of this document, we will not be testing exhaustively all possible input errors, but just some of them. In any case we will still hold the premise that each possible use case extension should be covered at least by one test story.

5.4. RTM story dependencies

The testing strategy we define in chapter 6 requires analysing the dependencies between the defined stories.

5.4.1. Context

5.4.1.1. Definition

In our context, we say that one use case is dependant on another one, when the first use case needs partially or totally the functionality of the second one in order to successfully execute all its possible scenarios. Hence, story dependencies are based on use case dependencies as well. A test story is dependant on another one when it has event occurrences which are the main test objective of another test story.

5.4.1.2. Motivation

We have seen along the specification in sec. 4.5 that the number of unique use cases the system consists on is 65. In this section we are going to show that most of these use cases have numerous dependencies. We are also going to reveal which are those dependencies they have on an individual basis.

Despite the information presented in this section should not be novel to the reader (i.e. it is already implicit in the use cases themselves), we think it is noteworthy because it helps to understand the system as a whole and grasp its internal relations and dependencies in a straightforward manner. High dimensionality along with the inconvenience of having to manually examine each of the individual use case's possible scenarios and the fact that some dependencies are non-obvious, could hinder the overall understanding or at least deter the needed insight to visualise the dependencies.

5.4.1.3. Graphical representation

The use cases dependencies are presented here in the form of directed graphs. These are the parts that constitute them :

1. Nodes: each node represents a single use case. The name on a node is usually the same one as its use case or a close abbreviation. Depending on their colour, each node has additional meaning:

    a) White: these nodes represent the use cases belonging to the thematic grouping being analysed. Each thematic grouping is announced in the figure name.

    b) Dark gray: these nodes represent the use cases that other white nodes depend upon. The main difference between white nodes that also have dependant use cases and dark gray ones, is that the later would not belong to the thematic grouping, and are just added because they are needed despite being from different groupings.

2. Edges: each edge going from one node to another node represents that the first use case has at least one dependency with the second one. We can also notice that there are no mutually dependant use cases, so each edge has one and only one arrow end. Similarly to what happened with nodes, each colour brings additional meaning.

    a) Light gray: these edges represent some of the dependencies that nodes outside the thematic grouping have with other nodes.

    b) Dark gray: these edges represent the dependencies that dark gray nodes have with other nodes.

    c) Black: these edges represent the dependencies that white nodes have with other nodes.

3. Groups of nodes: to make it easier to discern, nodes have been grouped into "boxes" which contain a small subset of highly cohesive nodes. This further layer of abstraction should help visualisation and accentuate particular existent relationships between nodes. They also have different colour representations :

    a) Light gray: they represent the thematic grouping.

    b) Dark gray: they represent clusters of nodes that are highly cohesive. They also have a tag on them to make this relation more evident.

In order to improve the graph's readability, some of the nodes that depart or arrive at the same node have been simplified and unified. In some cases, this makes only one edge colour visible, but it should not seriously affect the overall understanding.

5.4.2. Groupings nomenclature

In the following list, the chosen nomenclature is explained for each of the groupings. *The order in which the following groupings are presented is arbitrary.*

CRUD task: Stands for Create, Read, Update and Delete a task. It includes those 4 use cases.

Share/send task & contacts: It makes reference to all use cases involved in sharing and sending a task to contacts.

Share/send task & groups: It makes reference to all use cases involved in sharing and sending a task to groups.

Show tasks: It only involves the use case this very same name.

CRUD note: Stands for Create, Read, Update and Delete a note from a task. It includes those 4 use cases with equivalent functionalities.

Task completion: It makes reference to all use cases involved in completing and uncompleting a task.

Postpone task: It only involves the use case this very same name.

Duplicate task: It only involves the use case this very same name.

CRUD priority: Stands for Create, Read, Update and Delete a priority from a task. It includes those 3 use cases with equivalent functionalities.

CRUD account: Stands for Create, Read, Update and Delete an account. It includes those 4 use cases with equivalent functionalities and also the use cases related to logging in and out of an account.

CRUD reminder: Stands for Create, Read, Update and Delete a reminder schedule. It includes those 4 use cases and also the use case related to sending reminders.

CRUD list: Stands for Create, Read, Update and Delete a list of tasks. It includes those 4 use cases with equivalent functionalities and also the use case related to moving tasks to a list.

Default list: It makes reference to all use cases involved in setting and unsetting a default list.

Share/publish list & contacts: It makes reference to all use cases involved in sharing and publishing a list for contacts.

Share/publish list & groups: It makes reference to all use cases involved in sharing and publishing a list for groups.

Publish list_anyone: It makes reference to all use cases involved in the publication and unpublication of a list to anyone.

Archive list: It makes reference to all use cases involved in archiving and unarchiving a list.

CRUD contact: Stands for Create, Read, Update and Delete a contact. It includes those 3 use cases with equivalent functionalities.

CRUD note: Stands for Create, Read, Update and Delete a group of contacts. It includes those 3 use cases with equivalent functionalities and also the use cases related to moving a contact to a group.

Weekly planner: It only involves the use case this very same name.

Change schedule: It only involves the use case this very same name.

CRUD location: Stands for Create, Read, Update and Delete a location. It includes those 4 use cases.

Default location: It makes reference to all use cases involved in setting and unsetting a default location.

5.4.3. Graphical representation

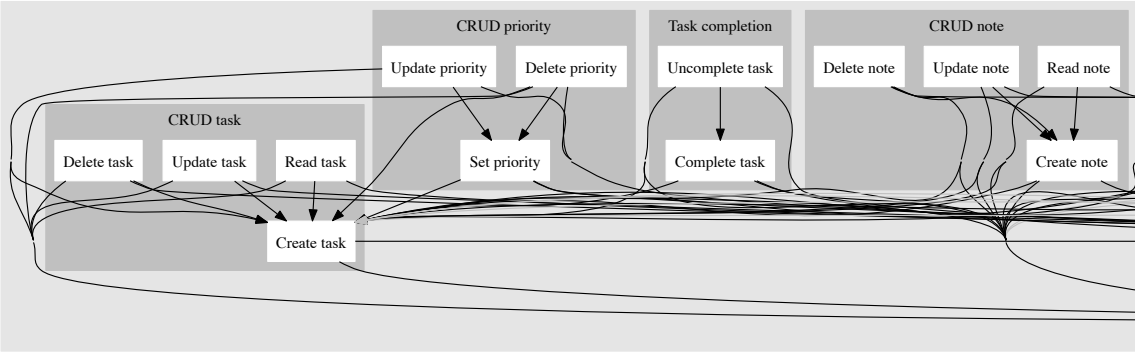

Figure 5.1.: Task dependency tree (leftmost part)

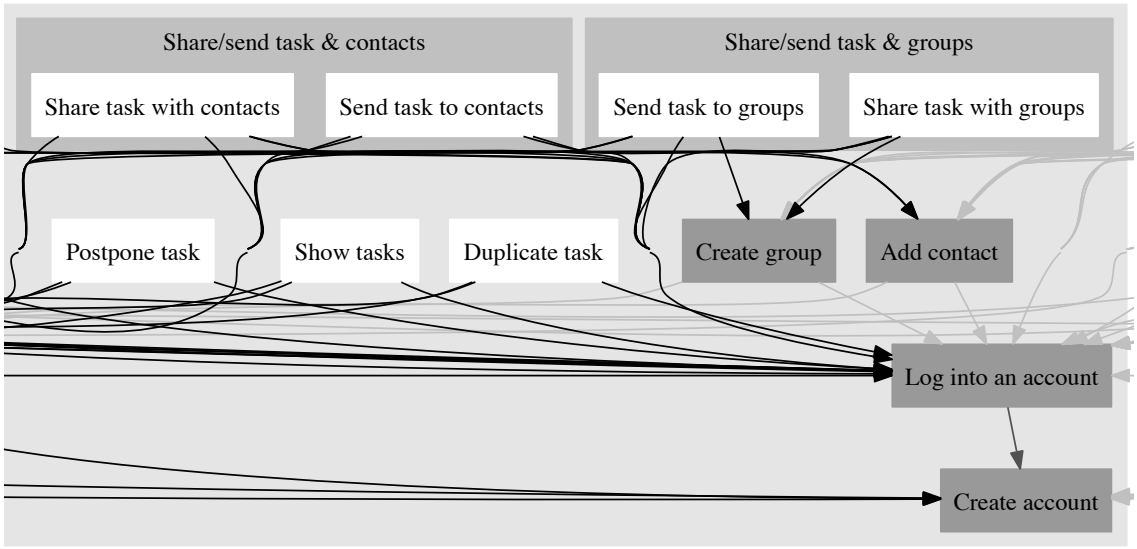

Figure 5.2.: Task dependency tree (rightmost part)

5.4.4. Graphical summary

In this section we are presenting with the aid of Fig. 5.9 all the system's dependencies. In Figure 5.9 we can see that most of the graph nodes do not precisely match the name of the specified use cases in section 4.5. This is the case because in order to provide a better understanding we grouped together use cases that were conceptually close, linked to one specific functionality. The names used are those mentioned in subsection 5.4.2.

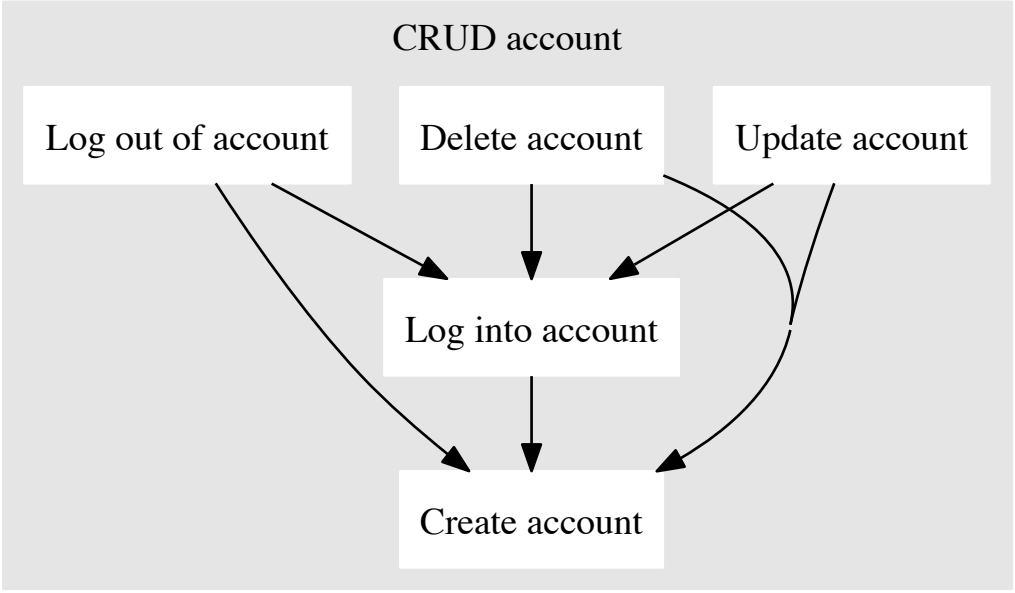

Figure 5.3.: Account dependency tree

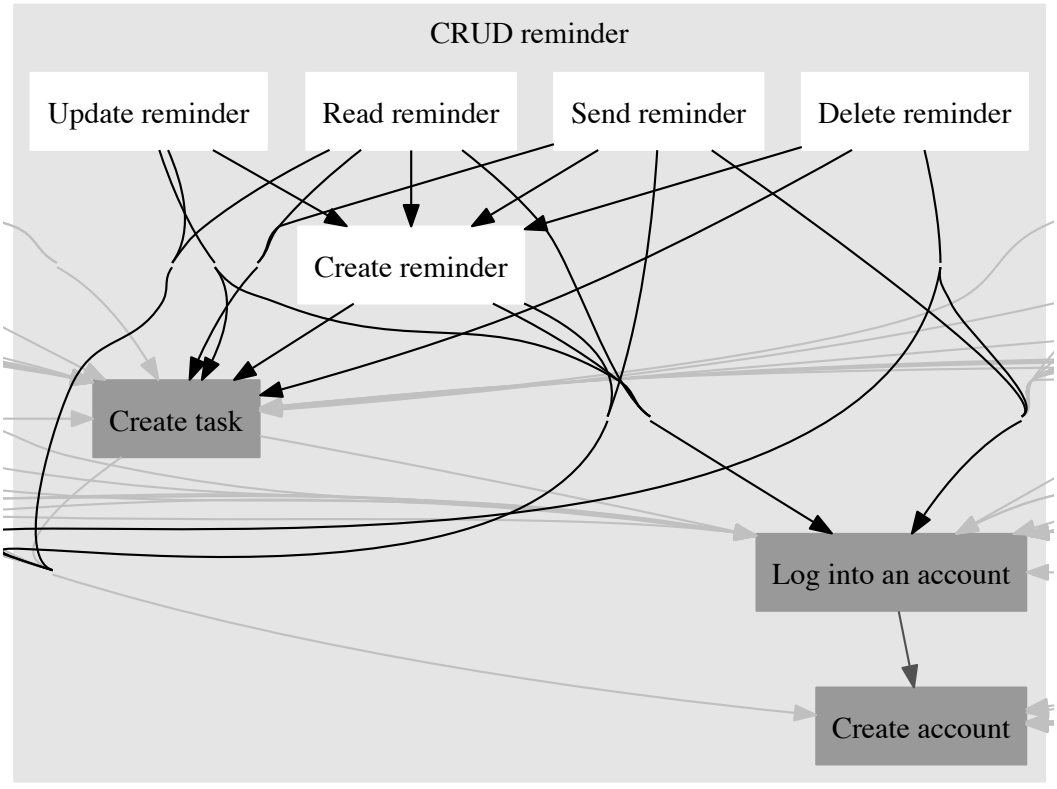

Figure 5.4.: Reminder dependency tree

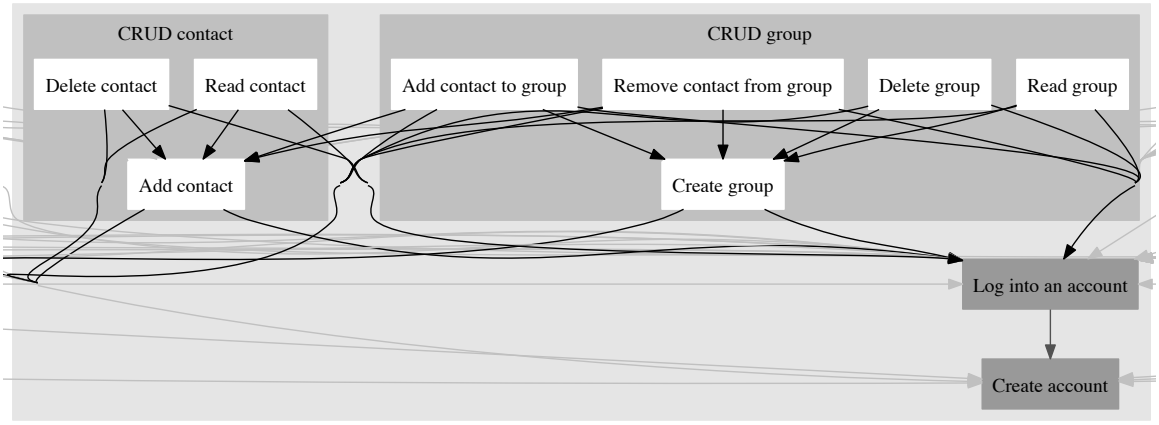

Figure 5.5.: Contact dependency tree

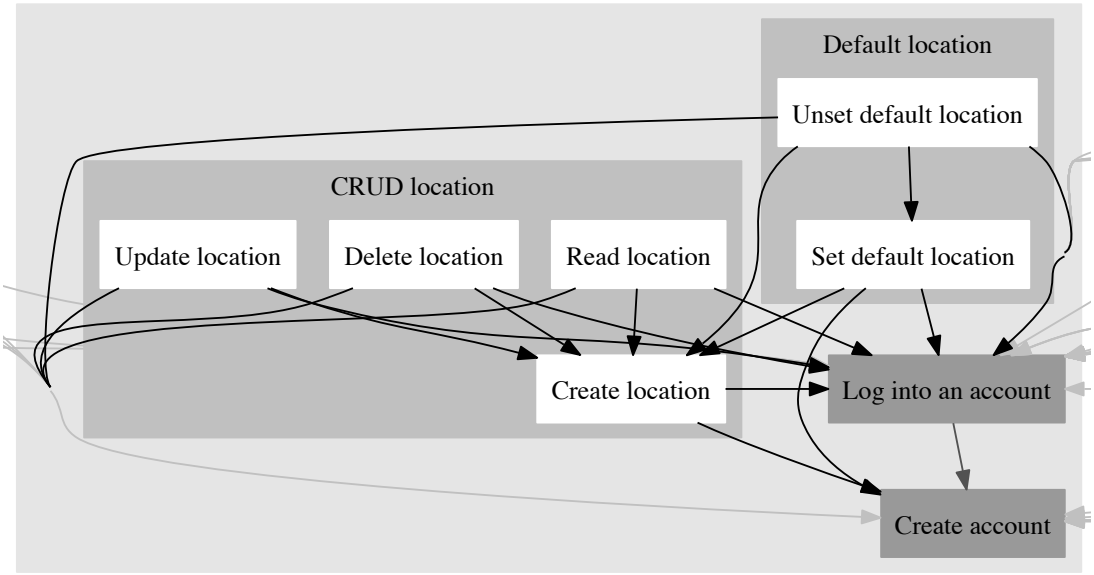

Figure 5.6.: Location dependency tree

5.5. RTM Test stories

In this section, we use an example of the full set of test stories of a relevant use case of RTM, the same from section 4.5. All this project's test-stories are available in Appendix B.

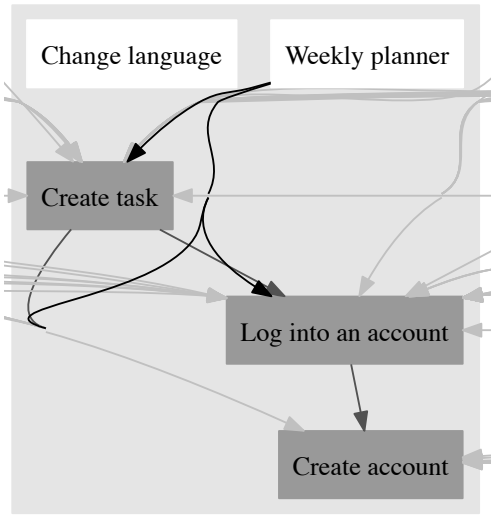

Figure 5.7.: Planner and language dependency tree

5.5.1. Update priority from a task

5.5.1.1. Test story 1: Modification of a priority from a task

Test objective: Update a priority from a task (Main success scenario sec. 4.5.1.1).

Alice wants to update the priority from a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to update the priority of the task. The system provides her with an interface with the current priority value that she can modify, along with a caption which instructs the allowed range values. In this case, the priority is the highest possible, but in case it did not have any assigned one it would simply be blank.

Alice chooses this time the second highest priority for the task. The system informs Alice that the priority has been modified successfully for the task "Go to the cinema with Bob (Saturday evening)". Now the task has the second highest priority.

5.5.1.2. Test story 2: Increasing the priority of a task

Test objective: Increment the current priority of a task (Extension 4a sec. 4.5.1.1).

Alice wants to increment the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the post office to send a package to Carol (Monday morning)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the post office and Carol's contact details.

Alice informs the system that she wants to increment the priority from the task. The system informs Alice that the priority has been incremented successfully for the task "Go to the post office to send a package to Carol (Monday morning)" and now the task has one level above standard priority, because it did not have any priority assigned before.

5.5.1.3. Test story 3: Decreasing the priority of a task

Test objective: Decrement the current priority of a task (Extension 4b sec. 4.5.1.1).

Alice wants to decrement the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to decrement the priority from the task. The system informs Alice that the priority has been decremented successfully for the task "Go to the cinema with Bob (Saturday evening)". Now the task has the second highest priority.

5.5.1.4. Test story 4: Failed update of a priority (priority out of bounds)

Test objective: Attempt to increment a priority from a task which was already set as the highest priority (Extension 6a + Extension 4a sec. 4.5.1.1).

Alice wants to increment the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to increment the priority from the task. The system informs Alice that the priority can not be incremented because the task "Go to the cinema with Bob (Saturday evening)" already had the highest possible priority.

5.5.1.5. Additional notes

These test stories represent each of the different possible scenarios that belong to the use case in sec. 4.5.1.1 . The three first stories completely cover the main success scenario, and both extensions 4a and 4b. The extension 6a, even if it is already covered with one test story, it might prove not be enough to guarantee its validity, because it is so open-ended. With this test story, we validate that the use case can detect anomalous states, but not that it can detect them all. The main reason for this, is that it is not reasonable to take into account every possible error-producing sequence of actions at this stage. Moreover, some of this invalid states are not known in advance.

However, this does not mean that any other possible error besides this will not be detected. To start with, the way the use case and the test stories are built, they hint a more or less restrictive interface. This interface, for example, already restricts any user to be able to modify only the priority of his/her own tasks.

In order to keep the test stories short and to the point, we assumed the instantiation of some classes prior to the story beginning. An example of this would be the tasks that appear to Alice when she logs into the system. We assumed that these tasks had been created previously, because it is legitimate. The use case for task creation sec. A.1.1 is a dependency of the use case for updating the priority of a task sec. 5.5.1, as can be seen in Fig. 5.1. Therefore, we could simply have added the necessary steps to create some use cases just after the step where Alice logs in. Nevertheless, we thought this was just unnecessary.

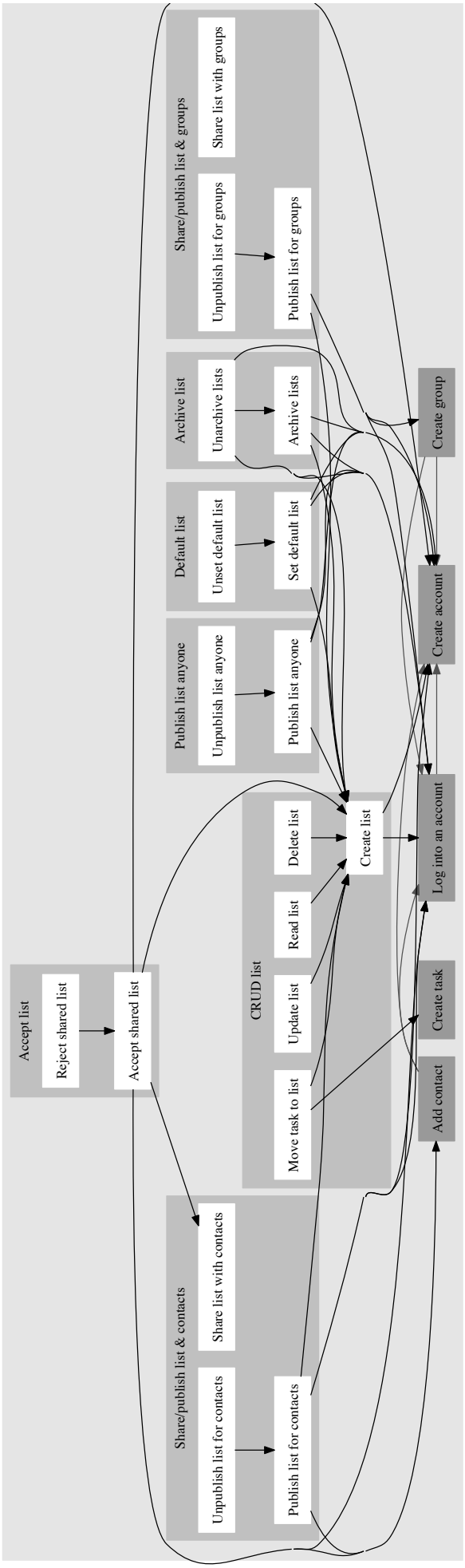

Figure 5.8.: List dependency tree

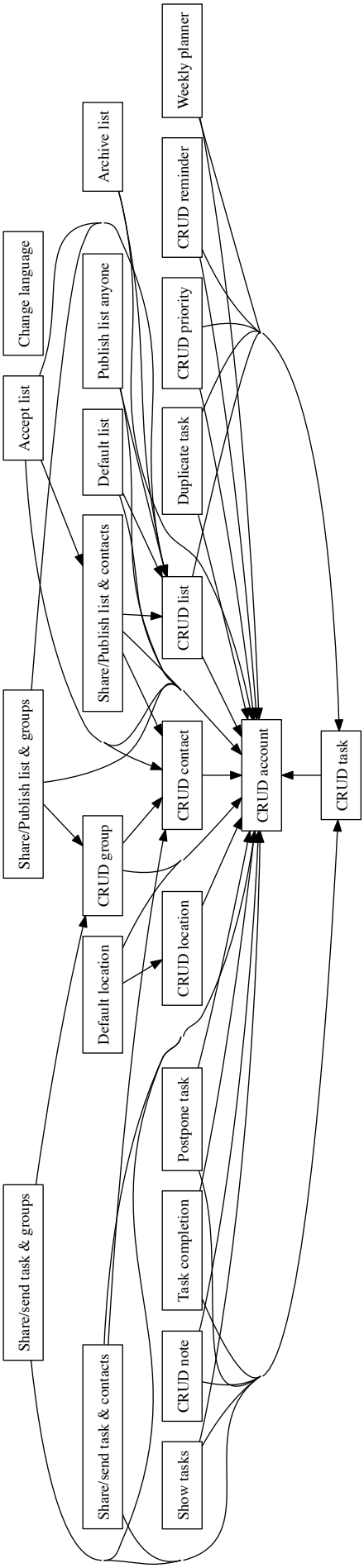

Figure 5.9.: Compact version of the system's dependency tree

6. Testing strategy

6.1. Overview

The aim of this chapter is to explain what a testing strategy is, why we are using one, which approach we took and how the strategy ended up being. In section 6.2 we define what a testing strategy is and how we are going to apply it in the context of the Test-Driven Conceptual Modelling (TDCM). In section 6.3 we discuss why a strategy is needed. In section 6.4 we introduce the model we used to determine the value of each use case. Finally, in section 6.5, section 6.6 and all subsequent sections, we comment on the followed methodology and show how the strategy was applied, by dividing all the use cases in different groups.

6.2. Method

In order to apply the TDCM it is convenient to define first a testing strategy. A testing strategy consists on:

1. Determining which is the set of stories that will be tested.
2. Decide in which order the stories will be processed.

All use cases will have several different stories, written in a way as to represent all possible situations. The prioritisation will be decided according to two criteria:

1. Use case value provided to the stakeholders, according to the Kano Model in section 6.4.
2. Dependencies that each use case requires in order to be executed as seen in section 5.4.

Considering these criteria, the algorithm we use to select each time the following use case to test could be roughly described as:

1. Select the use case which provides the most value that has not yet been tested.
2. Select all the use cases that the previous one depends on (i.e. are required for its execution).
3. Order the selected use cases according to their dependencies.
4. Select the first use case from the ordered subset and process all its test stories.

Along this section, we are going to further discuss and develop the reasoning behind these main ideas.

6.2.1. Coverage

Each grouping of prioritised use cases represents a subset of the use cases the system has. Each subset includes the contents of the previous one (and hence, by recursivity, all the previous ones). In this manner, because each subset contains all the previous subsets' use cases and some additional ones, we can guarantee that eventually the least prioritised subset of use cases will hold all the system's use cases. This way we will accomplish, in an orderly manner, the objective of testing the system's complete conceptual schema.

6.2.2. Goals

The subsets have been designed to fulfill one main goal. This goal is, for each subset, to provide enough and only enough use cases to achieve coherent functionalities that complement each other and at the same time provide reasonable (and ever increasing) value, capabilities and appeal that the average user would expect of a standalone software system.

6.3. The importance of value assessment

Because we live in a finite world, all kind of resources are scarce to a greater or lesser extent. Hence, optimization of the ratio value/cost is something constantly looked for and there is an extensive corpus about the topic. This is of special interest, considering the fact that for most of the aspects, the increase on value - and more importantly the perceived value - is not proportional to its required amount of resources and effort.

To present-day software projects and their foreseeable future, a tendency can be seen where more and more of the budget is actually spent on person-hours, rather than equipment or any other resources. An additional constraint in software production is calendar time, which can be exemplified by the rapidly evolving systems, user needs and constant obsolescence, making deadlines extremely important in a project success.

All this brings us to the conclusion that it would be wise to invest some time in optimising, prioritising and selecting in which order we are going to test and develop the system's use cases once they are known. Not only this is useful for our relatively narrow conceptual modelling stage, but it could (and should) also be applied to further stages (e.g. implementation and planning of future upgrades/additional

features) without any significant modification, proving the effort even more cost-effective.

Assessing the generated value of some functionalities has always been a tough challenge [12]. Due to the nature of the TDCM, we are going to pinpoint the small set that conforms this issue and provide a sensible solution (or a way to deal with it effectively) to it, as it is present in almost all our scope.

The persistence of this problem (inadequate value assertion) is especially true in the context of software design, where even the needs to be solved are not clearly envisioned by their future users and hence translated into a software solution poorly conceived, usually not meeting their unreasonable expectations. The Second-system Effect [3] and what is known as Feature Creep or just Featuritis are clear examples of this.

6.4. Functionality prioritisation: the Kano Model

The *Kano Model*, named after its inventor, Dr. Noriaki Kano, is an analysis model first presented in 1984 [11] where he unveils the fact that performance on certain requirements produces higher levels of satisfaction than others. Its goal is to identify and classify the different user requirements into a defined set of categories whose degree of completion / perceived value follow some recognised patterns.

With this classification in mind, it is possible to effectively order and prioritise the user requirements in the most convenient way, also taking into account how much they should be emphasised/focused. It is also possible to maximise user satisfaction given a limited amount of resources available (i.e. not being able to fully implement them all). And most importantly in our context, its principles are directly applicable with little to no modification as a roadmap to foster sensible use case selection.

6.4.0.1. Example scenario

In order to better understand the different requirements, we will be using the same example for all the properties. Let's imagine a daily newspaper that publishes each day a weekly a weather forecast. These forecasts are computed each day using a software system with a dedicated computer that takes the current weather station readings. We also know that the printed edition of the newspaper is in black and white.

All those requirements were categorised into 4 distinct classes of qualities[1] :

[1]Being the original categories written in Japanese, there is no universally accepted translation into English for them. Other authors referred them (or their equivalent counterparts) in their papers with alternative nomenclatures such as "Dissatisfier, Satisfier, Critical, Neutral" [4] , "Minimum requirement, Value enhancing, Hybrid and Unimportant as determinant" [1] and

6.4.1. Must-be (a.k.a. Basic)

These attributes are expected and more often than not considered to be common sense enough not to be necessary to make them explicit. They are also the ones with a greater potential to cause insatisfaction if they are not sufficiently fulfilled, as probably would render the whole system unusable or impractical. Nevertheless, outperforming the needed degree of achievement will not produce any further satisfaction to the end user.

Context 1

With our example scenario[2] in mind, one such requirement would be computational speed. If the software takes more than one day to make the prediction on the given hardware it is definitely not fit, and not desirable at all. On the other hand, if the software is extremely fast and takes just milliseconds to compute the weather forecast is not something that adds value for the newspaper manager in comparison to another software that takes 10 seconds to do the computation, even if it is orders of magnitude faster.

Context 2

Dealing again with the example scenario[3], another requirement would be the period of the actual forecast. If the software can only compute 3 days ahead it is definitely not usable in the context of a weekly forecast. On the other hand, a 30-day prediction would not be any more valuable in that context than a 10-day one.

6.4.2. Attractive (a.k.a. Excitement)

These attributes are usually neither expected nor required by the user. Because of this, if they are not implemented nobody is going to miss them, and hence no dissatisfaction will be produced. On the other hand, if they are implemented successfully and provide some innovation/interesting feature they will produce a great amount of satisfaction for the effort invested into it, as it is something that will surpass expectations.

"Flat, Value-added, Key, Low" [23] just to name a few. In this document we will use none of these.

[2] See sec. 6.4.0.1

[3] See sec. 6.4.0.1

Context 1

With our example scenario[4] in mind, one such requirement would be the temperature sensation. It is known that wind, humidity and other factors affect the human temperature feeling. If the software predicting the weather forecast could also predict the temperature sensation it will probably be greatly appreciated, as a distinctive factor to be published in the newspaper. If the software does not predict it, it will probably go unnoticed, as other published forecasts currently in the market do not offer such a feature.

6.4.3. One-dimensional (a.k.a. Performance)

The name derives from the fact that ideally the curve value/effort could be represented in a single dimension along the whole spectrum of efforts, that is, that instead of a curve it would be in fact a straight line. Furthermore, it would indicate a high correlation between effort increase and increase and perceived value which would be directly proportional. In reality, this is a simplification, and further consideration of other factors like de Law of diminishing (marginal) returns[5] should be taken into account.

This could be summarised as satisfaction if the feature is represented and dissatisfaction if it is not (or not enough). Usually these kind of attributes are the ones more actively looked for, and the ones used to compare a product (software) to each other, as they represent the "core" of the perceived value.

Context 1

With our example scenario[5] in mind, one such requirement would be the prediction accuracy. If the software's prediction hit-rate is an unimpressive 60%, probably it is beyond any utility and it will cause dissatifaction because it is not fit to be published in the paper. On the other hand, if the accuracy is 95% probably the newspaper manager will be quite content, as it is a very good result. Furthermore, he will still be much more satisfied with a software that has a hit-rate of 98%, because this means that the miss rate is reduced to less than a half.

6.4.4. Secondary

These attributes have no apparent value (neither negative nor positive) to the final user, and hence, do not cause satisfaction or dissatisfaction whatsoever. There is nothing to be gained by putting effort into these attributes, so they should be avoided whenever possible.

[4] See sec. 6.4.0.1
[5] See sec. 6.4.0.1

Context 1

With our example scenario[6] in mind, one such attribute would be the ability of the software to produce colour symbols for the sun/clouds and other forecasting graphics as well as black and white. The newspaper manager will not be satisfied or dissatisfied with the fact that it can produce colour graphics, because it is something he/she will not be using in the foreseeable future, and hence it has no value for him/her.

6.4.5. Reverse[7]

These attributes cause dissatisfaction to the user, and hence should be avoided whenever there is not an evident and big enough advantage to at least counterbalance it.

Context 1

With our example scenario[8] in mind, one such attribute would be watermarks. The fact that the software adds publicity of itself - in the form of watermarks - in the printout graphic of the weather forecast, dissatisfies the newspaper manager. This happens because one important source of income for the newspaper is publicity printed in it. It also dissatisfies him because everyone will know where to obtain the forecast and substitute the need to check the newspaper. Nevertheless, if this is counterbalanced with the version being free of charge (which would satisfy him/her) it might be enough for the newspaper manager to accept using the software to publish the forecast.

6.4.6. Graphical representation

After having defined each possible requirement category - according to the Kano Model - we are going to visualise them represented in a cartesian drawing. It is important to understand that each requirement should be analysed independently from any others. Hence, in a same working system is common to have different requirements that follow different curves.

The axis of abscissas represents the degree of fulfillment of the requirement or more informally how well the requisite is being covered. The higher value of X, the more thoroughly the feature has been developed. On the other hand, the vertical axis represents the costumer satisfaction, which is highly correlated with the perceived

[6] See sec. 6.4.0.1
[7] This attribute is not part of the original set, and sometimes it is not taken into consideration
[8] See sec. 6.4.0.1

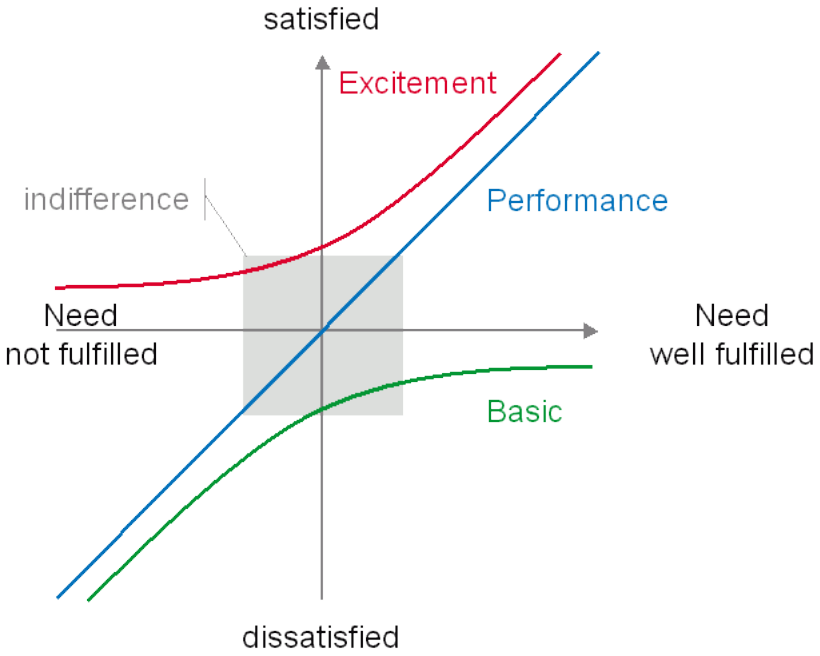

Figure 6.1.: The Kano Model illustrated

The curves regarding Secondary and Reverse requirements have not been included. We have done so because they are not looked after and should be avoided whenever possible (in order to attain them effort has to be wasted, and no value is produced). Furthermore, it makes the drawing less cluttered.

utility the system has for them. Again, a high value on the Y scale implies happiness on the part of the user, at least regarding that aspect.

Basic

As we can see in the drawing, these requirements are along the whole curve (i.e. in all possible scenarios) the ones that provide less satisfaction per amount of resources put. Furthermore, one can see that even if infinite fulfillment could be achieved on them, it would not be enough to make any costumer satisfied, even moderately. This means that these requirements alone by themselves are of no value for the user. Therefore, they should be developed only enough to let all the other requirements be build on the top of them and work smoothly. The extra amount of resources to make the basic requirements really outperform the user needs is of low value. There is one point where the perceived gained value is infinitesimal in comparison to the fulfillment, because the perceived enhancements are each time less significant for the user. This information reinforces what we already said in sec. 6.4.1.

Performance

As we already discussed earlier in sec. 6.4.3, the degree of fulfillment affects the satisfaction in the same way along all its spectrum. We can confirm this intuition by noticing that the function follows an ascending straight line. It is worthy of remark that this line passes through the center of the graph, where both axis intersect. Furthermore, the angle is approximately 45 degrees or what is the same, the line equation follows this formula $x = y + 0$, splitting the graphic in two equal-sized halves. The exaggerated precision of this information must be taken with a grain of salt thought, but it serves as an illustration of the trend that those requirements follow. This trend shows us that further performance will bring noticeable additional amounts of user happiness, and hence it should be maximised whenever there are the necessary conditions.

Excitement

This kind of requirements are the ones with more beneficial potential. Their main advantage, as we saw in sec. 6.4.2, is that they will not dissatisfy the user in any case, so all attempts to fulfill them are good opportunities, even if only partially functional. This fact is further stressed as we can observe that for any degree of effort or fulfillment put, the user will experience more happiness with this kind of requirements than with any other. Nevertheless, a system to properly work needs the basic requirements and maybe even some of the performance, because the lack of them could provide more dissatisfaction than what we would achieve with excitement requirements to outweigh it.

Indifference

The grey square surrounding the area where both axis cross represents the results a user would expect or just do not care about. This is why there are no extremes. The user expects the system to have more or less accomplished functionalities of the types and to the degree found within the square. All the features found beyond (i.e. outside of) it will most probably be noticed by the user for their notability. Failure to achieve this gray area is bound to cause some serious disappointment and have negative implications. On the other hand, exceeding the boundaries could even end up with additional rewards for the developing party.

6.4.7. Limitations

Classification of requirements

The main limitation that would prevent the utilisation of this model is the apparent difficulty in assessing which requirement belongs to each category. Even if the de-

veloping team has proven expertise and long experience developing similar systems, it is up to the final users to decide if it suits their needs. After all, they are the ones who know more accurately their business strategy. And at the same time they are the ones that know less the capabilities of the technology. This is why prototyping is encouraged along its usage, because the stakeholders can better assess the progress if they can try it and see it work.

Customer Requirements → / Functional Question ↓		Dysfunctional Question: Like	Expect	Neutral	Live with	Dislike
Functional Question	Like	Q	E	E	E	L
	Expect	R	I	I	I	M
	Neutral	R	I	I	I	M
	Live with	R	I	I	I	M
	Dislike	R	R	R	R	Q

M	Must-have	R	Reverse
L	Linear	Q	Questionable
E	Exciter	I	Indifferent

Table 6.1.: The Kano Model questionnaire matrix

Besides this cooperation between parties, this approach offers a standardised questionnaire that helps stakeholders decide which category each requirement fits without having to consciously think or have a deep knowledge about it. Each survey asks the user about every feature. Specifically, how would he/she feel if the functionality was not present, or on the other hand, how would he/she feel if it was present. Once the answers are gathered, developers only need to categorise them according to the matrix in sec. 6.4.7 .

Because the main goal of this project is to investigate the benefits of using Test-Driven Conceptual Modelling (TDCM) and specifically the power of Conceptual Schema Testing Language (CSTL), the survey was skipped altogether. Nevertheless, the construction and evaluation has not been blindfold, because prior to this section, the system has been used extensively in order to become familiar with it.

Modification of requirements

The Kano Model acknowledges the fact that software systems are not static entities, because the stakeholders want them to adapt to their ever changing needs and challenges. The methodology says that once the novelty is over, the features that once were considered excitement start to become flatter in the graphic, more like a

straight line, and their behaviour is slowly modified until they become performance requirements. Similarly, performance requirements eventually become basic, because the user is so dependant and reliant on them that their view upon the matter also changes.

Because this project does not span a very long period of time, this fact has not been taken into consideration.

6.5. Methodological approach

In order to fulfill the goal, we will make use of the dependency tree as shown in sec. 5.4. Each of the subsets will be self-contained, meaning that there will not be any use case whose dependencies are not entirely satisfied. Furthermore, this way we can ensure that the additionally provided value on each iteration is real, as it would not only maximise the value attainable on each one (we are ordering them accordingly), but also guarantee that it is usable and understandable, as it has all required use cases to be correctly executed and fully exploited. This philosophy is not dissimilar to that of prototyping [12].

6.5.1. Graphical representation

The graphical representation we use to visualise the selected use cases in each subset, is not dissimilar to that in sec. 5.4.1.3. Nevertheless, we are revising it again, as there are some minor changes.

The use cases dependencies are presented here in the form of directed graphs. These are the parts that constitute them :

1. Nodes: each node represents a single use case. The name on a node is usually the same one as its use case or a close abbreviation. Depending on their colour, each node has additional meaning:
   a) White: these nodes represent the use cases being added this iteration to the subset being analysed. Each subset is announced in the figure name.
   b) Dark gray: these nodes represent the use cases that other white nodes depend upon. The main difference between white nodes that also have dependant use cases and dark gray ones, is that the later would not belong to the subset additions of this iteration, and are just added because they are needed despite being from different groupings.
2. Edges: each edge going from one node to another node represents that the first use case has at least one dependency with the second one. We can also notice that there are no mutually dependant use cases, so each edge has one and only one arrow end. Similarly to what happened with nodes, each colour brings additional meaning.

   a) Light gray: these edges represent some of the dependencies that nodes added in previous iterations have with other nodes.

   b) Dark gray: these edges represent the dependencies that dark gray nodes have with other nodes.

   c) Black: these edges represent the dependencies that white nodes have with other nodes.

3. Groups of nodes: to make it easier to discern, nodes have been grouped into "boxes" which contain a small subset of highly cohesive nodes. This further layer of abstraction should help visualisation and accentuate particular existent relationships between nodes. They only have one colour representation :

   a) Light gray: they represent the use cases added to the subset this iteration and the others needed for them to work.

In order to improve the graph's readability, some of the nodes that depart or arrive at the same node have been simplified and unified. In some cases, this makes only one edge colour visible, but it should not seriously affect the overall understanding.

The different subsets are grouped and described in the following classes:

6.6. Subset 1: Basic use cases (i)

Rational

The purpose for this subset of use cases is to implement a very limited set containing the most *basic*[9] requirements and only those. This decision was made because this is the first subset, and consequently, it should focus in making the system usable and testable. With the chosen use cases, the system can already achieve the minimum functionalities to cover the most important needs of the user. Even if those necessities would be barely covered and some use cases would certainly be missed, there would be just enough of them to fulfill the role of a simplistic task manager. The rule of thumb is that all additional petitions a user would request from this point on should be in the form of "Wouldn't it be nice if" or "I would also like to" meaning that they can already make use of the system as it is.

Use cases

The use cases that comprise this set are :

Create task: The inclusion of this use case is self-explanatory. Without the ability to create tasks there can not be a task manager system.

[9] See sec. 6.4.1

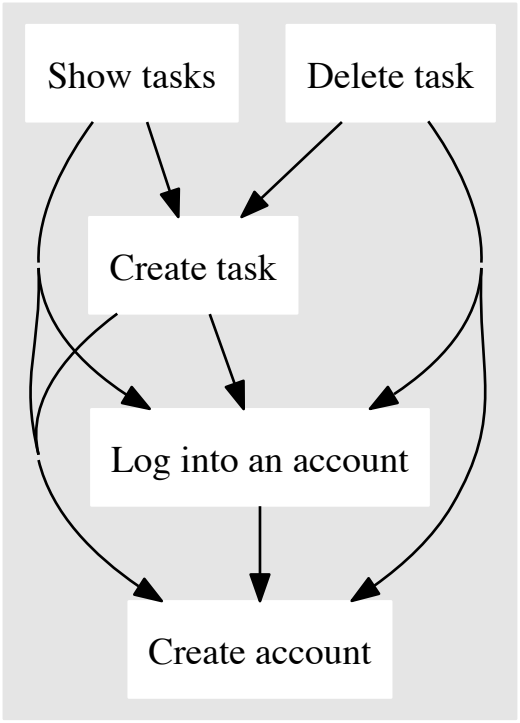

Figure 6.2.: Dependency tree of Basic use cases (i)

Delete task: The ability to delete previously created tasks is essential in a task manager. It lets the user know which tasks are pending giving him/her the option to delete those that are not. It also lets him/her delete any task with incorrect or outdated information he/she could have entered and create a new task again with the amendments.

Show tasks: A task manager must have means to let the user retrieve which tasks are into it. Nevertheless, not all the potential for this use case is needed at this point, such as complex filtering patterns and/or orderings, so it should be revisited later on to refine those aspects as soon as they are found desirable and plausible.

Create account: If the system was intended to be single user this would not be strictly necessary. Even if privacy is paramount to us, there would be other means beyond the system to prevent undesired access. Because the system is conceived as multiuser from the very beginning this is not the case, so an account system is mandatory.

Log into an account: There is no use for any account if it can not be accessed, thus, this use case must be implemented as well. A log out it is not really needed at this point because the user could be asked to log in every time he/she performs an action or, alternatively, the system could terminate the user session after a lapse of inactivity time.

6.7. Subset 2: Basic use cases (ii)

Rational

The purpose for this subset of use cases, is to implement a limited set containing all the *basic*[10] requirements that were not already included in sec. 6.6. Even if strictly speaking we already included all of them before, there would probably still be some users that would find the prior system too simplistic. Those users would probably complain that they still are still lacking some additional features in order to be able to use the system properly. All these use cases address those possibly perceived missing functionalities that prevented the usage of the system to some users the way they might expect.

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

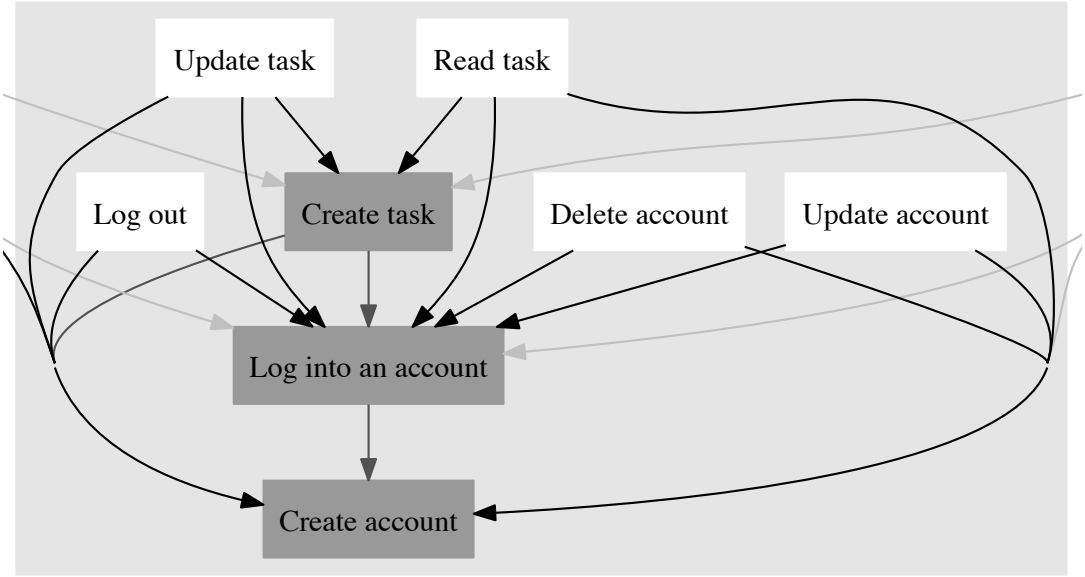

Figure 6.3.: Dependency tree of Basic use cases (ii)

Read task: The ability to read tasks is necessary if further information beyond the task name wants to be retrieved. The capabilities of this use case are beyond the ones from Show tasks: on the facing page. Henceforth, the concept of task must be expanded to fit all this additional knowledge.

[10] See sec. 6.4.1

Update task: Some of the information of a task may change in a period of time prior to its completion, or may even be unknown when it is created. This use case makes such modifications much more convenient than the deletion and later creation of a task with the updated information. Furthermore, it lets the user modify more of its properties, not only the task name.

Update account: The ability to modify account settings and information can prove especially useful when someone thinks his/her account could have been compromised and wants to change the password in order to keep his/her privacy. If this situation happens, it is much more desirable to update the account by changing the password than the alternative of using only the subset1 use cases, which would force a user to create a new account and reintroduce all his/her tasks. There are many more situations where this use case could provide great value to the user.

Delete account: The ability of account deletion is quite necessary from the point of view of the user, as well as the point of view of the service provider. The first one might not want to have to manually delete all the tasks and update all his personal information to null values when he/she decides that he/she does not want to use the system anymore (maybe because he/she thinks his/her information is not safe there). The second one might want to delete inactive accounts to free resources. Even if this problem could be solved by simply adding more resources, being able to use the ones already present more efficiently sometimes it is not a choice that can be rejected, because the additional resources are not available. So in this sense the use case could be understood as a *basic* one.

Log out of an account: The ability to log out permits the user be confident about the safety of his/her privacy, because he/she can control when they make the account inaccessible to those not knowing the password instead of having to rely on the program detection of lack of activity to assume the user is not anymore making use of the system. Additional means of protecting personal possibly confidential data sometimes are relevant enough as deeming a system unusable without them. Therefore, this use case can be considered *basic* in most situations.

6.8. Subset 3: Performance use cases (i)

Rational

The purpose for this subset of use cases, is to build a set containing the most fundamental performance [11] requirements. Ideally we would like to include all of

[11] See sec. 6.4.3

them at the same stage, because they are in the same spectrum or category according to our satisfaction model [12], but we decided not to do it this way. One of the reasons is because the use cases in this category are so plentiful that packaging them all together would difficult comprehension. Another reason is that it would not really be sticking to the philosophy of prioritisation to include such a massive (especially in proportion to the whole amount of them) package of use cases at the same time. Finally and most important, even though they are all performance use cases, they do not share the same degree of "performance". Some of them are closer to the basic[13] category while some others are closer to the excitement[14] one. Furthermore, the gap between them is such as to naturally make such groupings as we are going to see in the next subsection. In such situation, as we already justified in sec. 6.6, the best thing to do is to start with the most basic ones that achieve coherent functionality.

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

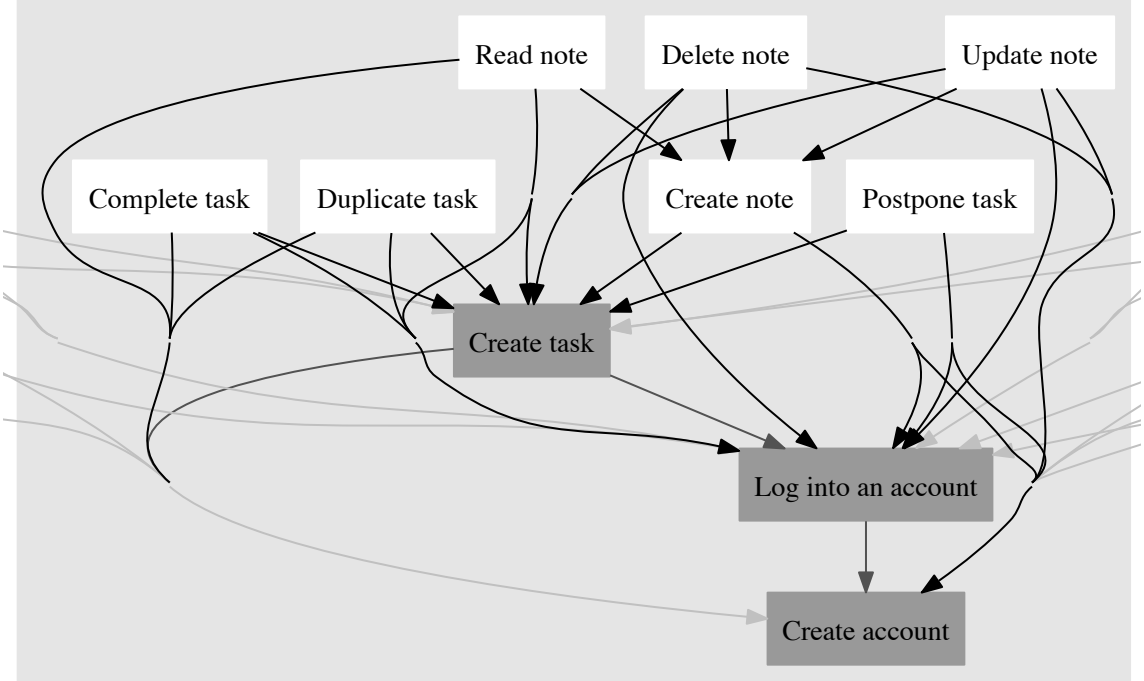

Figure 6.4.: Dependency tree of Performance use cases (i)

[12] See sec. 6.4
[13] See sec. 6.4.1
[14] See sec. 6.4.2

Create and add a note to a task: The ability to add notes to tasks is certainly a welcome feature for most of its users. It is an easy and flexible way to add additional information to a task. Default fields always limit users to the amount and kind of information they can input, as well as the format. This simple mechanism empowers them to introduce anything they may desire that was impossible to envision or predict by the designer. It is certainly not necessary to work with the task manager, but greatly improves its usability and appeal to the average user.

Read a note added to a task: There is no point in creating and adding notes to tasks if their content is not accessible.

Update a note added to a task: Updating the information is usually preferable to deletion and creation of the updated version.

Delete a note added to a task: Not strictly necessary, because many notes can be added and updated, it is an efficient way to use resources and to keep control of the number of notes and their thematic content. Furthermore, the ability to create instances of any class should always be paired with the ability to delete them whenever possible.

Complete task: The ability to mark a task as completed is not strictly necessary, because finished tasks could always be deleted manually by the user. The ability to complete them gives the user a neat way to filter[15] them out, and at the same time it provides the ability to keep his/her historical, which could be checked in the future.

Postpone task: In our context, we know that task postponing is probably the most common kind of task updates. The ability to postpone a task using this simpler and more straightforward use case (in comparison with the former Update task) will probably help the user save an additional amount of time.

Duplicate task: Frequently enough, the user needs to create a task which is virtually the same or very close to another one he/she already created. This use case, with the aid of task updating should save the user time in comparison to the creation of a new task and the consequent filling of all the properties.

6.9. Subset 4: Performance use cases (ii)

Rational

The purpose for this subset of use cases, is to implement a set containing most of the performance [16] requirements. The use cases added in this subset should help the

[15] See Show tasks: on page 61
[16] See sec. 6.4.3

user make his/her life easier, by helping him/her use the system more effectively, effortlessly and comfortably.

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

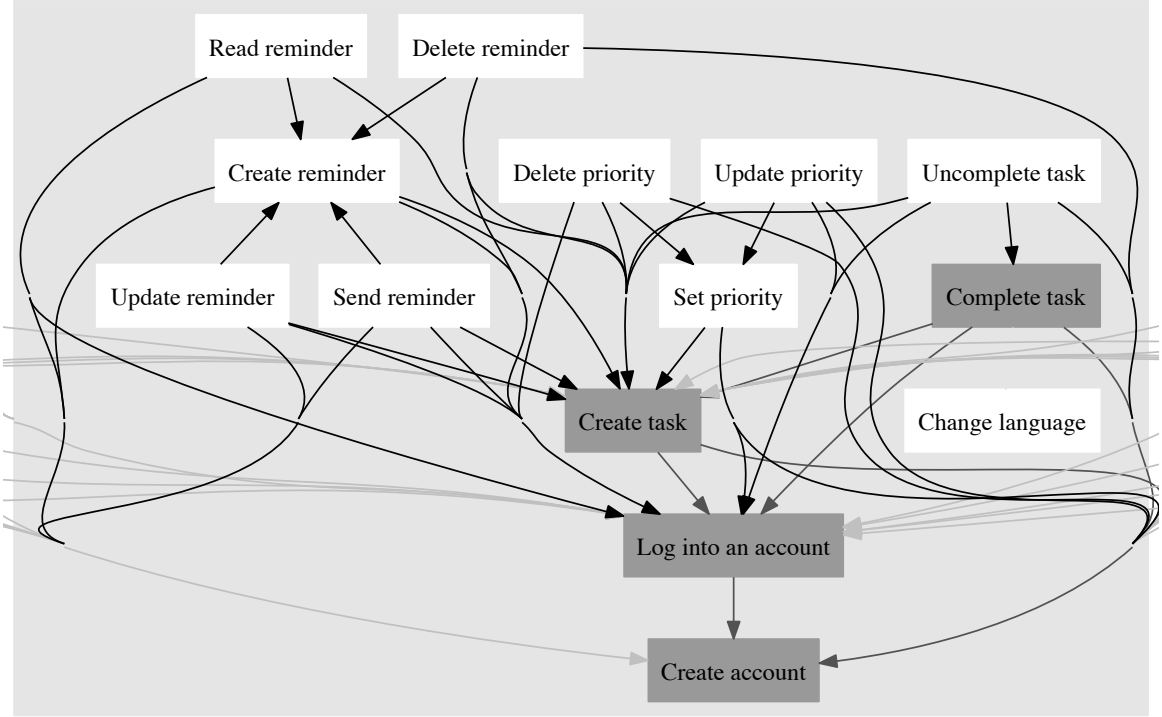

Figure 6.5.: Dependency tree of Performance use cases (ii)

Uncomplete task: The ability to complete tasks is already in the system, so implementing this makes sense. This was not included in the previous subset because the use of it should be much more sporadic. Its main purpose is to let a user uncomplete a task he/she completed before by error.

Set priority to a task: The ability to set priorities to tasks helps the user better manage his/her pending tasks. Furthermore, it also makes it easier to find[17] the most important ones at any point.

Update priority from a task: Updating task priorities is a logical mean to keep ever-changing information correct.

[17]See Show tasks: on page 61

Delete priority from a task: Even if it is not strictly necessary, because we could always update a task to a "standard" level of priority, it makes sense to have the ability to delete priorities the same way they can be created.

Create reminder schedule: The ability to create reminder schedules in some contexts could be thought as an excitement[18] feature instead of just a performance one, for example if the reminder is sent by SMS. We already discussed that the line that separates them can be fuzzy sometimes. Nevertheless, we include it here as a performance due to the fact that the key objective of a task manager is to make the user not skip a single task by having all of them present. Reminders achieve this in an additional way, not needing the user to access his/her account and check it constantly.

Read reminder schedule: The ability to read and see which reminder schedules are set is essential once they can be created. This way the user knows if he/she has to worry about setting a new one or the ones set already meet his/her needs.

Update reminder schedule: The ability to update the reminder schedule, again is a good alternative to deleting and creating it anew.

Delete reminder schedule: The ability to delete a reminder is almost as important as the ability to create one. It is true that one could simply ignore the reminders and this would suffice to some, but to most users this would be unacceptable.

Send reminder: This is the most important use case dealing with reminders. No matter what, if they can not be sent, they are completely useless.

Change language: The ability to change language might be basic to some users and performance to most of them. Even if the interface is used with as little text as possible, some text is unavoidable (especially in the form of explanations, help or tips) and most people find themselves more comfortable when they can interact in their native language.

6.10. Subset 5: Performance use cases (iii)

Rational

The purpose for this subset of use cases, is to include a set containing all the rest of the performance [19] requirements. The use cases added in this subset should contribute in helping the user get his/her tasks more organized and storing and accessing task-related information faster.

[18] See sec. 6.4.2
[19] See sec. 6.4.3

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

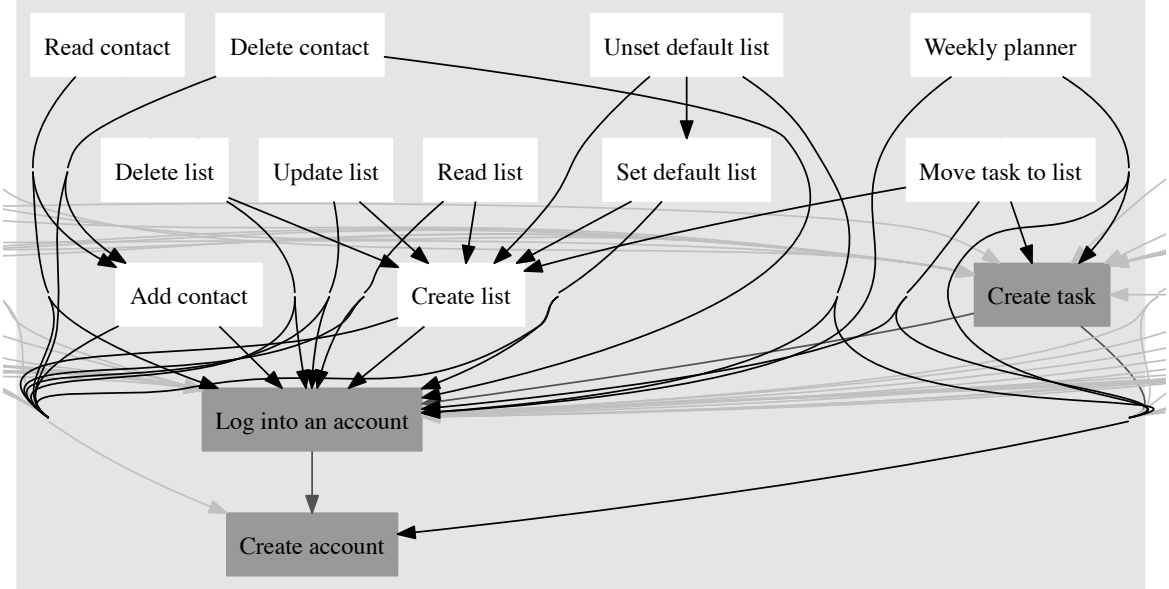

Figure 6.6.: Dependency tree of Performance use cases (iii)

Create list of tasks: The ability to create lists lets the user customise and group together tasks that may share some of their context. This extra layer of abstraction, even if it is not strictly necessary, may help the user get to know where every task is according to some patterns and focus more on the actual job of managing them rather than having to filter them or tediously search through all of them. This is especially true when the number of tasks grows into a considerable size.

Read list of tasks: The ability to read the list of tasks is essential once you have them. Without so many parameters and more readily available, is sure a better and more convenient choice than simply use the already included use case Show tasks: on page 62 in some cases.

Update list of tasks: This use case is the only reasonable way to update information from a list of tasks. Deleting and creating anew a list with the modified information is not an option when the number of tasks is in the order of hundreds or even bigger. And for most users, lists of tasks do get this big in not such a long time.

Delete list of tasks: The ability to delete lists is quite necessary from the hygienic standpoint. The user may feel the need to delete unused lists in order to

avoid the clutter this would represent and being overwhelmed by much more information than he/she is able to handle.

Move task to a list: The ability to move tasks to lists is what makes them useful. It is quite self-explanatory why this use case is included at this point and not afterwards (or before).

Set default list of tasks: The ability to set a default list of tasks, means that all newly created tasks will be automatically classified as belonging to that list. It is a way to save time when there is a predominant list of tasks. This again, to some users might be considered an excitement[20] feature instead of just performance[21] but we think otherwise.

Unset default list of tasks: The ability to unset default lists is just a logical pair to the ability to set them.

Show weekly planner: Besides already having reminders, some users may still want a more thorough mean to envision their schedules and appointments. Furthermore, a weekly planner can be printed and consulted anywhere without many of the restrictions any software has.

Add contact: The addition of the ability to add a contact is a natural way to expand the initial set of features of a task system. Most of the tasks will usually involve other people. Being able to keep an agenda with both, the people involved and the things to be done in one single place is convenient and potentially a decent timesaver.

Read contact: Once the addition of contacts is in the system it just makes sense to be able to read them.

Delete contact: For a reason analogous to the deletion of lists of tasks, the ability to delete old unneeded contacts just makes more useful and accessible those that are still in the system. Hence, it is positive to have such a use case at this stage.

6.11. Subset 6: Excitement use cases (i)

Rational

Having already included all the basic and performance use cases, the next logical step would be to include the excitement ones. As happened before, this set is split in two, and the excitement use cases more close to being performance are included here. The use cases added in this subset should give the user new possibilities to interact with the system and the administration of his/her tasks collaboratively, something that it is not essential to a task manager system.

[20] See sec. 6.4.2
[21] See sec. 6.4.3

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

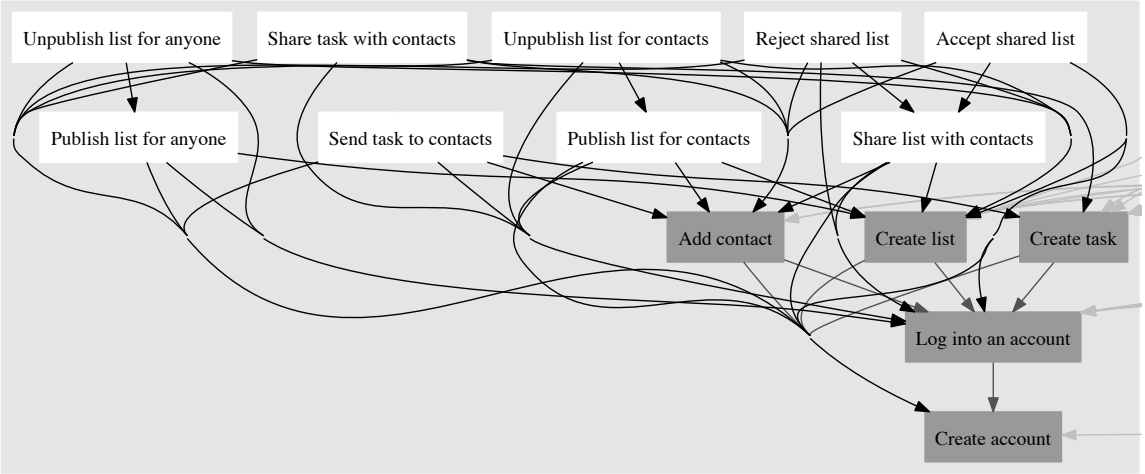

Figure 6.7.: Dependency tree of Excitement use cases (i)

Share task with contacts: The ability to share a task with contacts within the same system has lots of potential. A group of users could be doing the same task, and just have a copy of it with the previous version of the system. The problem, is that the required communication to keep its information up to date with all members using other methods would waste a lot of resources and would also suffer from delays, as everyone could not update the task information instantly and at the same moment. So, despite being neither necessary nor expected it could certainly be appreciated.

Send task to contacts: The ability to send tasks to contacts is very similar to the ability to share them. The main difference, is that the user that sends them, has his/her tasks moved to his/her outbox list, and consequently can not modify it or alter it in any way. It is useful when someone has to assign tasks and manage other people.

Share list with contacts: The same advantages that lists of tasks have over individual tasks are applied here. This could be a potential time saver in comparison to share multiple tasks individually.

Publish list for contacts: The difference between publishing and sharing, is that publishing only allows the other users to read the tasks, not to modify or update them in any way. This can come handy when some user, for example in a managerial position, wants to use tasks as a way to report progress, and he/she should be the only one making modifications to them according to plans, etc.

Publish list for anyone: Sometimes a user needs a list to be read for someone that does not use the system. This is an easy way to accomplish it, without having to send the task each time via other communication methods (e.g. e-mail) when there are modifications.

Unpublish list for contacts: The necessary pair to publishing for contacts. Imagine that some user is expelled from the group of people working with that list of tasks. Once there is the ability to publish, this turns into a necessity.

Unpublish list for anyone: The reason for this use case to be here is analogous to the one above it. It might not be desirable to have a list accessible by everyone.

Accept shared list: This use case provides an interesting twist to the Share list with contacts. We let the other users decide if they want to accept a shared list. And we also let the other party know if he/she conforms with the acceptance of the list, so misunderstandings are avoided.

Reject shared list: This is the complementary pair to the above use case. We could always leave some shared lists "on hold", but, because the purpose was to make things clear and avoid misunderstandings, it just seems logical that those lists could be rejected as well.

6.12. Subset 7: Excitement use cases (ii)

Rational

The purpose for this subset is to include all the excitement use cases that did not fit in the other one. The use cases presented here complement and expand the concepts on the ones which were introduced in Excitement use cases (i). Basically, they make collaborative administration easier, faster and more effective. They also could hardly be understood without the previous ones.

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

Share task with groups: This is the equivalent of sharing tasks with contacts but applied to groups. The same reasoning persists.

Send task to groups: As with the previous use case, this is the same we reviewed in Excitement use cases (i) but applied to groups.

Create group: This use case could be described informally as being to contacts what lists are to tasks. The difference that makes the creation of lists of tasks be a performance requirement and the group an excitement one, is the focus.

The group creation lets the user customise and join together contacts that may share some of their context. But contacts play a much lesser role in a task management system than tasks, hence it is not seen as a performance or basic requirement. Nevertheless, it can still be very useful to have an efficient way to manage them, and this is what makes it fit in the category of excitement.

Read group: This use case is a convenient way for a user to know who the members of one of his/her groups are, which of his/her tasks are shared with them, etc.

Delete group: It just makes sense being able to delete groups once you can create them. The argument for list deletion could be applied.

Add contact to group: Groups would be pointless without the addition of contacts, as they would be empty.

Remove contact from group: Removal of contacts complements the addition of them to groups. More suitable in most situations than the deletion of a whole group.

Share list with groups: This use case is analogous as sharing a task with some groups, with the advantage of being a list.

Publish list for some groups: The same explanation for publishing tasks for some groups holds.

Unpublish list for some groups: The logical and needed counterpart to publishing lists for groups.

6.13. Subset 8: Indifference use cases (i)

Rational

The purpose of this final subset of use cases is not very clear from the standpoint of the end-user. It is true that a minority of users might find these use cases interesting (and then this would make them fall into the excitement category), but to most of them it simply falls in the indifference category. Locations, if better integrated might prove useful, but in the Remember the Milk (RTM) system their appearance and utility seems anecdotic. Likewise, the ability to archive and unarchive lists seem to be something seldom useful at most. Because of this, the use cases names are just going to be cited instead of being described or delved into why they belong to the category.

Use cases

All the use cases belonging to the previous set are included in this one. These are the additional use cases that comprise this set :

- Create location
- Read location
- Update location
- Delete location
- Set default location
- Unset default location
- Archive list of tasks
- Unarchive list of tasks

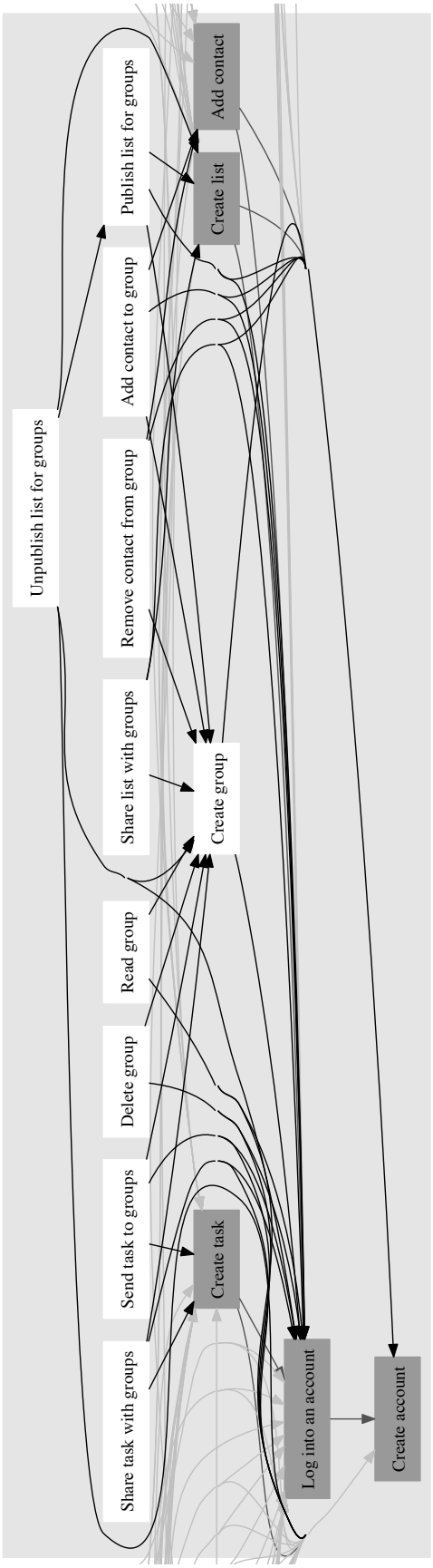

Figure 6.8.: Dependency tree of Excitement use cases (ii)

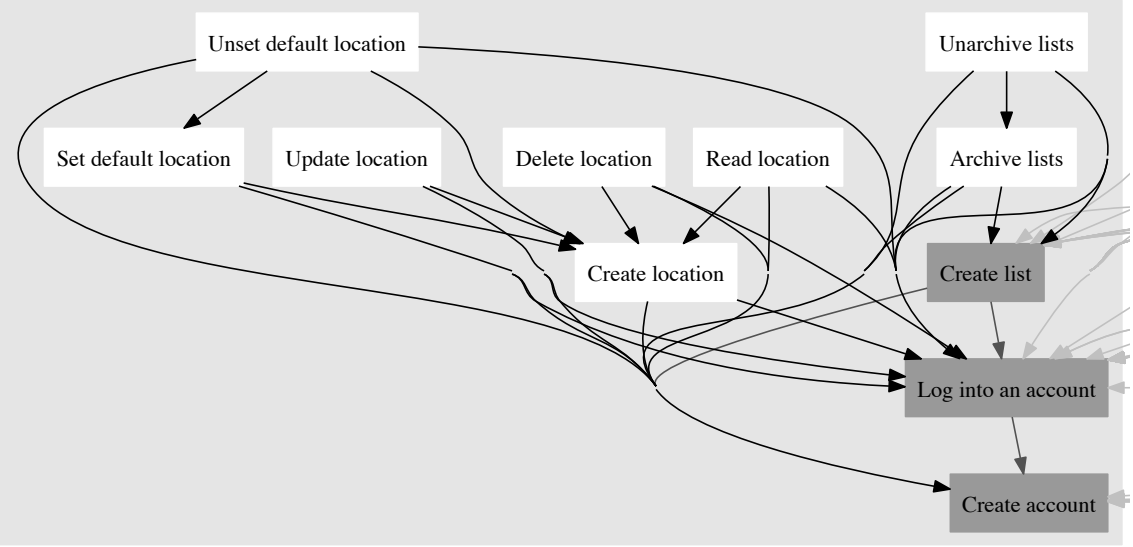

Figure 6.9.: Dependency tree of Indifference use cases (i)

7. Application of TDCM

7.1. Overview

The aim of this chapter is to give an insight on the application of Test-Driven Conceptual Modelling (TDCM), concretely to our case study: the Remember the Milk (RTM) system. In section 7.2 we talk about the tool we use to apply the TDCM cycle and its language, the Conceptual Schema Testing Language (CSTL).

In section 7.3 we talk about what the TDCM cycle consists on, its philosophy and its application. In section 7.4 we talk about the limitations of the TDCM in the context of this project. In section 7.5 we show the RTM test cases in CSTL and how a test story is translated from natural language to CSTL. Finally, in section 7.6 we apply the TDCM to our case study, showing its different iterations and the evolution of the conceptual schema.

7.2. CSTL Environment

7.2.1. Language

In this section we are going to briefly outline the CSTL language characteristics. It is a language designed to write formal tests in the conceptual level. Some of the details of its syntax are well beyond the scope of this document and a complete definition can be found in [20].

7.2.1.1. Assertions

Assertions are the means of test cases to provide verdicts[1] different than "Pass". Therefore, they are of key importance. We are going to discuss the different kinds of assertions the CSTL provides, which are Consistency, Domain events occurrence and Contents of an information Base (IB) state [20]

Consistency: The CSTL provides us with two instructions to check the consistency of an IB state. They are a good way to ensure that all the conceptual schema constraints are fulfilled. These simply are *assert consistency* and *assert inconsistency*.

[1] See subsubsection 7.2.2.3

Domain events occurrence: The CSTL provides us with two instructions to check domain events occurrence. These simply are *assert occurrence eventId* and *assert non-occurrence eventId*. They are used to give confidence about the defect-free nature of the events. Specifically, they can detect that :

1. The constraints of the event type may not allow the occurrence of valid events.
2. The postconditions may not precisely define the intended effect of events.
3. The method of the effect operation may produce an IB state that does not satisfy both the postconditions and the schema constraints.

Contents of an IB state: The CSTL provides us with four instructions to check domain events occurrence. Two of them are meant to check boolean conditions. These are *assert true booleanExpression* and *assert false booleanExpression*. The other pair is meant to check equality. These are *assert equals valueExpression1 valueExpression2* and *assert not equals valueExpression1 valueExpression2*. Their purpose is to check that one or more derivation rules derive the expected results, or that a navigational expression yields the expected results or that the effect of one or more domain events implies an expected result in the IB.

7.2.1.2. Test cases

In CSTL, there are three kinds of test cases: concrete, abstract and abstract invocation[20]. We are only going to talk about concrete ones, as the other were not used in this project.

A concrete test case is a set of statements that builds a state of the IB, defines values of its variables, and executes one or more tests of one of the five test kinds described in the previous section.

7.2.2. Processor

The CSTL Processor, is a java-based research prototype designer by Albert Tort. It works as a standalone application and supports the specification, management and execution of automated tests of executable conceptual schemas. Because of this, it will be the main tool in our application of Test-Driven Conceptual Modelling (TDCM), as it fully supports it.

The languages it uses to define and specify schemas, are an extended version of USE[8] syntax and Object Constraint Language (OCL) from Unified Modelling Language (UML). Automated conceptual test cases are written in the language we described earlier, the CSTL.

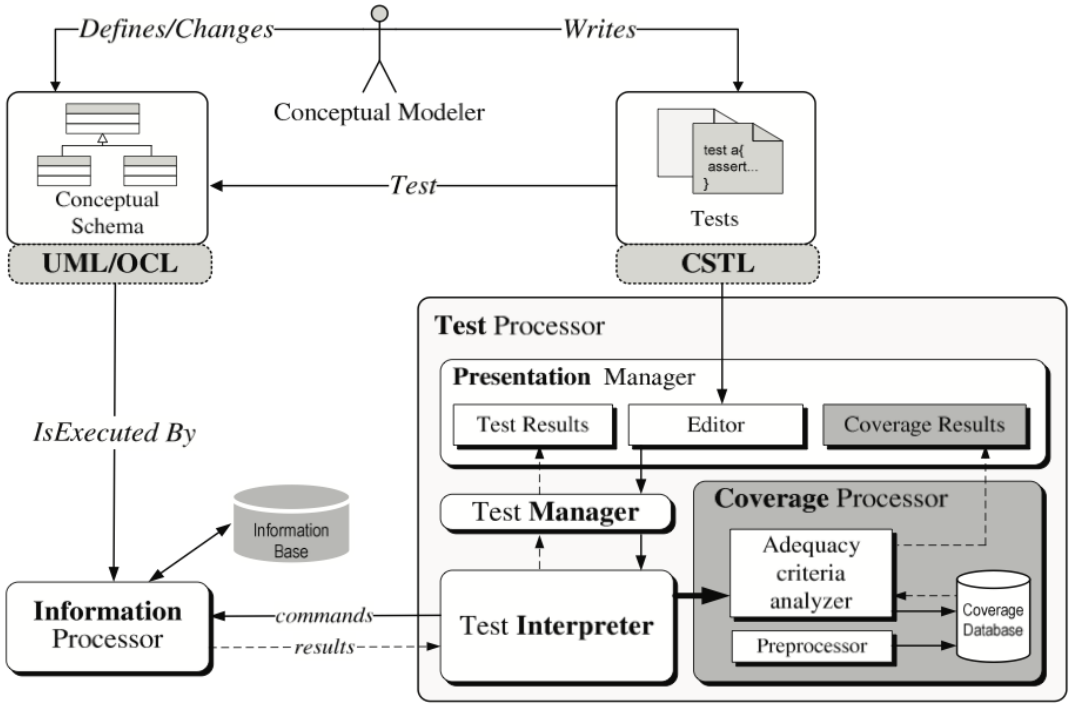

Figure 7.1.: The CSTL Processor testing environment[20]

In figure Figure 7.1 we see the main elements of the CSTL Processor. In this project we extensively used two of them, the Information processor and the Test processor, which are referred below [22].

7.2.2.1. The Information processor

The information processor is able to setup Information Base (IB) states by creating, deleting and changing entities, attributes and associations. It also evaluates OCL expressions of test assertions. It deals with language features such as derived attributes, default values, multiplicities in attributes, domain events, temporal constraints, initial integrity constraints and generalisation sets.

7.2.2.2. The Test processor

The test processor implements the execution of the test cases and consists of the presentation manager, the test manager and the test interpreter.

The test manager stores the CSTL programs in order to make it possible to execute the test set at any time. When the conceptual modeller requests the execution of the test programs, the test manager requests the test interpreter to execute them. The test manager also keeps track of the test results and maintains test statistics.

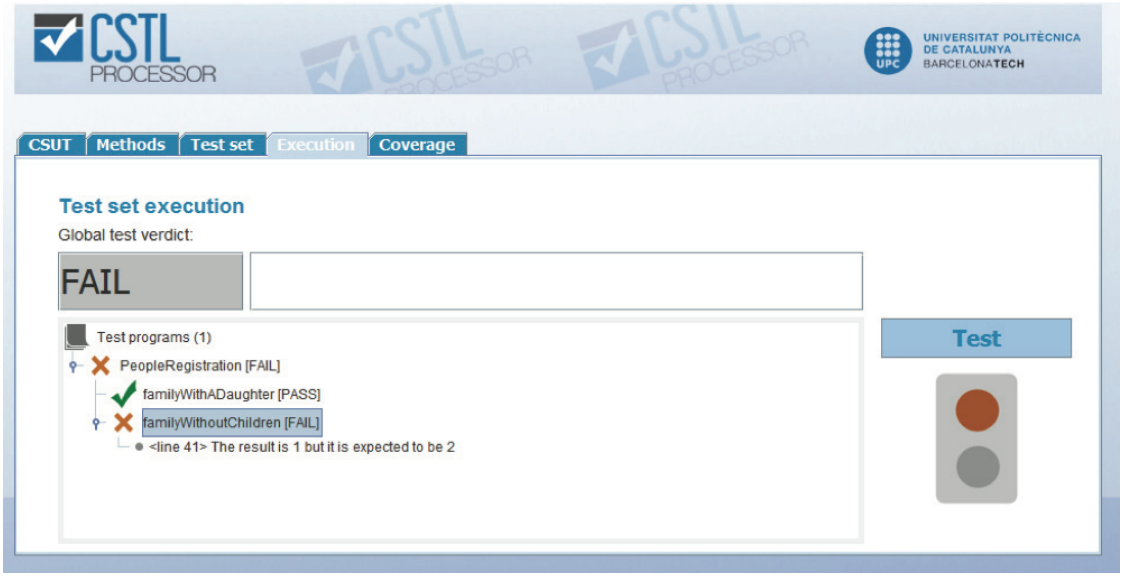

Figure 7.2.: Screenshot of the test execution tab

The test interpreter reads and executes CSTL programs. For each test case, the interpreter sets up the common fixture (if any), executes the statements of each test case and computes the verdicts. The interpreter invokes the services of the information processor to build the IB states according to each test case and check the specified assertions.

The presentation manager provides means for writing CSTL test programs and for displaying the results of their executions. Built-in editors are provided for the definition of the conceptual schema, its methods and the test programs. Moreover, after each execution of the test set, test programs' verdicts are displayed. Figure 7.2 shows a screenshot of the verdicts presentation screen. The test processor indicates the number of the lines where test cases have failed and gives an explanation of the failure in natural language.

7.2.2.3. Verdicts

Verdicts are the ultimate goal for concrete test cases. The CSTL processor can provide the tester with three different types of verdict, and their purpose is to inform him/her whether a test is passed, failed or there are errors. With this information, the tester knows if he/she has to modify some information from the UML/OCL from the conceptual schema or the methods file (if the final result is not a pass) or he/she can continue developing the conceptual schema to represent further use cases.

The verdicts are reached using roughly this algorithm.

1. Analyse the conceptual schema. If there are any errors the verdict is "Error" and the testing ends.

2. For each test case do:

    a) For each instruction belonging to the test case do :

        i. If the instruction is not a valid instance of the corresponding metaschema, the verdict for this test case is “Error”, overriding any other previous possible verdict. The testing of this case ends.

        ii. If the instruction is of the assertion type[2] then :

            A. If it makes the assertion invalid, the verdict is “Fail”.

3. If there is any “Error” verdict, the final verdict is “Error”. If there are no “Error” verdicts, but there are some “Fail” verdicts, then the final verdict is “Fail”. In any other situation, the final verdict is a “Pass”.

7.3. TDCM

The Test-Driven Conceptual Modelling (TDCM) methodology, can be understood as a cycle where a conceptual schema is defined incrementally in short iterations. An iteration starts by adding new test cases to the passing test set of the previous iteration.

The objective of each iteration is to change the schema, if necessary, so that it includes the knowledge to correctly execute the new test case. The previous set of tests in addition to the new test cases are the current test set for the iteration. A TDCM iteration can only finish when the overall verdict of the current set of tests is “Pass”.

As we have seen, each TDCM iteration, which is a particular instantiation of the TDCM cycle, is guided by different stages. Those stages consist on writing a test case, changing the schema and refactoring it and can be seen in Fig. 7.3. We describe these tasks in the following subsections.

7.3.1. Write a test case

The first task of the TDCM cycle consists of setting up a new test case whose verdict will be Pass once the conceptual schema includes the knowledge to be added in the current iteration.[21]

According to the test initial verdict[3], we can distinguish two different types of test cases, the ones that pass and the ones that not.

[2] See subsubsection 7.2.1.1
[3] See sec. 7.2.2.3

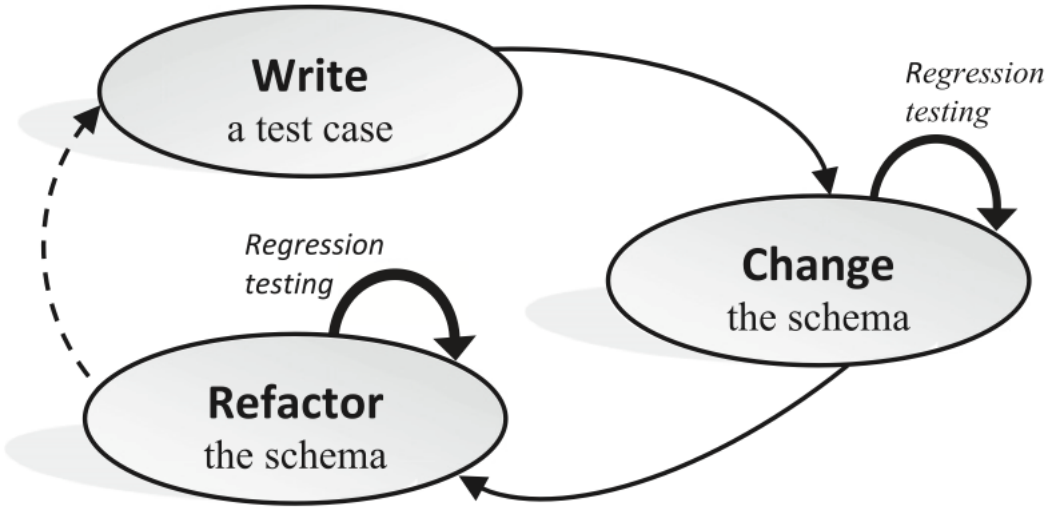

Figure 7.3.: TDCM cycle [21]

7.3.1.1. Tests whose initial verdict is "Error" or "Fail"

In the general case, the schema does not include that knowledge initially, and therefore the verdict will be Error or Fail. When this happens, progress is to be made in the TDCM cycle, as new knowledge needs to be put into the conceptual schema, making it evolve in order to end up with a "Pass" verdict.

7.3.1.2. Tests whose initial verdict is "Pass"

When the initial verdict of the new set of test cases is "Pass", then the iteration objective is already achieved and, consequently, the iteration finishes. Despite being fruitless in terms of conceptual schema evolution, such iterations can be convenient in order to increase the confidence about the already defined knowledge.

7.3.2. Change the schema

The objective of this task, is to obtain a global "Pass" verdict. The accomplishment of this task, also marks the end of it. The conceptual modeller has to change the schema, while remaining true to the knowledge it represents, to fix these errors or failures. The testing environment provides additional information about the failure or the error. At this point, changing the schema and checking if the new verdict for the test cases becomes a "Pass" is the main activity to make progress in the TDCM.

There are several different possible outcomes that would not be a "Pass", and all of them have their own meanings that we are explaining in the following lines.

7.3.2.1. The added tests do not "Pass"

This is by far the most common outcome to happen once we are in this task. Again, there are two reasons for this:

1. Error verdict: the required knowledge is not defined in the schema. Information about the error helps to find out required knowledge to be added to the schema. In this way, TDCM drives the completeness of the schema according to the user needs expressed in test cases.
2. Fail verdict: the schema needs to be changed because it does not produce the expected result (the schema is not correct according to the test assertions). The information about the failure provided by the testing environment helps to find out the changes to be done in the schema in order to fix the failure.

7.3.2.2. The existing tests do not "Pass"

This less common outcome is discovered because each time all the tests are checked. Again, there are two reasons for this:

1. Error verdict:
   a) The conceptual schema has become incomplete: A schema element that was necessary for a previous test case has been removed.
   b) The IB state of a previous test case has become inconsistent: IB states of previous test cases may become inconsistent when a new constraint is added to the schema. Inconsistent states need to be updated to ensure that test cases formalise consistent stories.
2. Fail verdict:
   a) Previous defined knowledge ceases to be correct due to the last changes: A derivation rule ceases to derive information as expected, an event occurrence ceases to produce the expected IB state or a domain event ceases to be applicable as expected. The conceptual schema needs to be changed to fix the failure.
   b) Inconsistent requirements are revealed: If it is not possible to change the schema to pass both the new test case and the previous failing test cases, an inconsistency between requirements has been revealed. The inconsistency must be resolved. Involved test cases may be changed, if needed, to reflect updated expectations.

7.3.3. Refactor the schema

> Refactoring aims to improve the quality of the conceptual schema without changing the knowledge specified in it.[21]

In this last step of the TDCM cycle, the conceptual modeller is encouraged to refactor the schema and, afterwards, request the execution of the current test set (the older ones we already had plus the ones added in this iteration), to check if it is still valid or the refactoring process has not preserved the knowledge of the schema.

7.4. Limitations

The limitations of this section may be similar to those found in sec. 5.3. After all, the project scope and the cases to be tested are the same, and the test-case stage is strongly linked to that of test-story creation. We refer them in the following subsections.

7.4.1. Scope

In this matter, like we saw in section 4.3 and in section 5.3, the scope has been a limitation to the number of use cases that TDCM could be applied. This is because RTM has 65 use cases and even more test stories and the project barely lasted 4 months.

The application of TDCM has not been exhaustive, and not all test stories have been translated into CSTL. Nevertheless, due to the solid testing strategy we defined in chapter 6, this has not affected significantly the overall results, as only the least important use cases were left out. Having pointed that out, all test cases from the Basic use cases subsets[4] (i and ii), the Performance use cases subsets[5] (i, ii and iii) and a good amount of the Excitement use cases subsets[6] (i and partially ii) were modelled according to TDCM. This only left out the valueless use cases from the Indifference subset[7] and some from the Excitement subset (ii), which are in fact very similar to the ones already tested from the Excitement subset (i).

We have tested all the stories from use cases that were in the domain events category of the said subsets. The minority that were belonging to the "query" category were skipped altogether. Those are the ones that do not modify the state of the IB, and they were not tested not only because of time constraints, but also because there are not interesting from the testing standpoint and there is not really much to test about with assertion instructions, as they do not affect or alter the state in any way.

The complete code for the final version of the conceptual schema and methods files is to be found in Appendix C and Appendix D, and they are not individually commented because it would surpass again the scope of a project like this one. The

[4] See section 6.6
[5] See section 6.8
[6] See section 6.11
[7] See section 6.13

complete code for the tests applied is omitted entirely, because its size is in excess of 10,000 lines of code, and this was deemed too many pages in comparison with the appeal it would bring to most of the readers.

7.4.2. Testing environment

These tests are to be executed using the CSTL processor, not in the real system (the RTM). As such, there are some differences, which are significant. The nature of the CSTL processor makes some of the tests obsolete. Furthermore, some of the translations can not be exact, precise and literal, but only very close approximations. So there are expected to be some minor differences that do not affect the rigour or quality of the testings in any way.

The CSTL processor's architecture, can prevent the execution of some of the cases that in real life might cause the system to fail, or lead it to an inconsistent IB state. This is one of its main strengths, because it would throw an "Error" verdict from the very beginning. Nevertheless, this kind of verdict is not interesting in the documentation of test stories, as it implies that some modifications are needed and it is not correct. The final goal is for all test cases to have a "Pass" verdict.

On the other hand, there are situations that might cause malfunction or unexpected behaviour that in real life could not happen, but in the CSTL environment are possible. An example would be for a user to modify a task not belonging to him. In real life, because of the interface, a user is not expected to select any task without having the right permissions or authorisation, and in fact, they should not even be aware of their existence. In the CSTL Processor, however, there is not a Graphical User Interface (GUI) that limits this, and theoretically, more unexpected situations than those conceived in the test stories could happen. They are not of a high value, because in real life these situations could not happen. This is the reason why most of them will go untested, even if some are indeed tested to demonstrate the robustness and correctness that the conceptual schema achieves with the TDCM. It is also a way to show that there are no missing invariants or preconditions that could be exploited by malevolent parties.

7.5. From test stories to formal test cases

In this section we are going to show how the use cases we discussed in sec. 4.5 and sec. 5.5 are finally made into formal test cases. But first, we need to know what a test case is :

> Test cases consist of sequences of expected Information Base (IB) states (which may change through the occurrence of events) and assertions about them.[22]

In our project, we are using the CSTL language to encode tests. This gives some more properties that define the structure of a test case:

> It is assumed that the execution of each test case of a CSTL program starts with an empty IB state. With this assumption, the test cases of a program are independent each other, and therefore the order of their execution is irrelevant.[20]

7.5.1. Update priority from a task

These test cases are entirely based on test-stories found in sec. 5.5.1

7.5.1.1. Test case 1: Modification of a priority from a task

Test objective: Update a priority from a task (Main success scenario sec. 4.5.1.1)

Before actually reaching the test objective, we first need to build an IB with enough elements to let us test it. First of all we create a few accounts as seen in Algorithm 7.1:

Algorithm 7.1 IB Account creation

```
test ModificationOfAPriority{
    na := new CreateAccount(username := 'Alice',
                            password := '1234',
                            creationDate := 20111106);

    assert occurrence na;
    assert equals na.username 'Alice';
    assert equals na.password '1234';
    assert true Account.allInstances()->size()=1;

    na2 := new CreateAccount(username := 'Bob',
                             password := '4321',
                             creationDate := 20111106);

    assert occurrence na2;
    assert true Account.allInstances()->size()=2;
```

Next we log Alice in, so she can later perform the update as seen in Algorithm 7.2:

Algorithm 7.2 IB Log in

```
alice := [Account.allInstances()->any(a |
                                    a.username = 'Alice')];
assert false alice.isLoggedIn;
nlog := new LogIntoAccount(username := 'Alice',
                           password := '1234',
                           isLoggedIn := true);

assert occurrence nlog;
assert true alice.isLoggedIn;
assert true Account.allInstances()->size()=2;
```

We add some tasks in the IB as seen in Algorithm 7.3:

Algorithm 7.3 IB Task creation (i)

```
assert true alice.task->size()=0;
assert true Task.allInstances()->size()=0;
ntask := new CreateTask(
              descriptor := 'Go to the cinema with Bob',
              creationDate := 20111110,
              dueDate := 20111112,
              user := alice);

assert occurrence ntask;
assert true ntask.creationDate=20111110;
assert true alice.task->size()=1;
assert true Task.allInstances()->size()=1;
```

We add some other more tasks in the IB as seen in Algorithm 7.4:

Algorithm 7.4 IB Task creation (ii)

```
ntask2 := new CreateTask(
                    descriptor := 'Buy rice for lunch',
                    creationDate := 20111110,
                    dueDate := 20111112,
                    user := alice);

assert occurrence ntask2;
assert true alice.task->size()=2;
assert true Task.allInstances()->size()=2;

ntask3 := new CreateTask(descriptor := 'Go to the post
                    office to send a package to Carol',
                    creationDate := 20111110,
                    dueDate := 20111114,
                    user := alice);

assert occurrence ntask3;
assert true alice.task->size()=3;
assert true Task.allInstances()->size()=3;
```

We assign some priorities to tasks as seen in Algorithm 7.5:

Algorithm 7.5 IB Priority assignation

```
taskCinema := [Task.allInstances()->any(t |
          t.descriptor = 'Go to the cinema with Bob' and
          t.dueDate = 20111112 and
          t.account.username->includes('Alice'))];

assert true taskCinema.prio->isEmpty();
prio1 := new CreatePrio(user := alice,
                        task := taskCinema,
                        level := 1);

assert occurrence prio1;
assert true alice.task->size()=3;
assert true Task.allInstances()->size()=3;
assert true taskCinema.prio.level=1;
```

Finally, we are ready to reach the test objective as seen in Algorithm 7.6:

Algorithm 7.6 Priority update test (i)

```
assert occurrence prio1;
assert true alice.task->size()=3;
assert true Task.allInstances()->size()=3;
assert true taskCinema.prio.level=1;

prioUp2 := new UpdatePrio(user := alice,
                          task := taskCinema,
                          prioNew := 2);

assert occurrence prioUp2;
assert true alice.task->size()=3;
assert true Task.allInstances()->size()=3;
assert true taskCinema.prio.level=2;
assert consistency;
```

7.5.1.2. Test case 2: Increasing the priority of a task

Test objective: Increment the current priority of a task (Extension 4a sec. 4.5.1.1).

The IB can be build like in the first test case. That is, we assume the code from Algorithm 7.1, Algorithm 7.2, Algorithm 7.3, Algorithm 7.4 and Algorithm 7.5 to be already defined. Therefore, we can directly test the objective in Algorithm 7.7.

Algorithm 7.7 Priority update test (ii)

```
test ModificationOfAPriorityIncrease {
    //the test name should be the one above
    //and the code from figures should go here

    prioInc := new UpdatePrio(user := alice,
                              task := taskCinema,
                              increase := true);

    assert occurrence prioInc;
    assert true alice.task->size()=3;
    assert true Task.allInstances()->size()=3;
    assert true taskCinema.prio.level=2;
    assert consistency;
```

7.5.1.3. Test case 3: Decreasing the priority of a task

Test objective: Decrement the current priority of a task (Extension 4b sec. 4.5.1.1).

The IB can be build almost like in the first test case. That is, we assume the code from Algorithm 7.1, Algorithm 7.2, Algorithm 7.3 and Algorithm 7.4 to be already defined. Therefore, we only need to assign some task priorities as seen in Algorithm 7.8 before we can directly test the objective in Algorithm 7.9.

Algorithm 7.8 IB Priority assignation

```
taskCinema := [Task.allInstances()->any(t |
          t.descriptor = 'Go to the cinema with Bob' and
          t.dueDate = 20111112 and
          t.account.username->includes('Alice'))];

assert true taskCinema.prio->isEmpty();
prio1 := new CreatePrio(user := alice,
                        task := taskCinema,
                        level := 3);

assert occurrence prio1;
assert true alice.task->size()=3;
assert true Task.allInstances()->size()=3;
assert true taskCinema.prio.level=3;
```

Algorithm 7.9 Priority update test (iii)

```
test ModificationOfAPriorityDecrease {
    //the test name should be the one above
    //and the code from figures should go here

    prioDec := new UpdatePrio(user := alice,
                              task := taskCinema,
                              decrease := true);

    assert occurrence prioDec;
    assert true alice.task->size()=3;
    assert true Task.allInstances()->size()=3;
    assert true taskCinema.prio.level=2;
    assert consistency;
```

7.5.1.4. Test story 4: Failed update of a priority (priority out of bounds)

Test objective: Attempt to increment a priority from a task which was already set as the highest priority (Extension 6a + Extension 4a sec. 4.5.1.1).

The IB can be build like in the first test case. That is, we assume the code from Algorithm 7.1, Algorithm 7.2, Algorithm 7.3, Algorithm 7.4 and Algorithm 7.5 to be already defined. Therefore, we can directly test the objective in Algorithm 7.10.

Algorithm 7.10 Priority update test (iv)

```
test InvalidModificationOfAPriorityOutOfBounds {
    //the test name should be the one above
    //and the code from figures should go here

    prioDec := new UpdatePrio(user := alice,
                              task := taskCinema,
                              decrease := true);
    assert non-occurrence prioDec;

    prioInc1 := new UpdatePrio(user := alice,
                               task := taskCinema,
                               increase := true);

    assert occurrence prioInc1;
    assert true taskCinema.prio.level=2;

    prioInc2 := new UpdatePrio(user := alice,
                               task := taskCinema,
                               increase := true);

    assert occurrence prioInc2;
    assert true taskCinema.prio.level=3;

    prioInc3 := new UpdatePrio(user := alice,
                               task := taskCinema,
                               increase := true);

    assert non-occurrence prioInc3;
    assert consistency;
```

7.5.1.5. Additional notes

The first difference we notice is that test cases need to build up a Information Base (IB) before they can test and execute the case itself. This is because test stories were written in natural language and this information was implied. The IB was treated as having been introduced earlier. But because all test cases must be independent one from the other, each one has to build its own IB, so essentially they remain the same as test stories in this aspect.

One other difference we see are assertions. Test stories assume that the system will "know" when there is a wrong input or when use cases end with the expected result. This knowledge that is implied in test stories has to be made explicit in test cases, and assertions are precisely the way to do it. Furthermore, we see that all test cases must end with "assert consistency". It is the correct way to verify that there have not been unexpected changes that would make the IB inconsistent at the end of case.

The lack of an interface and real "interaction" between user and system is another difference. This interaction is not necessary to test a conceptual schema, and almost everything in relation to it is dropped in test cases (e.g. system messages).

There are also some possible legitimate test stories that would not make any sense as test cases in Conceptual Schema Testing Language (CSTL). One example would be a test story that does not input all the mandatory parameters, but asks the system to proceed anyway. One such test case would not be valid, because to execute a method, the test case needs all its parameters. This would simply output "Error" and not a "Pass", so it does not make sense to translate this kind of stories.

On the other hand, there are test cases that would make sense, even if they would not from a test story standpoint. One example would be testing wether a user can update a priority without being logged in or not. In a test story it would not make sense, because in order to access to tasks one must log in first, as the interface already places descriptions, but CSTL, being more technology independent and general makes this test case a valid one, and necessary if all the invariants want to be checked. This case is coded as follows in Algorithm 7.11, assuming the code from Algorithm 7.1, Algorithm 7.2, Algorithm 7.3, Algorithm 7.4 and Algorithm 7.5 to be already defined.

Algorithm 7.11 Priority update test (v)

```
test InvalidModificationOfAPriorityNotLoggedIn {
    //the test name should be the one above
    //and the code from figures should go here

    nlogout := new LogOutAccount(user := alice);
    assert occurrence nlogout;
    assert false alice.isLoggedIn;

    prioUp := new UpdatePrio(user := alice,
                             task := taskCinema,
                             prioNew := 2);

    assert non-occurrence prioUp;
    assert consistency;
```

Finally another example of test cases that make sense but not as test stories. We assume the code from Algorithm 7.1, Algorithm 7.2, Algorithm 7.3, Algorithm 7.4 and Algorithm 7.5 to be already defined. Therefore, we only need to look at Algorithm 7.12.

Algorithm 7.12 Priority update test (vi)

```
test InvalidModificationOfAPriorityContradictoryAction {
    //the test name should be the one above
    //and the code from figures should go here

    prioIncDec := new UpdatePrio(user := alice,
                                 task := taskCinema,
                                 increase := true,
                                 decrease := true);

    assert non-occurrence prioIncDec;
    assert consistency;
```

7.6. Iterations

The Test-Driven Conceptual Modelling (TDCM) methodology states that, according to Test-Driven Development (TDD), each iteration must consist on one test. Nevertheless, it also suggests that, as the modeller gains confidence, he/she may introduce more than one new test each iteration.

In the first few iterations, we only introduced one test at a time, but as the project went on, we introduced several of them, even spanning more than one new use case each time. These iterations, which are a summary[8] of their final outcome, are showed to grasp more details about the execution of TDCM. The algorithm followed to determine which use case use is described in section 6.2. Classes, invariants, relationships, etc. are created as they are required.

7.6.1. Iteration 1

This iteration involves the creation of the conceptual schema, because it did not exist yet, and the first use case from section 6.6. This use case is "Create account".

7.6.1.1. Test case codification

We encode a test similarly to how we did in section 7.5. This test is basically the one seen in Algorithm 7.1, with its test name being "CreationOfAnAccount" and ending with the instruction "assert consistency;".

7.6.1.2. Conceptual schema codification

Being the first use case, we had to create the model. Not having any class of type "Account", we had to define it with its required attributes (Algorithm 7.13).

Algorithm 7.13 Account class

```
model RememberTheMilk

class Account
        attributes
            username: String
            password: String
            creationDate: Integer
            email : String
end
```

At that point, we were ready to define the "Create account" event (Algorithm 7.16).

[8] We are not showing all the steps the whole process required (e.g. error messages as they appeared during time or the modifications that were performed to correct them).

Algorithm 7.14 Create account event

```
event CreateAccount
    attributes
        username: String
        password: String
        creationDate: Integer
    operations
        effect()
end
```

Along with its postcondition (Algorithm 7.15).

Algorithm 7.15 Create account postcondition

```
context  CreateAccount::effect()
  post:  Account.allInstances()->exists(a | a.oclIsNew() and
                        a.username = self.username and
                        a.password = self.password and
                        a.creationDate = self.creationDate)
```

It is noteworthy that all its final attributes and relationships may not be present. That is because they are not needed at this point. For example, as soon as we introduce the "Log into an account" use case, the account must be able to keep track for each user. A boolean attribute that indicates this, is the decision we took. Another alternative, would have been creating subclasses of account, one which was LoggedInAccount and another which represented NotLoggedInAccount. In any case, modifications to some already codified classes and invariants are to be expected as the size of the conceptual schema grows.

7.6.1.3. Methods codification

In Algorithm 7.16 we can see the codification for the "Create Account" method.

Algorithm 7.16 Create account method

```
method CreateAccount{
    res := new Account(username := self.username,
                       password := self.password,
                       creationDate := self.creationDate,
                       email := '');
}
```

7.6.2. Iteration 2

7.6.2.1. Test case codification

We encode the second test case for "Create account" (Algorithm 7.17).

Algorithm 7.17 Create account test (ii)

```
test FailedCreationOfAnAccountDupplicatedName{
    na := new CreateAccount(username := 'Alice',
                            password := '12345678',
                            creationDate := 20111106);

    assert occurrence na;
    assert equals na.username 'Alice';
    assert equals na.password '12345678';
    assert true Account.allInstances()->size()=1;

    na := new CreateAccount(username := 'Alice',
                            password := '1234',
                            creationDate := 20111106);

    assert non-occurrence na2;
    assert true Account.allInstances()->size()=1;
    assert consistency;
```

7.6.2.2. Conceptual schema codification

Some modifications had to be done in order for the new test to get a "Pass" verdict. Some "Account" invariants were added (Algorithm 7.18).

Algorithm 7.18 Account invariants (i)

```
context Account inv usernameAccountIdentifier:
    Account.allInstances->isUnique(username)
```

And some "Create account" preconditions were added as well (Algorithm 7.19).

Algorithm 7.19 Create account preconditions (i)

```
context CreateAccount ini inv usernameAccountIdentifier:
    Account.allInstances()->forAll(a|
                                   a.username <> self.username)
```

7.6.2.3. Methods codification

Methods only need to be codified or modified when a new use case is introduced. Therefore, there are no changes at this point.

7.6.3. Iteration 3

7.6.3.1. Test case codification

We encode the third and last test case for "Create account" (Algorithm 7.20).

Algorithm 7.20 Create account test (iii)

```
test FailedCreationOfAnAccountNameTooLong{
    na := new CreateAccount(username := 'Alice Pleasance
                                        Lidell from Wonderland,
                                        beneath Oxfordshire',
                            password := '1234',
                            creationDate := 20111106);

    assert non-occurrence na;
    assert true Account.allInstances()->size()=0;
```

7.6.3.2. Conceptual schema codification

Additional code had to be included in order for the new test to get a "Pass" verdict. This are the "Account" invariants that were added (Algorithm 7.21).

Algorithm 7.21 Account invariants (ii)

```
context Account inv usernameMaxLength:
    Account.allInstances->forAll(a| a.username.size() >= 2
                                and a.username.size() <= 50)

context Account inv passwordMaxLength:
    Account.allInstances->forAll(a| a.password.size() >= 4
                                and a.password.size() <= 50)
```

And some "Create account" preconditions were added as well (Algorithm 7.22).

Algorithm 7.22 Create account preconditions (ii)

```
context CreateAccount ini inv usernameMaxLength:
    self.username.size() <= 50 and self.username.size() >= 2

context CreateAccount ini inv passwordMaxLength:
    self.password.size() <= 50 and self.password.size() >= 4
```

7.6.3.3. Methods codification

As explained, there are no new methods.

7.6.4. Iteration 4

Having already codified and finished the "Create account" use case, we followed with the next one in the list, "Log into account". In this iteration we revisited the "Account" class definition, to include the knowledge required for a user to be able to log in. We added a new event "LogIntoAccount", with its postcondition, method and some preconditions. The rest of the process is analogous to that of previous iterations.

Part III.

Results

8. Conceptual schema

8.1. Overview

This chapter is about the obtained Remember the Milk (RTM) conceptual schema through Test-Driven Conceptual Modelling (TDCM) application.

In section 8.2 we see two different views of the structural schema, with its classes and their associations, multiplicities and attributes. In section 8.3 we show the event definition along with its postconditions and invariants of a relevant test case. The rest of them are available in Appendix C. In section 8.4 there is a small summary of statistics.

8.2. Structural schema

In Figure 8.1 we see RTM's partial structural schema. The image is a screenshot from the program Conceptual Schema Testing Language (CSTL) Processor uses to check conceptual schema correctness (USEx). It shows a Unified Modelling Language (UML) class diagram containing all classes belonging to the 50 use cases (out of a total of 65) with the highest priority, according to the assessment we did in chapter 6.

In this class diagram, we also see the different associations between classes and their multiplicities. These classes and associations were created in order to get "Pass" verdicts to all tests, which expressed the RTM system structural knowledge. As we can see, these class diagram correctly represents the RTM knowledge.

In Figure 8.2 the attributes belonging to each class are shown. There are no association names or multiplicity numbers, because this way the image is clearer, but they remain the same as in Figure 8.1.

8.3. Behavioural schema

Not visible in the structural schema, would be the events and their postconditions. We present again that of "Update priority of a task", as we did earlier (section 4.5, subsection 5.5.1 and subsection 7.5.1).

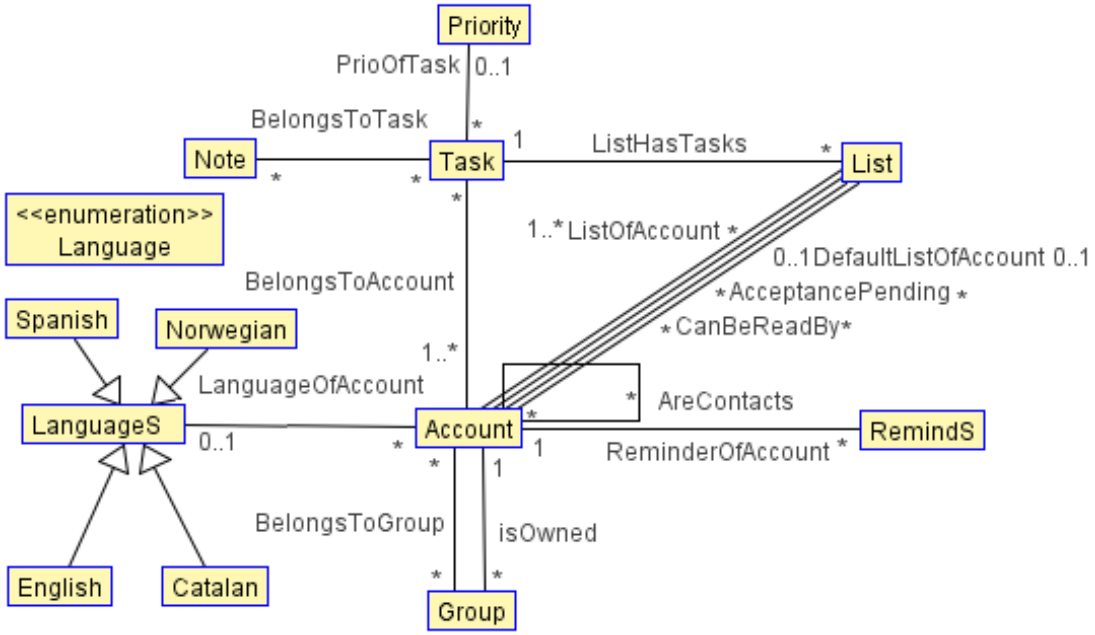

Figure 8.1.: RTM Partial conceptual schema (i)

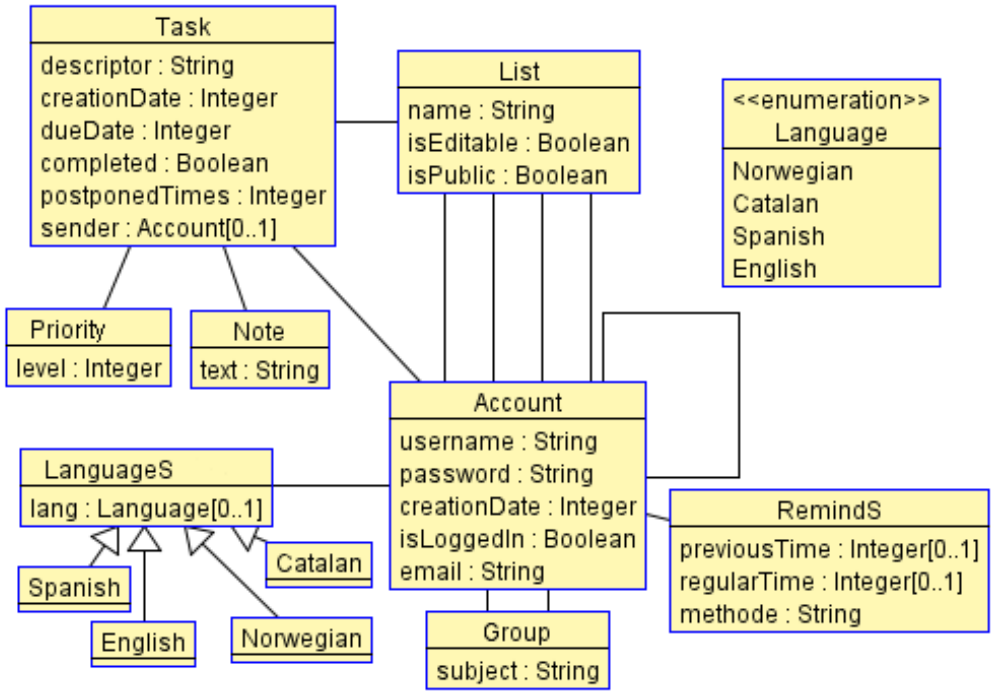

Figure 8.2.: RTM Partial conceptual schema (ii)

In Algorithm 8.1 we can see the event definition.

Algorithm 8.1 Update priority definition

```
event UpdatePrio
    attributes
        user: Account
        prioNew: Integer[0..1]
        increase: Boolean[0..1]
        decrease: Boolean[0..1]
        task: Task
    operations
        effect()
    end
```

In Algorithm 8.2 we can see its postcondition.

Algorithm 8.2 Update priority postcondition

```
context UpdatePrio::effect()
  post: Prio.allInstances()->exists(p |
    -- first clause
    ((self.increase.isUndefined() and self.increase.isUndefined())
       or (not (self.increase=true) and not (self.increase=true)))
       implies p.level = self.prioNew
    and -- second clause
    ((self.increase=true) and not (self.decrease=true)
       and p.level@pre < 3) -- 3 is the max. prio. level
       implies p.level = p.level@pre +1
    and -- third clause
    ((self.decrease=true) and not (self.increase=true)
       and p.level@pre > 1) -- 1 is the min. prio. level
       implies p.level+1 = p.level@pre
    and -- fourth clause, unconditional
    p.task->includes(self.task)
    and -- fifth clause, unconditional
    p.task.account->includes(self.user))
```

And finally inAlgorithm 8.3 we observe the event preconditions.

Algorithm 8.3 Update priority preconditions

```
context UpdatePrio ini inv alreadyPrio:
    self.task.prio->notEmpty()

context UpdatePrio ini inv loggedUser:
    self.user.isLoggedIn = true

context UpdatePrio ini inv notIncreaseAndDecrease:
    not ((self.increase=true) and (self.decrease=true))

context UpdatePrio ini inv notOutOfUpperBounds:
    Prio.allInstances()->exists(p | ((self.increase=true)
            and not (self.decrease=true) implies p.level < 3)
            and p.task->includes(self.task)
            and p.task.account->includes(self.user))

context UpdatePrio ini inv notOutOfLowerBounds:
    Prio.allInstances()->exists(p | (not (self.increase=true)
                and (self.decrease=true) implies p.level > 1)
                and p.task->includes(self.task) and
                    p.task.account->includes(self.user))
```

8.4. Statistical summary

All this statistical data that makes reference to the conceptual schema is provided by USEx.

- The schema has 41 events[1] and 41 postconditions (one of each event).
- It has 12 different classes, 4 of them being generalisations. Their association count is 13 and their total number of attributes is 21.
- The schema has 250 invariants + preconditions.
- The number of different test cases that drove the development of this schema is 79.

[1] This would be equivalent to 41 use cases instead of 50, but this is because we did not implement any use case whose type was query. They do not change the conceptual schema, as no additional attributes, classes or associations are needed for them. So the sentence where we said this represents the 50 top priority use cases remains true.

9. Lessons learnt

9.1. Overview

The aim of this chapter is to describe what we have learnt from this project, focusing on a retrospective view of the goals set in section 1.5.

In section 9.2 we describe the gathered experiences from the application of Test-Driven Conceptual Modelling (TDCM) on a real-sized project. In section 9.3 the complementary goal of suggesting modifications to the Remember the Milk (RTM) system is explored.

9.2. TDCM

9.2.1. Feasibility

Beyond any doubt, this project reinforces the idea that TDCM is a viable solution to build conceptual schemas of real-sized information systems. The schemas in chapter 8, obtained by the application of TDCM, are a proof of this fact, because they show the almost complete schema of RTM, a real-sized information system.

9.2.2. Advantages

At every point in its evolution, including the final version, schemas created by applying TDCM are always correct and complete with respect to the defined test cases. Such confidence is usually not possible to achieve when schemas are developed using a method that does not test them.

We have also showed that it is possible for stakeholders to define test stories in natural language that represent the system's knowledge. These test stories can later be translated into test cases. This active involvement of everyone that knows the domain helps to ensure even more that all the knowledge is valid and relevant according to their expectations and needs. Again, this approach may be not possible with other methodologies when stakeholders lack technical skills in computer science and more specifically in software engineering.

9.2.3. Patterns of use

We have noticed that, because several test cases must be written for each use case at conceptual modelling stage, the development of a conceptual schema by applying TDCM is bound to take more time than applying other methodologies. Nevertheless, in most cases, this extra time should be small relative to the overall time spent.

Furthermore, because validity is ensured through stakeholder participation, we can state that the probability of the final system not meeting stakeholders' expectations due to the conceptual schema is greatly reduced. Additionally, because completeness and correctness are ensured through automated testing, we can guarantee, as long as tests correctly represent all the knowledge, that the conceptual schema is error-free and represents exactly what we meant.

Modification of requirements because the conceptual schema does not represent what stakeholders expected, specially at later stages (design, implementation, etc.) can be extremely time consuming and resource intensive. Likewise, the resources needed to fix an schema error detected at stages later than conceptual modelling, is orders of magnitude higher. Because the correct application of TDCM prevents and minimises these situations, we believe that the whole development time and resources spent on a complete information system implementation will be significantly lower in most cases, despite the conceptual modelling stage being usually longer.

Regarding time spent in subparts of TDCM application, our findings, which mostly support previous TDCM applications [21, 18], are the following ones :

1. Iteration-wise, most of them took about the same time to resolve (i.e. get a "Pass" verdict). This may look contradictory in relation to other project results, because as the modeller familiarises with the domain and the language, the time spend per iteration is expected to diminish. However, I chose to add more tests per iteration instead of just reducing their time. Hence, in general, the amount of time spent per test resolution was in fact decreasing.
2. The frequency of some types of errors varied along the TDCM application. For example, syntax errors were much more frequent during the first iterations, as I was not familiar with the Conceptual Schema Testing Language (CSTL).
3. There is not always a strong correlation between the time spent on a test to obtain a "Pass" verdict and its number of errors/failures fixed. On one hand, there are some kinds of errors, like missing relevant types which are trivial to be fixed (they just need to be added). On the other hand, fixing a failing assertion may require the modification of postconditions and invariants.
4. On average, the time spent writing test cases is much smaller than the time needed to pass them. On one hand, this is because most of the test cases are built on other test cases' Information Bases (IB), so only few new events and few new assertions are needed. On the other hand, the modification of the schema and/or methods can require much more thought and time. This means

that most of the time is actively spent modifying the conceptual schema, and not just writing tests.

9.3. RTM

9.3.1. Conceptual deficiencies

Thanks to the developed conceptual schema, we found that the system has no way to differentiate the creator and original owner of a task/list from other users that have had this task/list shared. Real-life testing of RTM system confirmed those findings, which can lead to many different issues:

1. The original owner can not control who has modification or viewing rights on the task/list once he/she shares it. The user that receives sharing rights, can share it with more other users without the general owner being able to do anything to prevent it.
2. The original owner of a list is unable to know who has sharing or reading rights for it.
3. Everyone that has sharing rights on a task can see who else has those rights.
4. Anyone from the sharing parties can delete tasks, without any other parties even getting any kind notice of this.

Most of these points are undesirable in a number of situations. We think that the original owner of a task/list, if he/she wanted so, should have more rights on it, and could control more precisely which rights to grant to other parties. A simple addition of an association between the "Account" class to "List" and another one to "Task" would allow for the development of use cases that permitted this. These associations should probably have a multiplicity of 1 on the "Account" end and an * on the other ends. This multiplicities would roughly mean that each list/task must have one and only one original owner, and that each account can be the original owned of many different tasks/lists. All this, again, should be developed in accordance with TDCM, writing tests that represent the knowledge, etc.

The introduction of this association would also enable us to add a few more use cases that were missed. This family of "Unshare" use cases would allow the original owner of a task/list to revoke granted sharing privileges to other parties. It would be similar to "Unpublish" use cases, but they were not possible before because the system could not discern the original owner of a task/list.

9.3.2. Lack of validation

We included at least a validation step in all use cases that had to deal with user input. RTM does not do this in most cases, or its ability to check for errors is

highly defective. One example would be the user's ability to create arbitrarily long task names. There was no apparent limit, so we created a ~150,000 characters long task name. RTM did not trunk the display of the task name, so when the task was listed among the others, it was completely disrupting the interface and hindering its usability considerably. Furthermore, a long enough task name could possibly crash the account or the entire system. We solved this and other lack of validation problems by adding invariants which prevented them when needed.

9.3.3. Additional features

With the Kano[1] process of categorisation of use cases, we found that the system is lacking "excitement" features, because the ones it actually has are going to be downgraded to "performance" soon. Some interesting "excitement" features are presented.

1. Social wish list: items in this special kind of list, would consist on activities you intend or would like to do, but you would prefer to do in a group. The system would aggregate them, and possibly use geolocation algorithms to provide better matchings. Practical applications for this range from the rental of a bus for a trip of a large group of people willing to go to an international music festival to car-sharing.
2. List subscription: this would be the ability for a user to search for public published lists and "subscribe" to them. The system would notify the user of updates as the list is modified. Practical applications for this could be someone releasing weekly recipes and their required ingredients list.
3. Tag[2] subscription: this would be the ability for a user to search for (or create) tags and subscribe to them. The system would notify him/her of updates as new tasks with the correct permissions are assigned with those tags. Practical applications for this could be the subscription of a football team or a specific type of music.
4. Work request list: each user would have a list where they could request assistance in some kind of task. Qualified workers would get permission to see the tasks related to their field. Its philosophy would not be much different than that of craigslist[3]. Furthermore geolocation algorithms could be taken into account to provide even a better service. Practical applications of this would range from soliciting a plumber to fix a pipe to soliciting a baby sitter for the weekend.
5. Service recommendation: this would be the ability for RTM to analyse the tasks/notes/etc. in search for keywords. With this information, RTM would

[1] See section 6.4

[2] Tags for tasks actually exist in RTM, but we considered them already included in the "Update task" use case, as just another attribute.

[3] is a centralised network of online communities featuring free online classified advertisements.

recommend the user related products or services. A practical application for this would be the highlighting of a disc title to a spotify, amazon, youtube or itunes link. The user could disable the feature or restrict it if he/she did not like it.

9.3.4. Counterintuitive Interface

Graphical User Interfaces (GUIs) are not the main topic on this project. However, a study should be carried on to provide data to fix the GUI. One example of things to fix would be list-related use cases. In order to create a new list, a user must go to the "Settings" tab, then to the "Lists" tab and finally click on the smallish "Add list" button for a new blank input box to appear and fill its name. It probably is too many clicks and steps, but the inevitable problem is that "List" actions are not unified. In order to share a list, a user must go to the "Tasks" tab, choose the list and then share it. Having two such different processes for dealing with similar things can be confusing and difficult to remember for end-users.

Another example in the "List" category, would be the fact that a user is not told in any way when he/she has a new shared list pending of acceptance. The only means a user has to know it is by regularly checking the "Settings" tab, then the "Lists" tab and finally searching thoroughly for the small acceptance pending icon. This is an unnecessary waste of time apart from being inconvenient.

10. Conclusions and further work

10.1. Overview

The aim of this chapter (section 10.2) is to revise the accomplishment levels of the goals set in section 1.5 by summarising chapter 9. In section 10.3 we give the next steps a project should follow if it was to continue the work presented in this document, embracing related objectives.

10.2. Conclusions

10.2.1. Test-Driven Conceptual Modelling (TDCM)

- TDCM is a viable solution to build conceptual schemas of real-sized information systems (subsection 9.2.1).
- TDCM-created schemas offer a higher degree of correctness and completeness than methods with non-tested schemas (subsection 9.2.2).
- TDCM ensures that all stakeholder's domain knowledge is represented in the schema (subsection 9.2.2).
- Conceptual modelling takes some more resources applying TDCM than other methodologies, because of test case definitions (subsection 9.2.3).
- TDCM minimises conceptual schema errors and ensures that stakeholders' expectations are met. This causes a significant reduction over total development costs (subsection 9.2.3).
- Our findings in TDCM application subparts support previous work[21, 18] (subsection 9.2.3).
  - In average, time spent per test resolution decreased with each iteration.
  - Type of error frequency varied along the TDCM application.
  - Time spent to "Pass" a test does not correlate with its number of errors/failures fixed.
  - Time spent writing test cases is much smaller than spent modifying the schema.

10.2.2. Remember the Milk (RTM)

- Necessary conceptual schema elements were missing and corrections were proposed (subsection 9.3.1).
- User input validation was deemed insufficient (subsection 9.3.2).
- Additional "excitement" features according to Kano[1] were suggested to replace RTM's current ones (subsection 9.3.3).
- RTM was found to have usability issues (subsection 9.3.4).

10.3. Further work

10.3.1. CSTL Processor

The Processor is still in prototype stage, and even if it is functional, there is room for improvement. A list of suggestions is presented.

Portability: the system does not work neither in Mac OS X nor in Linux. However, because it is java-based, it should be relatively easy to port it to those and other platforms.

Speed: test-case execution speed is okay in an individual basis[2]. Nevertheless, everything indicates that for bigger systems, the algorithm execution time needs to be diminished.

Bugs: we detected some bugs, which have been reported to Albert Tort, the creator of the tool[3].

Shortcuts: a great and simple addition would be the ability to have some sort of shortcuts for the most common tasks. Ctrl + S to save the current file and another one to execute sets of test-cases.

Statistics: keeping track of statistics automatically, including complex or customisable ones like the number of necessary changes for each type of error per iteration, would help to gather valuable information about use patterns.

[1] See section 6.4

[2] When several test cases (say >20) were to be tested for each modification, the time it took started to be annoying. When the number increased even more (say >40) the time it took was simply not tolerable. I just removed all the previous tests and retested them every few iterations to make sure they were still valid, but this is somehow against TDCM. The problems could be related to my computer (a 2-year-old macbook pro) running the processor in a virtual machine.

[3] One of them was related to the built-in text editor. When a test file is opened, if the editor detects the click of the mouse on it followed by the press of any keys (for example arrow keys) it believes that the file has been modified. When the modeller chooses to see another test case he/she is incorrectly informed that there are unsaved changes and is asked to save them.

Other suggestions I would add were already mentioned in another project[17], but they are still not completed:

Debugging: having access to the IB state at any moment would improve the process. This could be accomplished by the addition breakpoints.

Error messages: more descriptive error messages that would help the modeller detect and fix them.

10.3.2. TDCM Methodology

The TDCM methodology relies on test cases to drive the modeller to add domain knowledge into the conceptual schema. Nevertheless, it could be possible for a methodology, let's call it TDCM+, to build the schema itself from the test cases and the methods file alone. For example, in this hypothetical TDCM+, when testing detected a missing type, instead of outputting an "Error" verdict and stopping execution, it would add the relevant type to the schema. For other kind of errors or failures, this process could be much more complex, but the same philosophy could be applied. The TDCM+ cycle would then be reduced to (1) writing a new test case and (2) refactoring the resultant schema.

10.3.3. Experimentation

TDCM experimentation should not end with this project, as there are still more options to explore. Some of them are presented here.

Big scale projects. TDCM has not yet been tested in big projects. Such projects could be for example Enterprise Resource Planning (ERPs) with many modules like Customer Relationship Manager (CRM), Business Intelligence (BI), web services, etc.

Non-existing real-sized systems. TDCM has not yet been used to develop the conceptual schema of any real-sized projects which did not exist (unlike RTM).

Finishing the remaining use cases from the RTM system would not be of much value, as all the goals set have already been accomplished.

Part IV.

Appendixes and Bibliography

A. Use case specification

A.1. Task

A.1.1. Create a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to create a new task.

Main success scenario:

1. The actor indicates the system that he/she wants to create a new task.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor introduces a task (i.e. a sentence or string that describes a task).
4. The system validates all the information the actor has introduced.
5. The system creates a new task with the introduced information.
6. The system notifies the actor that the task has been successfully created.

Extensions:

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.2. Read a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to access to all the information regarding an existing task.

Main success scenario:

1. The actor indicates the system that he/she wants to read an existing task.
2. The system provides the actor with a representation which contains all currently existing tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with further information about the selected task.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

A.1.3. Update a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to modify information of an existing task.

Main success scenario:

1. The actor indicates the system that he/she wants to modify an existing task.
2. The system provides the actor with a representation which contains all currently existing tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with further information about the selected task.
5. The actor modifies some information about the selected task.
6. The system asks the actor for confirmation.

7. The actor confirms his/her modification/s.
8. The system validates all the information regarding the task.
9. The system updates the existing task with the modified information.
10. The system notifies the actor that the task has been successfully updated.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.4. Delete a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to access to all the information regarding an existing task.

Main success scenario:

1. The actor indicates the system that he/she wants to delete an existing task.
2. The system provides the actor with a representation which contains all currently existing tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system asks the actor for confirmation.
5. The actor confirms the deletion.
6. The system deletes the selected task.
7. The system notifies the actor that the task has been successfully deleted.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

A.1.5. Create an assign a note to task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to create a new note and add it to an existing task.

Main success scenario:

1. The actor reads a task[1].
2. The actor indicates the system that he/she wants to create a new note and add it to the task he/she has read in the previous step
3. The system provides the actor with a representation which contains all notes pertaining to the selected task and enough information to identify them.
4. The actor introduces text to a new blank note.
5. The actor indicates the system that he/she wants to save the note.
6. The system validates the introduced data.
7. The system creates a note with the introduced contents, additional metadata (e.g. date of creation) and associates it with the task.
8. The system notifies the actor that the note has been successfully created and assigned to the chosen task.

Extensions:

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.6. Read a note assigned to a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

[1] See sec. A.1.2

Trigger: The actor wants to read the information regarding an existing note from a task.

Main success scenario:

1. The actor reads a task[2].
2. The actor indicates the system that he/she wants to read an existing note from the task he/she has read in the previous step.
3. The system provides the actor with a representation which contains all notes pertaining to the selected task and enough information to identify them.
4. The actor selects one of the notes.
5. The system shows the actor the note's contents and some additional metadata (e.g. date of the note's creation).

Extensions:

2a. The task does not have any note associated with it:

1. The system notifies the actor.
2. The use case ends.

A.1.7. Update a note assigned to a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to modify the information regarding an existing note from a task.

Main success scenario:

1. The actor reads a task[3].
2. The actor indicates the system that he/she wants to modify an existing note from the task he/she has read in the previous step.
3. The system provides the actor with a representation which contains all notes pertaining to the selected task and enough information to identify them.
4. The actor selects one of the notes.

[2]See sec. A.1.2
[3]See sec. A.1.2

5. The system shows the actor the note's contents and some additional metadata (e.g. date of the note's creation).
6. The actor modifies some information about the selected note.
7. The system asks the actor for confirmation.
8. The actor confirms his/her modification/s.
9. The system validates all the information regarding the note.
10. The system updates the existing note with the modified information.
11. The system notifies the actor that the note has been successfully updated.

Extensions:

2a. The task does not have any note associated with it:

1. The system notifies the actor.
2. The use case ends.

9a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.8. Delete a note assigned to a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to delete an existing note from a task.

Main success scenario:

1. The actor reads a task[4].
2. The actor indicates the system that he/she wants to delete an existing note from the task he/she has read in the previous step.
3. The system provides the actor with a representation which contains all notes pertaining to the selected task and enough information to identify them.
4. The actor selects one of the notes.
5. The system asks the actor for confirmation.

[4]See sec. A.1.2

6. The actor confirms the deletion.
7. The system deletes the selected note.
8. The system notifies the actor that the note has been successfully deleted.

Extensions:

2a. The task does not have any note associated with it:

1. The system notifies the actor.
2. The use case ends.

A.1.9. Complete a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to complete an existing unfinished task

Main success scenario:

1. The actor indicates the system that he/she wants to complete an existing unfinished task.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system updates the existing task marking it as finished.
5. The system notifies the actor that the task has been successfully completed.

Extensions:

2a. The actor does not have any uncomplete task in the system:

1. The system notifies the actor.
2. The use case ends.

A.1.10. Uncomplete a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to uncomplete an existing finished task (i.e. mark a completed task as not finished).

Main success scenario:

1. The actor indicates the system that he/she wants to uncomplete an existing finished task.
2. The system provides the actor with a representation which contains all currently existing finished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system updates the existing task marking it as not finished.

Extensions:

2a. The actor does not have any complete task in the system:

1. The system notifies the actor.
2. The use case ends.

A.1.11. Set priority to a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to set a priority to an existing task.

Main success scenario:

1. The actor reads a task[5].
2. The actor indicates the system that he/she wants to modify the priority of task he/she has read in the previous step.

[5]See sec. A.1.2

3. The system provides the actor with a representation (e.g. a list or a colour schema) which contains all possible priorities.
4. The actor selects one priority from the representation.
5. The system validates the introduced data.
6. The system sets a priority for the task according to the chosen value by the actor.
7. The system notifies the actor that the priority has been successfully set to the task.

Extensions:

2a. The task already has a set priority:

1. The system notifies the actor.
2. The use case ends.

5a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.12. Update priority from a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to change the priority from an existing task.

Main success scenario:

1. The actor reads a task[6].
2. The actor indicates the system that he/she wants to modify the priority of task he/she has read in the previous step.
3. The system provides the actor with a representation (e.g. a list or a colour schema) which contains all possible priorities, modifications and the current value.
4. The actor selects one priority from the representation.

[6]See sec. A.1.2

5. The system temporarily sets the task's priority to the chosen level.
6. The system validates the introduced data.
7. The system sets a priority for the task according to the temporary value.
8. The system notifies the actor that the task's priority has been successfully updated.

Extensions:

4a. The actor wants to increase the current priority of the task:

1. The actor indicates the system that he/she wants to increase the priority of the task.
2. The system temporarily increments the task's priority by one level.
3. The execution returns to item 6 of the main success scenario.

4b. The actor wants to decrease the current priority of the task:

1. The actor indicates the system that he/she wants to decrease the priority of the task.
2. The system temporarily decrements the task's priority by one level.
3. The execution returns to item 6 of the main success scenario.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.13. Delete priority from a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: sec. A.1.2 Read a task

Trigger: The actor wants to delete the priority from an existing task.

Main success scenario:

1. The actor reads a task[7].
2. The actor indicates the system that he/she wants to delete the priority of task he/she has read in the previous step.

[7] See sec. A.1.2

3. The system asks the actor for confirmation.
4. The actor confirms the deletion.
5. The system deletes the selected priority.
6. The system notifies the actor that it has successfully deleted the priority from the task.

Extensions:

2a. The task does not have any priority set:

1. The system notifies the actor.
2. The use case ends.

A.1.14. Postpone a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to postpone an existing unfinished task.

Main success scenario:

1. The actor indicates the system that he/she wants to postpone an existing unfinished task.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an unfinished task.
4. The system updates the unfinished task making its due date the current day.
5. The system updates the information regarding the number of times this task has been postponed, incrementing it by one.
6. The system notifies the actor that the task's due date has been successfully postponed, along with the new date.

Extensions:

2a. The actor does not have any unfinished task in the system:

1. The system notifies the actor.
2. The use case ends.

4a. The due date of the task has not yet passed:

1. The system updates the unfinished task making the due date one day later than the currently set date.

A.1.15. Share a task with contacts

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to share an existing unfinished task with some of his/her contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to share an existing unfinished task with some of his/her contacts.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with a graphical representation of his/her contacts previously added.
5. The actor chooses the contacts he/she wants to share the task with.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the task information regarding who is the task shared with.
10. The system assigns the task to all the contacts that were chosen by the actor.
11. The system notifies the actor that the task has been successfully shared.

Extensions:

2a. The actor does not have any uncomplete task in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.16. Send a task to contacts

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to send an existing unfinished task with some of his/her contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to send an existing unfinished task to some of his/her contacts.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with a graphical representation of his/her contacts previously added.
5. The actor chooses the contacts he/she wants to send the task to.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the task information regarding who is the sender of the task.
10. The system assigns the task to all the contacts that were chosen by the actor.
11. The system moves the task to the actor's "Sent" list.
12. The system notifies the actor that the task has been successfully sent.

Extensions:

2a. The actor does not have any uncomplete task in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.17. Share a task with groups

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to share an existing unfinished task with some of his/her groups of contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to share an existing unfinished task with some of his/her groups of contacts.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with a graphical representation of his/her groups previously added.
5. The actor chooses the groups he/she wants to share the task with.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the task information regarding who is the task shared with.
10. The system assigns the task to all the contacts pertaining to the groups that were chosen by the actor.
11. The system notifies the actor that the task has been successfully shared.

Extensions:

2a. The actor does not have any uncomplete task in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The groups the actor chose do not contain any contact :

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.18. Send a task to groups

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to send an existing unfinished task with some of his/her groups of contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to send an existing unfinished task with some of his/her groups of contacts.
2. The system provides the actor with a representation which contains all currently existing unfinished tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with a graphical representation of his/her groups previously added.
5. The actor chooses the groups he/she wants to send the task to.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the task information regarding who is the sender of the task.
10. The system assigns the task to all the contacts pertaining to the groups that were chosen by the actor.
11. The system moves the task to the actor's "Sent" list.
12. The system notifies the actor that the task has been successfully sent.

Extensions:

2a. The actor does not have any uncomplete task in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The groups the actor chose do not contain any contact :

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.19. Show tasks

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to view some of his/her tasks according to some criteria.

Main success scenario:

1. The actor indicates the system that he/she wants to view some of his/her tasks according to some criteria.
2. The system provides the actor with the necessary means for him/her to decide a suitable ordering.
3. The actor decides which ordering he/she wants.
4. The system provides the actor with the necessary means for him/her to decide which tasks are going to be filtered out.
5. The actor decides which filtering he/she wants.
6. The system provides the actor with a representation which contains all his/her tasks that meet the filtering criteria ordered as he specified.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

3a. The actor does not chose any ordering criteria:

1. The system chooses a criteria by default (e.g. due date > priority > alphabetical > creation date).

6a. The actor does not have any task in the system that meet the chosen criteria:

1. The system notifies the actor.
2. The use case ends.

A.1.20. Move a task to a list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to move an existing task to one of his/her lists other than the one it already is in.

Main success scenario:

1. The actor indicates the system that he/she wants to move an existing task to one of his/her lists other than the one it already is in.
2. The system provides the actor with a representation which contains all currently existing tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system provides the actor with a graphical representation of all his/her lists, excluding the one the task is already in and "Sent".
5. The actor chooses the list he/she wants to send the task to.
6. The system asks the actor for confirmation.
7. The actor confirms the selections. The system validates the introduced data.
8. The system updates the task and list information regarding where does the task belong to.
9. The system adjusts the task sharing and visibility permissions, if necessary, to make it coherent with the list sharing and publishing properties.

10. The system notifies the actor that the task has been successfully moved.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

4a. The task is in the "Sent" list :

1. The system notifies the actor that no task can be moved from the "Sent" list.
2. The use case ends.

4b. The actor account does not have any more lists besides "Sent" and "Inbox" :

1. The system notifies the actor that there is no other list to send the task.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.1.21. Duplicate a task

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to duplicate an existing task of his/hers.

Main success scenario:

1. The actor indicates the system that he/she wants to duplicate an existing task.
2. The system provides the actor with a representation which contains all currently existing tasks and enough information to identify them.
3. The actor selects an existing task.
4. The system creates a copy of the task, with the same values for all user-visible attributes and keeping the same shared contacts.
5. The system notifies the actor that the task has been successfully duplicated.

Extensions:

2a. The actor does not have any task in the system:

1. The system notifies the actor.
2. The use case ends.

4a. The selected task was already completed:

1. The system marks the task as not completed.

A.2. Account

A.2.1. Create an account

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must not be logged into the system.

Includes: none.

Trigger: The actor wants to create a new account.

Main success scenario:

1. The actor indicates the system that he/she wants to create a new account.
2. The system provides the actor with an interface which contains the necessary means for him/her to create an account.
3. The actor introduces the necessary information (e.g. a username and a password).
4. The system validates all the information the actor has introduced.
5. The system creates a new account with the introduced information.
6. The system notifies the actor that the account has been created and is ready for use.

Extensions:

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.2.2. Log into an account

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must not be logged into the system.

Includes: Create an account (A.2.1).

Trigger: The actor wants to log into an existing account.

Main success scenario:

1. The actor indicates the system that he/she wants to log into an existing account.
2. The system provides the actor with an interface which contains the necessary means for him/her to log into an account.
3. The actor introduces the necessary information (e.g. a username and a password).
4. The system validates all the information the actor has introduced.
5. The system notifies the actor that he has successfully logged in.

Extensions:

1a. The actor does not have an account:

1. The actor creates an account (A.2.1)
2. The execution returns to the 1st step of the main success scenario.

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.2.3. Update an account

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to modify information of an existing account.

Main success scenario:

1. The actor indicates the system that he/she wants to modify an existing account.
2. The system shows the actor further information about the account.
3. The actor modifies some information about the account.
4. The system asks the actor for confirmation.
5. The actor confirms the modification/s.
6. The system validates all the information regarding the account.
7. The system updates the existing account with the modified information.
8. The system notifies the actor that the account has been successfully modified.

Extensions:

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.2.4. Delete an account

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete an existing account.

Main success scenario:

1. The actor indicates the system that he/she wants to delete an existing account.
2. The system asks the actor for authentication.
3. The actor authenticates to verify he/she really is the owner of the account (e.g. introduces the password again or submits a digital certificate).
4. The system validates the authentication.
5. The system asks the actor for confirmation.
6. The actor confirms the deletion.
7. The system logs off the actor.
8. The system deletes the account.

9. The system notifies the actor that the account has been deleted.

Extensions:

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The system logs the actor off.
3. The use case ends.

A.2.5. Log out of an account

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to log out of an existing account.

Main success scenario:

1. The actor indicates the system that he/she wants to log out of an existing account.
2. The system logs the user out.
3. The system notifies the actor that he/she is no longer logged in.

A.3. Reminder

A.3.1. Create a reminder schedule

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to create a new reminder schedule.

Main success scenario:

1. The actor indicates the system that he/she wants to create a reminder schedule.
2. The system provides the actor with an interface which contains all the necessary parameters to customise and define a reminder schedule (e.g. via which medium/s shall the reminders be sent or what time of the day should daily reminders be emitted).
3. The actor fills in the required information according to his/her own desires.
4. The system asks for confirmation.
5. The actor confirms the creation.
6. The system validates the data introduced by the actor.
7. The system creates the reminder schedule.
8. The system notifies the actor that the reminder schedule has been created successfully.

Extensions:

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.3.2. Read a reminder schedule

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to read all the information about his/her reminder schedule.

Main success scenario:

1. The actor indicates the system that he/she wants to read his/her reminder schedule.
2. The system provides the actor with an interface which contains all the necessary parameters to define a reminder schedule (e.g. via which medium/s shall the reminders be sent or what time of the day should daily reminders be emitted).

3. The system fills all the fields that define the reminder schedule in, according to the actor's previously introduced information.

Extensions:

3a. The actor does not have any reminder schedule :

1. All the fields which would contain the parameters appear blank (not filled in).
2. The system notifies the actor that he/she does not have reminder schedules.

A.3.3. Update a reminder schedule

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to modify an existing reminder schedule.

Main success scenario:

1. The actor indicates the system that he/she wants to create a reminder schedule.
2. The system provides the actor with an interface which contains all the necessary parameters to customise and define a reminder schedule (e.g. via which medium/s shall the reminders be sent or what time of the day should daily reminders be emitted).
3. The actor modifies the information contained in some of the parameters or fields according to his/her own desires.
4. The system asks for confirmation.
5. The actor confirms the modification.
6. The system validates the data introduced by the actor.
7. The system updates the reminder schedule.
8. The system notifies the actor that the reminder schedule has been updated successfully.

Extensions:

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.3.4. Delete a reminder schedule

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete an existing reminder schedule.

Main success scenario:

1. The actor indicates the system that he/she wants to delete a reminder schedule.
2. The system provides the actor with an interface which contains all the necessary parameters to customise and define a reminder schedule (e.g. via which medium/s shall the reminders be sent or what time of the day should daily reminders be emitted).
3. The actor selects a previously introduced medium (i.e. already in the system for this account).
4. The system asks for confirmation.
5. The actor confirms the deletion.
6. The system deletes the scheduled reminders for that medium.
7. The system notifies the actor that the reminder schedule has been deleted.

Extensions:

2a. The system does not have any specified reminder schedule medium:

1. The system notifies the actor.
2. The use case ends.

A.3.5. Send reminder

Scope: RTM task management system.

Primary actor: System.

Preconditions: none.

Includes: none.

Trigger: The clock time has been updated.

Main success scenario:

1. The system accesses the reminder schedule notification information of all of its user accounts.
2. The system checks the date these reminders should be sent.
3. The system checks if the reminders for these schedules has already been sent.
4. The system checks the mediums for each of these reminders that should have been sent by the current date and have not already been sent.
5. The system checks the tasks each of these reminders that have to be sent.
6. The system sends the tasks to be reminded via each medium according to the information gathered.

A.4. Customisation

A.4.1. Change language

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: none.

Includes: none.

Trigger: The actor wants to change the language of the display (interface).

Main success scenario:

1. The actor indicates the system that he/she wants to change the language.
2. The system provides the actor with an interface which contains all the available languages (system translations).
3. The actor chooses one language.
4. The system asks for confirmation.
5. The actor confirms the selection.
6. The system updates the interface according to the selected language.

A.4.2. Show weekly planner

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to see the chronologically ordered task plan for the week in a printable-friendly format.

Main success scenario:

1. The actor indicates the system that he/she wants to see his/her weekly plan.
2. The system provides the actor with a representation of all his/her tasks due in one week or less. They are ordered chronologically and are clearly separated so it is easy to discern which day each task (or group of tasks) is due.

A.5. Contact and group

A.5.1. Add a contact

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to add a contact to his/her account.

Main success scenario:

1. The actor indicates the system that he/she wants to add a contact.
2. The system provides the actor with an interface which contains the necessary means for him/her to add contacts.
3. The actor selects another user from the system.
4. The system validates all the information the actor has introduced.
5. The system adds the chosen user as a contact.
6. The system notifies the actor that the contact has been added successfully.

Extensions:

5a. The introduced username is not valid:

1. The system notifies the actor that the chosen username does not correspond to any user registered in the system.
2. The use case ends.

5a. The user is already a contact of the actor:

1. The system notifies the actor that the chosen user is already his/her contact and therefore can not be added again.
2. The use case ends.

A.5.2. Read a contact

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to read information about one of his/her own added contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to read on of his/her contacts.
2. The system provides the actor with a graphical representation of his/her contacts previously added.
3. The actor chooses one of the contacts.
4. The system provides the actor with additional information about the contact.

Extensions:

2a. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

A.5.3. Delete a contact

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete one of his/her own added contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to delete one of his/her contacts.
2. The system provides the actor with a graphical representation of his/her contacts previously added.

3. The actor chooses one of the contacts.
4. The system asks the actor for confirmation.
5. The actor confirms the deletion.
6. The system deletes the contact bind from the actor account.
7. The system notifies the actor that the user is no longer one of his/her contacts.

Extensions:

2a. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

A.5.4. Create a group

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to create a group.

Main success scenario:

1. The actor indicates the system that he/she wants to create a new group.
2. The system provides the actor with a representation which contains all currently existing groups the actor created.
3. The actor introduces a group name (i.e. a sentence or string that describes a group).
4. The system validates all the information the actor has introduced.
5. The system creates a new group with the introduced information.
6. The system notifies the actor that the group has been created successfully.

Extensions:

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.5.5. Read a group

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to read information about one of his/her own created groups.

Main success scenario:

1. The actor indicates the system that he/she wants to read an existing group.
2. The system provides the actor with a representation which contains all currently existing groups the actor created.
3. The actor selects an existing group.
4. The system provides the actor with further information about the selected group.

Extensions:

2a. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

A.5.6. Delete a group

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete one of his/her own created groups.

Main success scenario:

1. The actor indicates the system that he/she wants to delete an existing group.
2. The system provides the actor with a representation which contains all currently existing groups the actor created.
3. The actor selects an existing group.

4. The system asks the actor for confirmation.
5. The actor confirms the deletion.
6. The system deletes the selected group.
7. The system notifies the actor about the deletion of the selected group.

Extensions:

2a. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

3a. The selected group is not empty (it still has contacts associated with it):

1. The system notifies the actor informing that this group still has at least one contact assigned to it, and therefore, it can not be deleted.
2. The use case ends.

A.5.7. Add a contact to a group

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to add one of his/her contacts to one of his/her own created groups.

Main success scenario:

1. The actor indicates the system that he/she wants to add a contact to one of his/her groups.
2. The system provides the actor with a graphical representation of his/her contacts previously added.
3. The actor indicates the system that he/she wants to add one of the contacts to a group.
4. The actor chooses one of the contacts.
5. The system provides the actor with a graphical representation of his/her groups previously added.
6. The actor chooses one of the groups.
7. The system asks the actor for confirmation.

8. The actor confirms the action.
9. The system adds the selected contact to the selected group.
10. The system notifies the actor about the insertion of the contact into the selected group.

Extensions:

2a. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The selected contact is already in the selected group:

1. The system notifies the actor that this contact is already in that group, and therefore it can not be added.
2. The use case ends.

A.5.8. Remove a contact from a group

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to remove one of his/her contacts form one of his/her groups.

Main success scenario:

1. The actor indicates the system that he/she wants to remove a contact from one of his/her groups.
2. The system provides the actor with a graphical representation of his/her contacts previously added.
3. The actor indicates the system that he/she wants to remove one of the contacts from a group.
4. The actor chooses one of the contacts.

5. The system provides the actor with a graphical representation of all the groups this contact belongs to.
6. The actor chooses one of the groups.
7. The system asks the actor for confirmation.
8. The actor confirms the action.
9. The system removes the contact from the selected group.
10. The system notifies the actor about the removal of the chosen contact from the selected group.

Extensions:

2a. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

5a. The selected contact does not belong to any group:

1. The system notifies the actor informing that this contact does not belong to any group, and therefore, it can not be removed from any.
2. The use case ends.

A.6. List of tasks

A.6.1. Create a list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to create a new list of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to create a new list.

2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor introduces a list name.
4. The system validates all the information the actor has introduced.
5. The system creates a new empty list with the introduced information.
6. The system notifies the actor that the list has been successfully created.

Extensions:

4a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.2. Read a list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to access to all the information regarding an existing list of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to read an existing list.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with further information about the selected list.

A.6.3. Update a list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to modify information of an existing list of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to modify an existing list.
2. The system provides the actor with a representation which contains all currently existing lists except and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with further information about the selected list.
5. The actor modifies some information about the selected list (e.g. the name).
6. The system asks the actor for confirmation.
7. The actor confirms his/her modification/s.
8. The system validates all the information regarding the list.
9. The system updates the existing list with the modified information.
10. The system notifies the actor that the list has been successfully updated.

Extensions:

2a. The actor does not have any list in the system apart from "Sent" and "Inbox":

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.4. Delete a list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete an existing list of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to delete an existing task.

2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects an existing list.
4. The system asks the actor for confirmation.
5. The actor confirms the deletion.
6. The system deletes the selected list.
7. The system notifies the actor that the list has been successfully deleted.

Extensions:

2a. The actor does not have any list in the system apart from "Sent" and "Inbox":

1. The system notifies the actor.
2. The use case ends.

4a. The selected list has some tasks assigned to it:

1. The system notifies the actor that the list still has some tasks assigned to it and hence it can not be deleted.
2. The use case ends.

4b. The selected list is locked for deletion:

1. The system notifies the actor that the list is locked for deletion.
2. The use case ends.

A.6.5. Set default list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to set an existing list of tasks as the default one.

Main success scenario:

1. The actor indicates the system that he/she wants to set his/her default list.
2. The system provides the actor with a representation which contains all currently existing lists.
3. The actor selects a list.
4. The system asks the actor for confirmation.

5. The actor confirms the choice.
6. The system sets the selected list as default.
7. The system notifies the actor that the list has been successfully set as default.

Extensions:

2a. The actor already has a default list in the system:

1. The system notifies the actor.
2. The use case ends.

A.6.6. Unset default list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unset the default list of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to unset his/her default list.
2. The system asks the actor for confirmation.
3. The actor confirms the choice.
4. The system sets the selected list as default.
5. The system notifies the actor that the list has been successfully set as default.

Extensions:

2a. The actor does not have any default list in the system:

1. The system notifies the actor.
2. The use case ends.

A.6.7. Share a list with some contacts

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to share an existing list of tasks with some of his/her contacts.

Main success scenario:

1. The actor indicates the system that he/she wants to share an existing list with some of his/her contacts.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her contacts previously added.
5. The actor chooses the contacts he/she wants to share the task with.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the list information regarding who is the list shared with (with acceptance pending status).
10. The system assigns the list with all its tasks to all the contacts that were chosen by the actor.
11. The system notifies the actor that the list of tasks has been successfully shared.

Extensions:

2a. The actor does not have any list in the system apart from "Sent" and "Inbox":

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.8. Share a list with some groups

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to share an existing list with some of his/her groups of contacts

Main success scenario:

1. The actor indicates the system that he/she wants to share an existing list with some of his/her groups of contacts.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her groups previously added.
5. The actor chooses the groups he/she wants to share the list with.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the list information regarding who is the list shared with (with acceptance pending status).
10. The system assigns the list with all its tasks to all the contacts pertaining to the groups that were chosen by the actor.
11. The system notifies the actor that the task has been successfully shared.

Extensions:

2a. The actor does not have any list in the system apart from “Sent” and “Inbox”:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The groups the actor chose do not contain any contact :

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.9. Publish a list for some contacts

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to publish an existing list of tasks with some of his/her contacts (i.e. wants to give some of his/her contacts reading permissions for one of his/her list of tasks).

Main success scenario:

1. The actor indicates the system that he/she wants to publish an existing list with some of his/her contacts.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her contacts previously added.
5. The actor chooses the contacts he/she wants to publish the list for.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the list information regarding who is the list published for.
10. The system updates the task information regarding who is the task published for, for all the tasks pertaining to the published list.
11. The system assigns the list with all its tasks to all the contacts that were chosen by the actor.

12. The system notifies the actor that the list of tasks has been successfully published for the chosen contact/s.

Extensions:

2a. The actor does not have any contact in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any list in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.10. Publish a list for some groups

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to publish an existing list of tasks for some of his/her groups of contacts (i.e. wants to give some of his/her groups reading permissions for one of his/her list of tasks).

Main success scenario:

1. The actor indicates the system that he/she wants to publish an existing list of tasks for some of his/her groups of contacts.
2. The system provides the actor with a representation which contains all currently existing lists of tasks and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her groups previously added.
5. The actor chooses the groups he/she wants to publish the list for.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.

8. The system validates the introduced data.
9. The system updates the list information regarding who is the list published for.
10. The system updates the task information regarding who is the task published for, for all the tasks pertaining to the published list.
11. The system assigns the list with read-only permissions with all its tasks to all the contacts pertaining to the groups that were chosen by the actor.
12. The system notifies the actor that the task has been successfully published for the chosen group/s.

Extensions:

2a. The actor does not have any group in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor does not have any list in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The groups the actor chose do not contain any contact :

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.11. Publish a list for anyone

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to publish an existing list of tasks for anyone, including people from outside his/her contacts (i.e. wants to give anyone reading permissions for one of his/her list of tasks).

Main success scenario:

1. The actor indicates the system that he/she wants to publish an existing list of tasks for anyone.
2. The system provides the actor with a representation which contains all currently existing lists of tasks and enough information to identify them.
3. The actor selects an existing list.
4. The system asks the actor for confirmation.
5. The actor confirms the selection.
6. The system validates the introduced data.
7. The system updates the list information making it public.
8. The system updates the task information regarding who is the task published for, for all the tasks pertaining to the published list.
9. The system assigns the list with read-only permissions with all its tasks to everyone who has not already higher permissions.
10. The system notifies the actor that the task has been successfully published for anyone.

Extensions:

2a. The actor does not have any list in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.12. Unpublish a list for some contacts

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unpublish an existing list of tasks with some of his/her contacts that had it published for them.

Main success scenario:

1. The actor indicates the system that he/she wants to unpublish an existing list with for some of his/her contacts.
2. The system provides the actor with a representation which contains all currently published lists for some contacts and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her contacts added to reading rights for the list.
5. The actor chooses the contacts he/she wants to unpublish the list for.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the list information regarding who is the list published for.
10. The system updates the task information regarding who is the task published for, for all the tasks pertaining to the published list.
11. The system unassigns the list with all its tasks to all the contacts that were chosen by the actor.
12. The system notifies the actor that the list of tasks has been successfully unpublished for the chosen contact/s.

Extensions:

2a. The actor does not have any list published for some contacts in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.13. Unpublish a list for some groups

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unpublish an existing list of tasks for some of his/her groups of contacts that had reading rights to it.

Main success scenario:

1. The actor indicates the system that he/she wants to unpublish an existing list of tasks for some of his/her groups of contacts.
2. The system provides the actor with a representation which contains all currently published lists for some groups and enough information to identify them.
3. The actor selects an existing list.
4. The system provides the actor with a graphical representation of his/her groups previously added to reading rights for the list.
5. The actor chooses the groups he/she wants to unpublish the list for.
6. The system asks the actor for confirmation.
7. The actor confirms the selections.
8. The system validates the introduced data.
9. The system updates the list information regarding who is the list published for.
10. The system updates the task information regarding who is the task published for, for all the tasks pertaining to the published list.
11. The system unassigns the list with read-only permissions with all its tasks to all the contacts pertaining to the groups that were chosen by the actor.
12. The system notifies the actor that the task has been successfully unpublished for the chosen group/s.

Extensions:

2a. The actor does not have any list published for some groups in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The groups the actor chose do not contain any contact :

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.14. Unpublish a list for anyone

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unpublish an existing list of tasks for anonymous access.

Main success scenario:

1. The actor indicates the system that he/she wants to unpublish an existing list of tasks for anonymous access.
2. The system provides the actor with a representation which contains all currently published lists for anonymous access and enough information to identify them.
3. The actor selects an existing list.
4. The system asks the actor for confirmation.
5. The actor confirms the selection.
6. The system validates the introduced data.
7. The system updates the list information making it private.
8. The system updates the task information regarding who the task is published for, for all the tasks pertaining to the unpublished list.
9. The system unassigns the list with read-only permissions with all its tasks to everyone who has not already higher permissions.
10. The system notifies the actor that the task has been successfully unpublished for anonymous access.

Extensions:

2a. The actor does not have any published list for anonymous access in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.15. Accept a shared list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: One of the actor's contacts wants to share a list of tasks with him/her.

Main success scenario:

1. The actor indicates the system that he/she wants to manage his/her lists.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The system indicates the actor that some of the lists are still pending for acceptance, and that those lists are originally from some of his/her contacts.
4. The system marks the lists pending for acceptance so they can be easily identified.
5. The actor selects a list of tasks which is still pending for acceptance.
6. The system provides the actor with further information about the selected list (e.g. who is the contact that shared it with him/her).
7. The system provides the actor with an interface to let him choose whether he/she accepts the list or rejects it.
8. The user accepts the shared list.
9. The system validates the data.
10. The system updates the list information regarding who is the list shared with (it removes the acceptance pending status).
11. The system updates the task information regarding who is the task shared with, adding the username of the actor, for all the tasks pertaining to the shared list.
12. The system notifies the actor that the list of tasks has been successfully shared, accepted and are now accessible to him/her.

Extensions:

9a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.16. Reject a shared list

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: One of the actor's contacts wants to share a list of tasks with him/her.

Main success scenario:

1. The actor indicates the system that he/she wants to manage his/her lists.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The system indicates the actor that some of the lists are still pending for acceptance, and that those lists are originally from some of his/her contacts.
4. The system marks the lists pending for acceptance so they can be easily identified.
5. The actor selects a list of tasks which is still pending for acceptance.
6. The system provides the actor with further information about the selected list (e.g. who is the contact that shared it with him/her).
7. The system provides the actor with an interface to let him choose whether he/she accepts the list or rejects it.
8. The user rejects the shared list.
9. The system validates the data.
10. The system updates the list information regarding who is the list shared with (it removes the acceptance pending status, along with the actors' name).
11. The system removes the list of tasks from the actor's account.
12. The system notifies the actor that the list of tasks has been successfully rejected

Extensions:

9a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.17. Archive lists

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to archive some of his/her lists of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to archive some of his/her lists.
2. The system provides the actor with a representation which contains all currently existing lists and enough information to identify them.
3. The actor selects the lists of tasks he/she wishes to archive.
4. The system asks the actor for confirmation.
5. The actor confirms the selections.
6. The system validates the introduced data.
7. The system marks the selected lists of tasks as archived and archives them.
8. The system notifies the actor that the list of tasks have been successfully archived.

Extensions:

2a. The actor does not have any unarchived task in the system:

1. The system notifies the actor.
2. The use case ends.

4a. The actor has chosen at least one list of tasks that is already archived:

1. The system notifies the actor.
2. The use case ends.

4b. The actor does not have any unarchived task in the system:

1. The system notifies the actor that the list of tasks is pending acceptance, and hence can not be archived.
2. The use case ends.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.6.18. Unarchive lists

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unarchive some of his/her archived lists of tasks.

Main success scenario:

1. The actor indicates the system that he/she wants to unarchive some of his/her already archived lists.
2. The system provides the actor with a representation which contains all currently archived lists and enough information to identify them.
3. The actor selects the archived lists of tasks he/she wishes to unarchive.
4. The system asks the actor for confirmation.
5. The actor confirms the selections.
6. The system validates the introduced data.
7. The system marks the selected lists of tasks as unarchived and unarchives them.
8. The system notifies the actor that the list of tasks have been successfully unarchived.

Extensions:

2a. The actor does not have any archived task in the system:

1. The system notifies the actor.
2. The use case ends.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.7. Location

A.7.1. Create a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to create a new location.

Main success scenario:

1. The actor indicates the system that he/she wants to create a new location.
2. The system provides the actor with a graphical representation (e.g. a map or a list with names and coordinates) which contains all currently existing locations.
3. The actor introduces a position.
4. The system asks the actor for further information regarding the location (e.g. custom location name).
5. The actor introduces further information.
6. The system validates all the information the actor has introduced.
7. The system creates a new location with the introduced information.
8. The system notifies the actor that the location has been successfully created.

Extensions:

3a. The actor knows the direction of the location:

1. The actor introduces the direction of the location (i.e. street number, street name, town name, etc.).
2. The system shows the found locations which matches with the information provided to the actor.
    a) The system only found one location that matched the specified criteria:
        i. The execution continues from the step 3a.4 of this extension.
    b) The system did not find any location that matched the specified criteria:
        i. The use case ends.
3. The actor chooses one of the locations supplied by the system.
4. The execution returns to the 4th step of the main success scenario.

6a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.7.2. Read a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to read all the information regarding an existing location.

Main success scenario:

1. The actor indicates the system that he/she wants to read an existing location.
2. The system provides the actor with a representation (e.g. a map or a list with names and coordinates) which contains all currently existing locations.
3. The actor selects a location.
4. The system shows further information about the location.

Extensions:

2a. The actor does not have any location in the system:

1. The system notifies the actor.
2. The use case ends.

A.7.3. Update a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to modify an existing location.

Main success scenario:

1. The actor indicates the system that he/she wants to modify an existing location.
2. The system provides the actor with a representation (e.g. a map or a list with names and coordinates) which contains all currently existing locations.
3. The actor selects a location.

4. The system shows further information about the location.
5. The actor modifies some information about the location.
6. The system asks the actor for confirmation.
7. The actor confirms his/her modification/s.
8. The system validates all the information regarding the location.
9. The system updates the existing location with the modified information.
10. The system notifies the actor that the location has been successfully modified.

Extensions:

2a. The actor does not have any location in the system:

1. The system notifies the actor.
2. The use case ends.

8a. The introduced data fails the validation tests:

1. The system shows the validation errors encountered during the previous step.
2. The use case ends.

A.7.4. Delete a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to delete an existing location.

Main success scenario:

1. The actor indicates the system that he/she wants to delete an existing location.
2. The system provides the actor with a representation (e.g. a map or a list with names and coordinates) which contains all currently existing locations.
3. The actor selects a location.
4. The system asks the actor for confirmation.
5. The actor confirms the deletion.
6. The system deletes the selected location.
7. The system notifies the actor that the location has been successfully deleted.

Extensions:

2a. The actor does not have any location in the system:

1. The system notifies the actor.
2. The use case ends.

A.7.5. Set a default a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to set an existing location as the default one.

Main success scenario:

1. The actor indicates the system that he/she wants to set his/her default location.
2. The system provides the actor with a representation (e.g. a map or a list with names and coordinates) which contains all currently existing locations.
3. The actor selects a location.
4. The system asks the actor for confirmation.
5. The actor confirms the choice.
6. The system sets the selected location as default.
7. The system notifies the actor that the location has been successfully set as default.

Extensions:

2a. The actor does not have any location in the system:

1. The system notifies the actor.
2. The use case ends.

2b. The actor already has a default location in the system:

1. The system notifies the actor.
2. The use case ends.

A.7.6. Unset a default a location

Scope: RTM task management system.

Primary actor: End-user.

Preconditions: The user must be logged into the system (for further information see sec. A.2.2 Log into an account)

Includes: none.

Trigger: The actor wants to unset the default location.

Main success scenario:

1. The actor indicates the system that he/she wants to unset his/her default location.
2. The system asks the actor for confirmation.
3. he actor confirms the choice.
4. The system unsets the default location.
5. The system notifies the actor that the location has been successfully unset as default.

Extensions:

2a. The actor does not have any default location in the system:

1. The system notifies the actor.
2. The use case ends.

B. Test stories

B.1. Task

B.1.1. Create a task

B.1.1.1. Test story 1: Creation of a task

Test objective: Create a task (Main success scenario).

Alice wants to add a task to the system.

She logs into the system successfully. The system provides Alice with a list of her currently pending tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice asks the system to create a new task with the subject "Buy rice for lunch" with due date Saturday morning.

The system confirms the creation of the new task, and the list of tasks now includes it.

B.1.1.2. Test story 2: Failed creation (duplicated name)

Test objective: Attempt to create a task with a duplicated name (Extension 4a).

Alice wants to add a task to the system.

She logs into the system successfully. The system provides Alice with a list of her currently pending tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Buy rice for lunch (Saturday morning)".

Alice asks the system to create a new task with the subject "Buy rice for lunch" with due date Saturday morning.

The system rejects the creation of the new task and informs Alice that she already has that same task scheduled (i.e. a task with the same name and date).

B.1.1.3. Test story 3: Failed creation (blank name)

Test objective: Attempt to create a task with a blank name (Extension 4a).

Alice wants to add a task to the system.

She logs into the system successfully. The system provides Alice with a list of her currently pending tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Buy rice for lunch (Saturday morning)".

Alice asks the system to create a new task without any subject or due date.

The system rejects the creation of the new task and informs Alice that every task must at least have a subject.

B.1.1.4. Test story 4: Failed creation (name too long)

Test objective: Attempt to create a task with a too long name (Extension 4a).

Alice wants to add a task to the system.

She logs into the system successfully. The system provides Alice with a list of her currently pending tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice asks the system to create a new task with a 2000 word rice recipe as the subject with due date Saturday at 12 p.m.

The system rejects the creation of the new task and informs Alice that the task subject is too long.

B.1.2. Read a task

B.1.2.1. Test story 1: Reading of a task

Test objective: Read a task (Main success scenario).

Alice wants to read a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

B.1.2.2. Test story 2: Failed Read (no tasks)

Test objective: Attempt to read a task when the system has none (Extension 2a).

Alice wants to read a task from the system.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.3. Update a task

B.1.3.1. Test story 1: Modification of a task

Test objective: Update a task (Main success scenario).

Alice wants to update a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to update some of its information.

The system provides Alice with further information about the task, like a map with the exact location of the cinema, the task's priority and Bob's contact details.

Alice changes the subject description to "Go to the cinema with Bob and Carol (Saturday evening)".

The system asks Alice to confirm the modification. She confirms the modification. The system informs Alice that the modification of the task has been successful, and the list of tasks now includes the modified version of the subject description.

B.1.3.2. Test story 2: Failed update (no tasks)

Test objective: Attempt to update a task when the system has none (Extension 2a).

Alice wants to update a task from the system.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.3.3. Test story 3: Failed update (invalid data)

Test objective: Attempt to update a task in such a manner that would leave the system in an invalid state (Extension 8a).

Alice wants to update a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to update some of its information.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice changes the subject description to a blank space. The system asks Alice to confirm the modification. She confirms the modification.

The system informs Alice that the modification of the task is not possible because it would lead to an error.

B.1.4. Delete a task

B.1.4.1. Test story 1: Deletion of a task

Test objective: Delete a task (Main success scenario).

Alice wants to delete a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to delete it.

The system asks Alice to confirm the deletion. She confirms the deletion.

The system informs Alice that the deletion of the task has been successful, and the list of tasks now excludes the chosen task.

B.1.4.2. Test story 2: Failed deletion (no tasks)

Test objective: Attempt to delete a task when the system has none (Extension 2a).

Alice wants to delete a task from the system.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.5. Create an assign a note to task

B.1.5.1. Test story 1: Creation and assignation of a note to a task

Test objective: Create and assign a note to a task (Main success scenario).

Alice wants to create and assign a note to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to create and assign a note to the task. The system provides Alice with a list of all her notes assigned to the task "Go to the cinema with Bob (Saturday evening)". The first one states "Bring enough money to be able to invite him (Bob paid the last time)" and the other one contains some information about the film and reviews.

Alice adds a new note to the task stating "Remember to bring 3D glasses to the cinema". Alice informs the system that she wants to save the note.

The system informs Alice that the note has been created and assigned successfully to "Go to the cinema with Bob (Saturday evening)". Now the list of notes assigned to the task includes "Remember to bring 3D glasses to the cinema".

B.1.5.2. Test story 2: Failed creation of a note (no content)

Test objective: Attempt to create a note with no content (Extension 6a).

Alice wants to create and assign a note to a task from the system.

She logs into the system successfully. The system provides Alice with a list of her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to create and assign a note to the task. The system provides Alice with a list of all her notes assigned to the task "Go to the

cinema with Bob (Saturday evening)". The first one states "Bring enough money to be able to invite him (Bob paid the last time)" and the other one contains some information about the film and reviews.

Alice asks the system to create a new task note without any kind of content. Alice informs the system that she wants to save the note.

The system rejects the creation of the new note and informs Alice that every note must at least have some content.

B.1.6. Read a note assigned to a task

B.1.6.1. Test story 1: Reading of a note assigned to a task

Test objective: Reading of a note assigned to a task (Main success scenario).

Alice wants to read a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to read a note assigned to the task. The system provides Alice with a list of all her notes assigned to the task "Go to the cinema with Bob (Saturday evening)". The first one states "Bring enough money to be able to invite him (Bob paid the last time)" and the other one contains some information about the film and reviews.

Alice chooses the note "Bring enough money to be able to invite him (Bob paid the last time)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the note she had previously entered, like the exact amount of money for each cinema ticket and the cost of a medium sized popcorn bag.

B.1.6.2. Test story 2: Failed read (no notes)

Test objective: Attempt to read a note from a task when this has none (Extension 2a).

Alice wants to read a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are “Go to the cinema with Bob (Saturday evening)”, “Go to the post office to send a package to Carol (Monday morning)” and “Go to the theatre with Carol (yesterday night)”.

Alice chooses the task “Go to the cinema with Bob (Saturday evening)” and informs the system that she wants to read more information about its notes. The system informs Alice that there are no notes assigned to this task.

B.1.7. Update a note assigned to a task

B.1.7.1. Test story 1: Modification of a note assigned to a task

Test objective: Update a note assigned to a task (Main success scenario).

Alice wants to update a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are “Go to the cinema with Bob (Saturday evening)”, “Go to the post office to send a package to Carol (Monday morning)” and “Go to the theatre with Carol (yesterday night)”.

Alice chooses the task “Go to the cinema with Bob (Saturday evening)” and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob’s contact details.

Alice informs the system that she wants to update a note assigned to the task. The system provides Alice with a list of all her notes assigned to the task “Go to the cinema with Bob (Saturday evening)”. The first one states “Bring enough money to be able to invite him (Bob paid the last time)” and the other one contains some information about the film and reviews.

Alice chooses the note “Bring enough money to be able to invite him (Bob paid the last time)” and informs the system that she wants to modify some information about it. The system provides Alice with further information about the note she had previously entered, like the exact amount of money for each cinema ticket and the cost of a medium sized popcorn bag.

Alice modifies some of the information related to the note, such as adding the cinema price for a medium sized glass of a coke.

The system asks Alice to confirm the modification. She confirms the modification. The system informs Alice that the modification of the note has been successful, and the note is now modified with the information added.

B.1.7.2. Test story 2: Failed update of a note (no notes)

Test objective: Attempt to update a note from a task when this has none (Extension 2a).

Alice wants to update a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to update some of the information about its notes. The system informs Alice that there are no notes assigned to this task.

B.1.7.3. Test story 3: Failed update of a note (no notes)

Test objective: Attempt to update a note from a task when this has none (Extension 2a).

Alice wants to update a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to update a note assigned to the task. The system provides Alice with a list of all her notes assigned to the task "Go to the cinema with Bob (Saturday evening)". The first one states "Bring enough money to be able to invite him (Bob paid the last time)" and the other one contains some information about the film and reviews.

Alice chooses the note "Bring enough money to be able to invite him (Bob paid the last time)" and informs the system that she wants to modify some information about it. The system provides Alice with further information about the note she had previously entered, like the exact amount of money for each cinema ticket and the cost of a medium sized popcorn bag.

Alice modifies some of the information related to the note, such as adding the cinema price for a medium sized glass of a coke.

The system asks Alice to confirm the modification. She confirms the modification.

The system informs Alice that the modification could not be performed because the note had been deleted after being shown to her and before she confirmed the changes.

B.1.8. Delete a note assigned to a task

B.1.8.1. Test story 1: Deletion of a note assigned to a task

Test objective: Deletion of a note assigned to a task (Main success scenario).

Alice wants to delete a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to delete a note assigned to the task. The system provides Alice with a list of all her notes assigned to the task "Go to the cinema with Bob (Saturday evening)". The first one states "Bring enough money to be able to invite him (Bob paid the last time)" and the other one contains some information about the film and reviews.

Alice chooses the note "Bring enough money to be able to invite him (Bob paid the last time)". The system asks Alice to confirm the deletion. She confirms the deletion. The system informs Alice that the deletion of the note has been successful, and the note now does not appear in the task's note list.

B.1.8.2. Test story 2: Failed note deletion (no notes)

Test objective: Attempt to delete a note from a task when this has none (Extension 2a).

Alice wants to delete a note assigned to a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to delete some of its notes. The system informs Alice that there are no notes assigned to this task.

B.1.9. Complete a task

B.1.9.1. Test story 1: Completion of a task

Test objective: Completion of a task (Main success scenario).

Alice wants to complete a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to complete it.

The system informs Alice that the completion of the task has been successful, and the task now appears as completed.

B.1.9.2. Test story 2: Failed completion (no unfinished tasks)

Test objective: Attempt to complete an unfinished task when the system has none (Extension 2a).

Alice wants to complete a task from the system.

She logs into the system successfully. The system informs Alice that there are no unfinished tasks.

B.1.10. Uncomplete a task

B.1.10.1. Test story 1: Incompletion of a task (Main success scenario).

Test objective: Incompletion of a task (Main success scenario).

Alice wants to uncomplete a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her currently finished tasks which is only one, "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the theatre with Carol (yesterday night)" and informs the system that she wants to uncomplete it.

The system informs Alice that the incompletion of the task has been successful, and the task now appears as not completed.

B.1.10.2. Test story 2: Failed incompletion (no finished tasks)

Test objective: Attempt to uncomplete a finished task when the system has none (Extension 2a).

Alice wants to uncomplete a task from the system.

She logs into the system successfully. The system informs Alice that there are no finished tasks.

B.1.11. Set priority to a task

B.1.11.1. Test story 1: Assignation of a priority to a task

Test objective: Set a priority to a task (Main success scenario).

Alice wants to set the priority for a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to set a priority to the task. The system provides her with an interface she can fill the desired priority in, along with a caption which instructs the allowed range values.

Alice chooses the highest priority for the task. The system informs Alice that the priority has been set successfully to "Go to the cinema with Bob (Saturday evening)". Now the task has the highest priority.

B.1.11.2. Test story 2: Failed assignation of a priority (the task already has a priority)

Test objective: Attempt to set a priority to a task that already has one (Extension 2a).

Alice wants to set the priority for a task from the system.

She logs into the system successfully. The system provides Alice with a list of her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go

to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to set a priority to the task. The system rejects the action and informs Alice that this task already has a priority.

B.1.11.3. Test story 3: Failed assignation of a priority (invalid priority)

Test objective: Attempt to set an out of range priority to a task (Extension 5a).

Alice wants to set the priority for a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to set a priority to the task. The system provides the user with an interface she can fill the desired priority in, along with a caption which instructs the allowed range values.

Alice inputs a non-valid priority for the task (e.g. '¿'). The system rejects the action and informs Alice that this priority is out of range.

B.1.12. Update priority from a task

B.1.12.1. Test story 1: Modification of a priority from a task

Test objective: Update a priority from a task (Main success scenario sec. 4.5.1.1).

Alice wants to update the priority from a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to update the priority of the task. The system provides her with an interface with the current priority value that she can modify, along with a caption which instructs the allowed range values. In this case, the priority is the highest possible, but in case it did not have any assigned one it would simply be blank.

Alice chooses this time the second highest priority for the task. The system informs Alice that the priority has been modified successfully for the task "Go to the cinema with Bob (Saturday evening)". Now the task has the second highest priority.

B.1.12.2. Test story 2: Increasing the priority of a task

Test objective: Increment the current priority of a task (Extension 4a sec. 4.5.1.1).

Alice wants to increment the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the post office to send a package to Carol (Monday morning)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the post office and Carol's contact details.

Alice informs the system that she wants to increment the priority from the task. The system informs Alice that the priority has been incremented successfully for the task "Go to the post office to send a package to Carol (Monday morning)" and now the task has one level above standard priority, because it did not have any priority assigned before.

B.1.12.3. Test story 3: Decreasing the priority of a task

Test objective: Decrement the current priority of a task (Extension 4b sec. 4.5.1.1).

Alice wants to decrement the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go

to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to decrement the priority from the task. The system informs Alice that the priority has been decremented successfully for the task "Go to the cinema with Bob (Saturday evening)". Now the task has the second highest priority.

B.1.12.4. Test story 4: Failed update of a priority (priority out of bounds)

Test objective: Attempt to increment a priority from a task which was already set as the highest priority (Extension 6a + Extension 4asec. 4.5.1.1).

Alice wants to increment the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob's contact details.

Alice informs the system that she wants to increment the priority from the task. The system informs Alice that the priority can not be incremented because the task "Go to the cinema with Bob (Saturday evening)" already had the highest possible priority.

B.1.13. Delete priority from a task

B.1.13.1. Test story 1: Deletion of a priority from a task

Test objective: Delete a priority from a task (Main success scenario).

Alice wants to delete the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task “Go to the cinema with Bob (Saturday evening) and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the cinema and Bob’s contact details.

Alice informs the system that she wants to delete the priority from the task. The system asks Alice to confirm the deletion. She confirms the deletion.

The system informs Alice that the deletion of the task’s priority has been successful, and now the task does not have any priority.

B.1.13.2. Test story 2: Failed priority deletion (the task has no priority)

Test objective: Attempt to delete the current priority of a task (Extension 2a).

Alice wants to delete the priority of a task from the system.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are “Go to the cinema with Bob (Saturday evening)”, “Go to the post office to send a package to Carol (Monday morning)” and “Go to the theater with Carol (yesterday night)”.

Alice chooses the task “Go to the post office to send a package to Carol (Monday morning)” and informs the system that she wants to read more information about it.

The system provides Alice with further information about the task, like a map with the exact location of the post office and Carol’s contact details.

Alice informs the system that she wants to delete the priority of the task. The system informs Alice that the task “Go to the post office to send a package to Carol (Monday morning)” does not have any priority, and therefore, its priority can not be deleted.

B.1.14. Postpone a task

B.1.14.1. Test story 1: Postponement of a task.

Test objective: Postponement of a task whose date is due (Main success scenario).

Alice wants to postpone an unfinished task from the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are “Go to the cinema with Bob (Saturday evening)”, “Go to the post office to send a package to Carol (Monday morning)” and “Go to the dentist (last week)”.

Alice chooses the task "Go to the dentist (last week)" and informs the system that she wants to postpone it.

The system informs Alice that the postponement of the task has been successful, and the task now appears due on that day and as it has been postponed once more.

B.1.14.2. Test story 2: Failed postponement a task (no finished tasks)

Test objective: Attempt to postpone an unfinished task when the system has none (Extension 2a).

Alice wants to postpone an unfinished task from the system.

She logs into the system successfully. The system informs Alice that there are no finished tasks.

B.1.14.3. Test story 3: Postponement of a not due task

Test objective: Postponement of a task which is not yet due (Extension 4a).

Alice wants to postpone an unfinished task from the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the post office to send a package to Carol (Monday morning)" and informs the system that she wants to postpone it.

The system informs Alice that the postponement of the task has been successful, and the task now appears due on Tuesday morning and as it has been postponed once more.

B.1.15. Share a task with contacts

B.1.15.1. Test story 1: Share a task with contacts

Test objective: Sharing of an unfinished task with some of the user's contacts (Main success scenario).

Alice wants to share an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to share it with some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to share the task with him. The system asks Alice to confirm the sharing.

She confirms the sharing. The system informs Alice that the sharing of the chosen task with the chosen contacts has been successful. Now the task is marked as shared with those contacts and the system assigns the task to all of them, in this case only to Bob.

B.1.15.2. Test story 2: Failed sharing of a task (no unfinished tasks)

Test objective: Attempt to share an unfinished task when the system has none (Extension 2a).

Alice wants to share an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unfinished tasks.

B.1.15.3. Test story 3: Failed sharing of a task (no contacts)

Test objective: Attempt to share an unfinished task when she does not have any contact in the system (Extension 2b).

Alice wants to share an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no contacts to share any task with.

B.1.15.4. Test story 4: Failed sharing of a task (some chosen contacts already have the task)

Test objective: Attempt at sharing an unfinished task with some of the user's contacts, when some of them already have the task shared previously (Extension 8a).

Alice wants to share an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to share it with some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to share the task with him. The system asks Alice to confirm the sharing. She confirms the sharing. The system rejects the sharing and informs Alice that the chosen task was already shared with some of the chosen contacts.

B.1.16. Send a task to contacts

B.1.16.1. Test story 1: Send a task to some contacts

Test objective: Sending an unfinished task to some of the user's contacts (Main success scenario).

Alice wants to send an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the post office to send a package to Carol (Monday morning)" and informs the system that she wants to send it to some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to send the task to him. The system asks Alice to confirm the sending.

She confirms the sending. The system informs Alice that the sending of the chosen task to the chosen contacts has been successful. Now the task's sender is marked with the "Alice" account username. The task has been moved to Alice's "Sent" list and the system has assigned to task to all the contacts that where chosen by Alice, in this case to "Bob".

B.1.16.2. Test story 2: Failed sending of a task (no unfinished tasks)

Test objective: Attempt to send an unfinished task when the system has none (Extension 2a).

Alice wants to send an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unfinished tasks.

B.1.16.3. Test story 3: Failed sending of a task (no contacts)

Test objective: Attempt to send an unfinished task when she does not have any contact in the system (Extension 2b).

Alice wants to send an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no contacts to send any task to.

B.1.16.4. Test story 4: Failed sending of a task (no chosen contact)

Test objective: Attempt at sending an unfinished task without specifying any of the user's contacts (Extension 8a).

Alice wants to send an unfinished task from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to send it to some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice does not choose anyone but asks the system to proceed. The system rejects the sending and informs Alice that she must at least choose one of her contacts.

B.1.17. Share a task with groups

B.1.17.1. Test story 1: Share a task with groups

Test objective: Sharing of an unfinished task with some of the user's groups of contacts (Main success scenario).

Alice wants to share an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with my friends (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with my friends (Saturday evening)" and informs the system that she wants to share it with some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are "Friends" and "Coworkers".

Alice chooses the group of contacts "Friends" and informs the system that she wants to share the task with them. The system asks Alice to confirm the sharing. She confirms the sharing.

The system informs Alice that the sharing of the chosen task with the chosen groups of contacts has been successful. Now the task is marked as shared with those contacts in the groups and the system assigns the task to all of them, in this case Bob and Carol.

B.1.17.2. Test story 2: Failed sharing of a task (no unfinished tasks)

Test objective: Attempt to share an unfinished task when the system has none (Extension 2a).

Alice wants to share an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unfinished tasks.

B.1.17.3. Test story 3: Failed sharing of a task (no groups of contacts)

Test objective: Attempt to share an unfinished task when she does not have any groups of contacts in the system (Extension 2b).

Alice wants to share an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no groups of contacts to share any task with.

B.1.17.4. Test story 4: Failed sharing of a task (empty group)

Test objective: Attempt to share an unfinished task with some groups of contacts that are empty (Extension 6a).

Alice wants to share an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are “Go to the cinema with my friends (Saturday evening)” and “Go to the post office to send a package to Carol (Monday morning)”.

Alice chooses the task “Go to the cinema with my friends (Saturday evening)” and informs the system that she wants to share it with some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends”, “Coworkers” and “Unknown”.

Alice chooses the group of contacts “Unknown” and informs the system that she wants to share the task with them. The system asks Alice to confirm the sharing.

She confirms the sharing. The system rejects the sharing and informs Alice that some of the chosen groups of contacts did not contain any contact in it, in this case the group “Unknown”.

B.1.17.5. Test story 5: Failed sharing of a task (already shared)

Test objective: Attempt at sharing an unfinished task with some of the user’s contacts belonging to groups, when some of them already have the task shared previously (Extension 8a).

Alice wants to share an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are “Go to the cinema with my friends (Saturday evening)” and “Go to the post office to send a package to Carol (Monday morning)”.

Alice chooses the task “Go to the cinema with my friends (Saturday evening)” and informs the system that she wants to share it with some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends” and “Coworkers”.

Alice chooses the group of contacts “Friends” and informs the system that she wants to share the task with them. The system asks Alice to confirm the sharing.

She confirms the sharing. The system rejects the sharing and informs Alice that the chosen task was already shared with some of the contacts belonging to the chosen groups, in this case “Bob”.

B.1.18. Send a task to groups

B.1.18.1. Test story 1: Send a task to some groups

Test objective: Sending an unfinished task to some of the user's groups of contacts (Main success scenario).

Alice wants to send an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send some packages to Carol (Monday morning)".

Alice chooses the task "Go to the post office to send some packages to Carol (Monday morning)" and informs the system that she wants to send it to some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These contacts are "Friends" and "Coworkers".

Alice chooses the group of contacts "Coworkers" and informs the system that she wants to send the task to them. The system asks Alice to confirm the sending.

She confirms the sending. The system informs Alice that the sending of the chosen task to the contacts belonging to the chosen groups has been successful.

Now the task's sender is marked with the "Alice" account username. The task has been moved to Alice's "Sent" list and the system has assigned the task to all the contacts that where belonging to the groups chosen by her, in this case to "Bob" and "Charlie".

B.1.18.2. Test story 2: Failed sending of a task (no unfinished tasks)

Test objective: Attempt to send an unfinished task when the system has none (Extension 2a).

Alice wants to send an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unfinished tasks.

B.1.18.3. Test story 3: Failed sending of a task (no groups)

Test objective: Attempt to send an unfinished task when she does not have any groups of contacts in the system (Extension 2b).

Alice wants to send an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no groups of contacts to send any task to.

B.1.18.4. Test story 4: Failed sending of a task (empty groups)

Test objective: Attempt to send an unfinished task to some groups of contacts that are empty (Extension 6a).

Alice wants to send an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with my friends (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the post office to send some packages to Carol (Monday morning)" and informs the system that she wants to send it to some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are "Friends", "Coworkers" and "Unknown".

Alice chooses the group of contacts "Unknown" and informs the system that she wants to send the task to them. The system asks Alice to confirm the sending.

She confirms the sending. The system rejects the sending and informs Alice that some of the chosen groups of contacts did not contain any contact in it, in this case the group "Unknown".

B.1.18.5. Test story 5: Failed sending of a task (no chosen)

Test objective: Attempt at sending an unfinished task without specifying any of the user's groups of contacts (Extension 8a).

Alice wants to send an unfinished task from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to send it to some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These contacts are "Friends" and "Coworkers".

Alice does not choose any group but asks the system to proceed. The system rejects the sending and informs Alice that she must at least choose one of her groups of contacts.

B.1.19. Show tasks

B.1.19.1. Test story 1: Show tasks

Test objective: Show the user his/her tasks in the system that meet some specified criteria in an orderly fashion (Main success scenario).

Alice wants to view all her tasks that meet some criteria in an orderly way.

She logs into the system successfully. The system provides Alice with a list with all the possible parameters related to ordering criteria. These include creation date, priority and due date.

Alice informs the system that she wants the tasks ordered regarding their priority, the ones with higher priority in first place.

The system provides Alice with a list with all the possible parameters related to filtering tasks. These include keywords, the list the tasks belong to and who are they shared with.

Alice informs the system that she wants to retrieve all the tasks due in less than one week that are uncompleted and involve her contact “Bob”.

The system provides Alice with all her tasks according to the ordering and filtering criteria, which in this case would only be “Go to the cinema with Bob (Saturday evening)”.

B.1.19.2. Test story 2: Failed showing of tasks (no tasks)

Test objective: Attempt to show the user his/her tasks in the system that meet some specified criteria in an orderly fashion (Extension 2a).

Alice wants to view all her tasks that meet some criteria in an orderly way.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.19.3. Test story 3: Show tasks with no ordering criterion

Test objective: Show the user his/her tasks in the system that meet some criteria without any specific order (Extension 3a).

Alice wants to view all her tasks that meet some criteria in an orderly way.

She logs into the system successfully. The system provides Alice with a list with all the possible parameters related to ordering criteria. These include creation date, priority and due date.

Alice informs the system that she wants to proceed without choosing any specific criterion. The system provides Alice with a list with all the possible parameters related to filtering tasks. These include keywords, the list the tasks belong to and who are they shared with.

Alice informs the system that she wants to retrieve all the tasks due in less than one week that are uncompleted and involve her contact "Bob".

The system provides Alice with all her tasks according to the chosen filtering criteria. As a ordering criterion it will use its default one (in this case with increasing due date and using the priority, alphabetical order and creation date to break ties), because Alice did not choose anything specific. The only task that meets all this is "Go to the cinema with Bob (Saturday evening)".

B.1.19.4. Test story 5: Show tasks (too restrictive filtering)

Test objective: Attempt to show the user his/her tasks in the system that meet some specified criteria in an orderly fashion (Extension 6a).

Alice wants to view all her tasks that meet some criteria in an orderly way.

She logs into the system successfully. The system provides Alice with a list with all the possible parameters related to ordering criteria. These include creation date, priority and due date.

Alice informs the system that she wants the tasks ordered regarding their priority, the ones with higher priority in first place.

The system provides Alice with a list with all the possible parameters related to filtering tasks. These include keywords, the list the tasks belong to and who are they shared with. Alice informs the system that she wants to retrieve all the tasks due in less than one week that are uncompleted and do not involve her contacts "Bob" or "Carol".

The system informs Alice that she does not have any task in the system that meet those selected filtering criteria.

B.1.20. Move a task to a list

B.1.20.1. Test story 1: Moving a task to a list

Test objective: Moving a task from its original list to another one (Main success scenario).

Alice wants to move one of her tasks from one of her lists to another one.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to move it to one of her other lists.

The system provides Alice with a list of all her lists of tasks except from the one she is already on and the list of tasks sent to contacts. These lists are "Personal" and "Work".

Alice chooses the list of tasks "Personal" and informs the system that she wants to move the task there. The system asks Alice to confirm the movement. She confirms the movement.

The system informs Alice that the movement of the chosen task to the selected list has been successful. Now the task appears on the "Personal" list and does not appear in the list named "Inbox" anymore.

B.1.20.2. Test story 2: Failed moving of a task to a list (no tasks)

Test objective: Attempt to move a task to another list when the system has none (Extension 2a).

Alice wants to move one of her tasks from one of her lists to another one.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.20.3. Test story 3: Failed moving of a task to a list (restricted list)

Test objective: Attempt to move a task from the "Sent" list to another one (Extension 4a).

Alice wants to move one of her tasks from one of her lists to another one.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to move it to one of her other lists.

The system informs Alice that the movement of the chosen task is not possible because it belongs to the "Sent" list which is restricted in task moving.

B.1.20.4. Test story 4: Failed moving of a task to a list (no unrestricted lists)

Test objective: Attempt to move a task from one list to another one, when the only other list available is the "Sent" one (Extension 4b).

Alice wants to move one of her tasks from one of her lists to another one.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the post office to send a package to Carol (Monday morning)" and informs the system that she wants to move it to one of her other lists.

The system informs Alice that the movement of the chosen task is not possible, because apart from the list the task is currently in (i.e. "Inbox") there is only the "Sent" list available, which is restricted in task moving.

B.1.20.5. Test story 5: Failed moving of a task to a list (no list chosen)

Test objective: Attempt to move a task from one list to another one without specifying which one (Extension 8a).

Alice wants to move one of her tasks from one of her lists to another one.

She logs into the system successfully. The system provides Alice with a list of all her currently unfinished tasks which are "Go to the cinema with my friends (Saturday evening)" and "Go to the post office to send a package to Carol (Monday morning)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to move it to one of her other lists.

The system provides Alice with a list of all her lists of tasks except from the one she is already on and the list of tasks sent to contacts. These lists are "Personal" and "Work". Alice does not choose any list of tasks but still asks the system to proceed. The system informs Alice that the movement of the chosen task is not possible because no valid list was chosen.

B.1.21. Duplicate a task

B.1.21.1. Test story 1: Duplicate task

Test objective: Duplication of a task (Main success scenario).

Alice wants to duplicate one of her tasks.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go

to the post office to send a package to Carol (Monday morning)" and "Go to the theater with Carol (yesterday night)".

Alice chooses the task "Go to the cinema with Bob (Saturday evening)" and informs the system that she wants to duplicate it.

The system informs Alice that the duplication of the chosen task has been done successfully. Now, another task appears in the list with all the others. This new task has the same properties (e.g. subject name, due date, shared contacts, etc.) as the original one it was duplicated from except from (possibly) the creation date and the text " copy" appended to the task subject. This would render the new task "Go to the cinema with Bob (Saturday evening) copy".

B.1.21.2. Test story 2: Failed duplication of a task (no tasks)

Test objective: Attempt to duplicate a task when the system has none (Extension 2a).

Alice wants to duplicate one of her tasks.

She logs into the system successfully. The system informs Alice that there are no tasks.

B.1.21.3. Test story 3: Duplicate a completed task

Test objective: Duplication of a completed task (Extension 4a).

Alice wants to duplicate one of her tasks.

She logs into the system successfully. The system provides Alice with a list of all her existing tasks which are "Go to the cinema with Bob (Saturday evening)", "Go to the post office to send a package to Carol (Monday morning)" and "Go to the theatre with Carol (yesterday night)".

Alice chooses the task "Go to the theatre with Carol (yesterday night)" which happens to be completed and informs the system that she wants to duplicate it.

The system informs Alice that the duplication of the chosen task has been done successfully. Now, another task appears in the list with all the others. This new task has the same properties (e.g. subject name, due date, shared contacts, etc.) as the original one it was duplicated from except from the creation date, the due date and its completed status along with the completion date. This would render the new task "Go to the theatre with Carol" with an incomplete status and with no due date or completion date.

B.2. Account

B.2.1. Create an account

B.2.1.1. Test story 1: Creation of an account

Test objective: Create an account (Main success scenario).

Alice wants to create an account in the system.

The system provides Alice with an interface with enough fields for her to create an account. Alice introduces her desired username “Alice”. She also introduces her desired password “1234”.

The system confirms the creation of the account with the introduced username “Alice” and password. The system informs Alice that her account has been successfully created and it is now ready for use.

B.2.1.2. Test story 2: Failed creation (duplicated name)

Test objective: Attempt to create an account with a duplicated username (Extension 4a).

Alice wants to create an account in the system.

The system provides Alice with an interface with enough fields for her to create an account. Alice introduces her desired username “Alice”, which will be used as a username. She also introduces her desired password “1234”.

The system rejects the creation of the account and informs Alice that there is already an account using that very same username.

B.2.1.3. Test story 3: Failed creation (name too long)

Test objective: Attempt to create an account with a username too long (Extension 4a).

Alice wants to create an account in the system.

The system provides Alice with an interface with enough fields for her to create an account. Alice introduces her desired username “Alice Pleasance Lidell from Wonderland, beneath Oxfordshire”, which will be used as a username. She also introduces her desired password “1234”.

The system rejects the creation of the account and informs Alice that the chosen username is too long.

B.2.2. Log into an account

B.2.2.1. Test story 1: Log into an account

Test objective: Log into an account (Main success scenario).

Alice wants to log into her account in the system.

The system provides Alice with an interface with enough fields for her to log into an account. Alice introduces her username “Alice”. She also introduces her password “1234”.

The system informs Alice that she has just logged in successfully.

B.2.2.2. Test story 2: Log into an account (no account)

Test objective: Log into an account without having one (Extension 1a).

Alice wants to log into her account in the system.

The system provides Alice with an interface with enough fields for her to log into an account. Alice does not have an account but she wants to use the system so she now wants to create an account and inform the system.

The system provides Alice with an interface with enough fields for her to create an account. Alice introduces her desired username “Alice”, which will be used as a username. She also introduces her desired password “1234”.

The system confirms the creation of the account with the introduced username “Alice” and password. The system informs Alice that her account has been successfully created and it is now ready for use.

With the account created, Alice informs the system that she wants to log into her account. The system provides Alice with an interface with enough fields for her to log into an account. Alice introduces her username “Alice”. She also introduces her password “1234”.

The system informs Alice that she has just logged in successfully.

B.2.2.3. Test story 3: Failed login (invalid password or username)

Test objective: Attempt to log into an account with an invalid password or username (Extension 4a).

Alice wants to log into her account in the system.

The system provides Alice with an interface with enough fields for her to log into an account. Alice introduces her username “Alice”. She also introduces her password incorrectly “12345”.

The system informs Alice that the username or password are not correct and she can not be logged in.

B.2.3. Update an account

B.2.3.1. Test story 1: Update an account

Test objective: Update an account (Main success scenario).

Alice wants to update information regarding her account.

She logs into the system successfully. The system provides Alice with all the details of her account which include “username: Alice”, “password: 1234” and “e-mail:“.

Alice fills in information regarding the e-mail. Now it is “e-mail: alice@wonderland.com”. The system asks Alice to confirm the modification. She confirms the modification.

The system informs Alice that the update of the account has been done successfully. Now the e-mail direction appears as “alice@wonderland.com”.

B.2.3.2. Test story 2: Failed update of an account (invalid data)

Test objective: Attempt to update an account with and invalid e-mail (Extension 6a).

Alice wants to update information regarding her account.

She logs into the system successfully. The system provides Alice with all the details of her account which include “username: Alice”, “password: 1234” and “e-mail:“.

Alice fills in information regarding the e-mail. Now it is “e-mail: al”. The system asks Alice to confirm the modification.

She confirms the modification. The system informs Alice that the update of the account is not possible because the provided e-mail address appears to be invalid.

B.2.4. Delete an account

B.2.4.1. Test story 1: Delete an account

Test objective: Deletion of an account (Main success scenario).

Alice wants to delete her account.

She logs into the system successfully. The system asks Alice to provide further authentication before it can delete the account, and provides her with an interface with enough fields for her to provide a username and a password. Alice introduces her username “Alice” and her password “1234”.

The system asks Alice to confirm the deletion of the account. She confirms the deletion of the account.

The system logs Alice out and informs her that her account has been deleted successfully. Now the account is not accessible by anyone.

B.2.4.2. Test story 2: Failed deletion of an account (invalid authentication)

Test objective: Attempt to delete an account with an invalid authentication (Extension 4a).

Alice wants to delete her account.

She logs into the system successfully. The system asks Alice to provide further authentication before it can delete the account, and provides her with an interface with enough fields for her to provide a username and a password. Alice introduces her username "Alice" and her password "1233".

The system informs Alice that she failed the authentication and logs her out.

B.2.5. Log out of an account

B.2.5.1. Test story 1: Log out of an account

Test objective: Log out of an account (Main success scenario).

Alice wants to log out of her account.

She logs into the system successfully. Alice informs the system that she wants to be logged out. The system logs Alice out and informs her about that fact.

B.3. Reminder

B.3.1. Creation of a reminder schedule

B.3.1.1. Test story 1: Log out of an account

Test objective: Create a reminder schedule (Main success scenario).

Alice wants to create a new reminder schedule.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means for her to create and define a reminder schedule.

Alice asks the system to create a new reminder schedule. She informs the system that she wants to be reminded 2 hours before the time a task is due. She also informs that she wants to be reminded via e-mail using the account "alice@wonderland.com" The system asks Alice to confirm the creation.

She confirms the creation. The system informs Alice that the creation of the schedule has been done successfully. Now whenever a task is at 2 hours from being due, an e-mail will be sent informing of that fact to Alice's provided e-mail address.

B.3.1.2. Test story 2: Failed creation of a reminder schedule (invalid reminder method)

Test objective: Attempt to create a reminder schedule with and invalid reminder method (Extension 6a).

Alice wants to create a new reminder schedule.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means for her to create and define a reminder schedule.

Alice asks the system to create a new reminder schedule. She informs the system that she wants to be reminded 2 hours before the time a task is due. She also informs that she wants to be reminded via e-mail using the account "alicewonderland.com".

The system asks Alice to confirm the creation. She confirms the creation.

The system informs Alice that the creation of the schedule is not possible because the provided e-mail address appears to be invalid.

B.3.2. Read a reminder schedule

B.3.2.1. Test story 1: Reading of a reminder schedule

Test objective: Reading a reminder schedule (Main success scenario).

Alice wants to read the existing reminder schedules.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means that define a reminder schedule.

In this case, the system shows Alice that she has a reminder schedule set at 2 hours before the time a task is due. It also shows that this same schedule will send the notifications via an e-mail account to alice@wonderland.com.

B.3.2.2. Test story 2: Reading of an unset reminder schedule

Test objective: Reading an unset reminder schedule (Extension 3a).

Alice wants to read the existing reminder schedules.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means that define a reminder schedule.

In this case, because Alice did not have any reminder schedule, all the information is blank. The system informs Alice that she does not have any reminder schedule.

B.3.3. Update a reminder schedule

B.3.3.1. Test story 1: Update a reminder schedule

Test objective: Update a reminder schedule (Main success scenario).

Alice wants to update an existing reminder schedule.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means that define a reminder schedule.

In this case, the system shows Alice that she has a reminder schedule set at 2 hours before the time a task is due. It also shows that this same schedule will send the notifications via an e-mail account to alice@wonderland.com.

Alice modifies the reminder to be send every day there is a task due at 7 a.m. She also modifies the method she would like to use from an e-mail to an SMS and introduces her mobile phone number.

The system asks Alice to confirm the update. She confirms the update.

The system informs Alice that the update of the schedule has been done successfully. Now, whenever a task is due, one SMS will be sent the same day at 7 a.m. informing of that fact to Alice's provided mobile phone number.

B.3.3.2. Test story 2: Failed update of a reminder schedule (invalid reminder method)

Test objective: Attempt to update a reminder schedule with and invalid reminder method (Extension 6a).

Alice wants to update an existing reminder schedule.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means that define a reminder schedule.

In this case, the system shows Alice that she has a reminder schedule set at 2 hours before the time a task is due. It also shows that this same schedule will send the notifications via an e-mail account to alice@wonderland.com.

Alice modifies the reminder to be send every day there is a task due at 7 a.m. She also modifies the method she would like to use changing the e-mail address from "alice@wonderland.com" to "aliceengland.com".

The system asks Alice to confirm the update. She confirms the update.

The system informs Alice that the modification of the schedule is not possible, because the provided e-mail address appears to be invalid.

B.3.4. Delete a reminder schedule

B.3.4.1. Test story 1: Deletion of a reminder schedule

Test objective: Delete a reminder schedule (Main success scenario).

Alice wants to delete an existing reminder schedule.

She logs into the system successfully. The system provides Alice with an interface with all the necessary means that define a reminder schedule. In this case, the system shows Alice that she has a reminder schedule set at 2 hours before the time a task is due. It also shows that this same schedule will send the notifications via an e-mail account to alice@wonderland.com.

Alice informs the system that she wants to delete a reminder schedule and selects the e-mail address alice@wonderland.com.

The system asks Alice to confirm the deletion. She confirms the deletion.

The system informs Alice that the deletion of the schedule associated with the e-mail address has been successful. Now, she will not be receive any further notifications for that set schedule on that e-mail address.

B.3.4.2. Test story 2: Failed deletion of a reminder schedule (no schedules)

Test objective: Attempt to delete a reminder schedule when the system has none (Extension 2a).

Alice wants to delete an existing reminder schedule.

She logs into the system successfully. The system informs Alice that there are no reminders in the system to be deleted.

B.3.5. Send reminder

B.3.5.1. Test story 1: Send a reminder

Test objective: Send a reminder a according to the user's preferences (Main success scenario).

The clock has advanced one more minute. Now it is 7 a.m.

The system retrieves the reminder schedules for all of its accounts, in this case Alice's, Bob's, Carol's and Charlie's accounts. The systems checks if there are any reminders due at 7 a.m. There is one reminder schedule set at 7 a.m.for Alice and another one for Carol.

The system verifies that none of this reminders has been sent yet. It also checks the medium it should use, SMS for Alice and e-mail for Carol. The system checks if

Alice has any tasks due today and finds out that indeed she has “Go to the cinema with Bob (Saturday evening)”. On the other hand, Carol has no tasks to be sent today.

The system verifies that Alice’s reminder has not yet been sent. The system sends Alice an SMS informing her of her upcoming task for that day.

B.4. Customisation

B.4.1. Change language

B.4.1.1. Test story 1: Change the language

Test objective: Change the language (Main success scenario).

Alice wants to change the system’s language. The system provides Alice with a list of all the languages available which include “Català”, “English” and “Norsk(bokmål)”. Alice chooses “Norsk(bokmål)”.

The system asks Alice to confirm the language change. She confirms the language change.

The system updates all its related information according to the chosen language. Now all menus and system-generated text are written in “Norsk(bokmål)”.

B.4.2. Show weekly planner

B.4.2.1. Test story 1: Show weekly planner

Test objective: Generate and show a weekly planner (Main success scenario).

Alice wants to see her weekly plan.

She logs into the system successfully. The system provides Alice with a list of all her tasks due in less than 7 days, ordered according to the due date and separated according to the day they are due.

In this case “Go to the cinema with Bob (Saturday evening)” and “Go to the post office to send a package to Carol (Monday morning)”.

B.5. Contact and group

B.5.1. Add a contact

B.5.1.1. Test story 1: Addition of a contact

Test objective: Add a contact (Main success scenario).

Alice wants to add a contact to her system's account.

She logs into the system successfully. The system provides Alice with an interface she can use in order to look for other users within the system. Alice introduces the name "Charles Dodgson".

The system confirms the addition of a new contact with username "Charles Dodgson" to Alice's account, and her list of contacts now includes him.

B.5.1.2. Test story 2: Failed addition of a contact (invalid username)

Test objective: Attempt to add a contact whose username it is not in the system (Extension 5a).

Alice wants to add a contact to her system's account.

She logs into the system successfully. The system provides Alice with an interface she can use in order to look for other users within the system. Alice introduces the name "Lewis Carroll".

The system informs Alice that the addition of the contact is not possible because there is no account in the system with the name "Lewis Carroll".

B.5.1.3. Test story 3: Failed addition of a contact (already added)

Test objective: Attempt to add a contact already added (Extension 5b).

Alice wants to add a contact to her system's account.

She logs into the system successfully. The system provides Alice with an interface she can use in order to look for other users within the system. Alice introduces the name "Bob".

The system informs Alice that the addition of the contact is not possible because "Bob" is already in her contact list.

B.5.2. Read a contact

B.5.2.1. Test story 1: Reading of a contact

Test objective: Read a contact (Main success scenario).

Alice wants to read a contact from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Bob" and informs the system that she wants to read some of his information.

The system provides Alice with further information about the contact, like in which groups he is in and which tasks have been shared with him.

B.5.2.2. Test story 2: Failed reading of a contact (no contacts)

Test objective: Attempt to read a contact when the system has none (Extension 2a).

Alice wants to read a contact from her system's account.

She logs into the system successfully. The system informs Alice that there are no contacts to read.

B.5.3. Delete a contact

B.5.3.1. Test story 1: Reading of a contact

Test objective: Read a contact (Main success scenario).

Alice wants to delete a contact from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Bob" and informs the system that she wants to delete him.

The system asks Alice to confirm the deletion. She confirms the deletion.

The system informs Alice that the removal of the contact has been successful, and the list of contacts does not include the deleted contact anymore.

B.5.3.2. Test story 2: 2 Failed deletion of a contact (no contacts)

Test objective: Attempt to delete a contact when the system has none (Extension 2a).

Alice wants to delete a contact from her system's account.

She logs into the system successfully. The system informs Alice that there are no contacts.

B.5.4. Create a group

B.5.4.1. Test story 1: Creation of a group

Test objective: Create a group (Main success scenario).

Alice wants to create a group in her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends" and "Coworkers".

Alice asks the system to create a new group with the subject "Family".

The system confirms the creation of a new group with name "Family" to Alice's account, and her list of groups now includes it.

B.5.4.2. Test story 2: Failed creation (duplicated name)

Test objective: Attempt to create a group with a duplicated name (Extension 4a).

Alice wants to create a group in her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends", "Coworkers" and "Family".

Alice asks the system to create a new group with the subject "Family".

The system rejects the creation of the new group and informs Alice that she already has that group in the system (i.e. a group with the same name).

B.5.4.3. Test story 3: Failed creation (blank name)

Test objective: Attempt to create a group with a blank name (Extension 4a).

Alice wants to create a group in her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends" and "Coworkers".

Alice asks the system to create a new group without any subject.

The system rejects the creation of the new group and informs Alice that every group must at least have a subject.

B.5.5. Read a group

B.5.5.1. Test story 1: Reading of a group

Test objective: Read a group (Main success scenario).

Alice wants to read a group from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends" and "Coworkers".

Alice chooses the group "Friends" and informs the system that she wants to read some of its information.

The system provides Alice with further information about the group, like which of her contacts belong to it.

B.5.5.2. Test story 2: Failed reading of a group (no groups)

Test objective: Attempt to read a group when the system has none (Extension 2a).

Alice wants to read a group from her system's account.

She logs into the system successfully. The system informs Alice that there are no groups of contacts.

B.5.6. Delete a group

B.5.6.1. Test story 1: Deletion of a group

Test objective: Delete a group (Main success scenario).

Alice wants to delete a group from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends", "Coworkers" and "Family".

Alice chooses the group "Family" and informs the system that she wants to delete it. The system asks Alice to confirm the deletion.

She confirms the deletion. The system informs Alice that the removal of the group has been successful, and the list of groups does not include the deleted group anymore.

B.5.6.2. Test story 2: Failed deletion of a group (no groups)

Test objective: Attempt to delete a group when the system has none (Extension 2a).

Alice wants to delete a group from her system's account.

She logs into the system successfully. The system informs Alice that there are no groups of contacts.

B.5.6.3. Test story 3: Failed deletion of a group (group not empty)

Test objective: Attempt to delete a not empty group (Extension 3a).

Alice wants to delete a group from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added groups which are "Friends", "Coworkers" and "Family".

Alice chooses the group "Friends" and informs the system that she wants to delete it.

The system informs Alice that the removal of the group is not possible because it still has some contacts in it, in this case "Bob".

B.5.7. Add a contact to a group

B.5.7.1. Test story 1: Addition of a contact to a group

Test objective: Add a contact to a group (Main success scenario).

Alice wants to add one of her contacts to one of her groups from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Bob" and informs the system that she wants to add him to one of her groups.

The system provides Alice with a list of her previously added groups which are "Friends" and "Coworkers". Alice chooses the group "Coworkers" and informs the system that she wants to add him to that group.

The system confirms the addition of the contact named "Bob" to the group with name "Coworkers" to Alice's account, and updates the contacts' and groups' status to reflect this fact.

B.5.7.2. Failed addition of a contact to a group (no contacts)

Test objective: Attempt to add a contact to a group when the system has no contacts (Extension 2a).

Alice wants to add one of her contacts to one of her groups from her system's account.

She logs into the system successfully. The system informs Alice that there are no contacts to add.

B.5.7.3. Failed addition of a contact to a group (no groups)

Test objective: Attempt to add a contact to a group when the system has no groups (Extension 2b).

Alice wants to add one of her contacts to one of her groups from her system's account.

She logs into the system successfully. The system informs Alice that there are no groups where to contact could be added.

B.5.7.4. Failed addition of a contact to a group (already added)

Test objective: Attempt to add a contact to a group when that contact is already in that group (Extension 6a).

Alice wants to add one of her contacts to one of her groups from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Bob" and informs the system that she wants to add him to one of her groups.

The system provides Alice with a list of her previously added groups which are "Friends" and "Coworkers". Alice chooses the group "Friends" and informs the system that she wants to add him to that group.

The system rejects the addition of the contact named "Bob" to the group with name "Friends" to Alice's account, informing that in fact that contact is already in.

B.5.8. Remove a contact from a group

B.5.8.1. Test story 1: Removal of a contact from a group

Test objective: Remove a contact from a group (Main success scenario).

Alice wants to remove one of her contacts from one of her groups from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Bob" and informs the system that she wants to remove him from one of her groups.

The system provides Alice with a list of all her groups Bob is in, which are "Friends" and "Coworkers". Alice chooses the group "Coworkers" and informs the system that she wants to remove him from that group.

The system asks Alice to confirm the deletion. She confirms the deletion.

The system confirms the removal of the contact named "Bob" from the group with name "Coworkers" from Alice's account. The system updates Bob's and the group's status to reflect this fact.

B.5.8.2. Test story 2: Failed removal of a contact from a group (no contacts)

Test objective: Attempt to remove a contact from a group when the system has no contacts (Extension 2a).

Alice wants to remove one of her contacts from one of her groups from her system's account.

She logs into the system successfully. The system informs Alice that there are no contacts to remove.

B.5.8.3. Test story 3: Failed removal of a contact from a group (no groups)

Test objective: Attempt to remove a contact from a group when the system has no groups (Extension 2b).

Alice wants to remove one of her contacts from one of her groups from her system's account.

She logs into the system successfully. The system informs Alice that there are no groups where to contact could be removed from.

B.5.8.4. Test story 4: Failed removal of a contact from a group (the contact is not in any group)

Test objective: Attempt to remove a contact from a group when that contact does not belong to any group (Extension 6a).

Alice wants to remove one of her contacts from one of her groups from her system's account.

She logs into the system successfully. The system provides Alice with a list of her previously added contacts which are "Bob", "Carol" and "Charlie".

Alice chooses the contact "Charlie" and informs the system that she wants to remove him from one of her groups.

The system rejects the removal of the contact named "Charlie" and informs that he does not belong to any of her groups.

B.6. List of tasks

B.6.1. Create a list

B.6.1.1. Test story 1: Creation of a list of tasks

Test objective: Create a list of tasks (Main success scenario).

Alice wants to create a list of tasks in her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice asks the system to create a new list with the subject "Work".

The system confirms the creation of a new list of tasks with name "Work" to Alice's account, and her list of lists of tasks now includes it.

B.6.1.2. Test story 2: Failed creation (duplicated name)

Test objective: Attempt to create a list with a duplicated name (Extension 4a).

Alice wants to create a list of tasks in her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice asks the system to create a new list with the subject "Personal".

The system rejects the creation of the new list and informs Alice that she already has that list of tasks in the system (i.e. a list with the same name).

B.6.1.3. Test story 3: Failed creation (blank name)

Test objective: Attempt to create a list with a blank name (Extension 4a).

Alice wants to create a list of tasks in her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice asks the system to create a new list without any subject.

The system rejects the creation of the new list and informs Alice that every list must at least have a subject.

B.6.2. Read a list

B.6.2.1. Test story 1: Reading of a list

Test objective: Read a list of tasks (Main success scenario).

Alice wants to read a list of tasks in her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list of tasks "Personal" and informs the system that she wants to read some of its information.

The system provides Alice with further information about the list, like which tasks are assigned to it. In this case it would be "Go to the cinema with Bob (Saturday evening)".

B.6.3. Update a list

B.6.3.1. Test story 1: Updating of a list

Test objective: Update a list of tasks (Main success scenario).

Alice wants to update a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to update some of its information.

The system provides Alice with further information about the list, like the tasks that belong to it. In this case this is "Go to the cinema with Bob (Saturday evening)".

Alice changes the name of the list to "Personal stuff".

The system asks Alice to confirm the modification. She confirms the modification. The system informs Alice that the modification of the list has been successful, and the list of lists now includes the modified version of the name description.

B.6.3.2. Test story 2: Failed update (no editable lists)

Test objective: Attempt to update a list when the system does not have any editable ones (Extension 2a).

Alice wants to update a list of tasks from her system's account.

She logs into the system successfully. The system informs Alice that there are no editable lists (i.e. there is only the "Sent" and the "Inbox" lists).

B.6.3.3. Test story 3: Failed update (duplicated name)

Test objective: Attempt to update a list with a duplicated name (Extension 8a).

Alice wants to update a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to update some of its information.

The system provides Alice with further information about the list, like the tasks that belong to it. In this case this is "Go to the cinema with Bob (Saturday evening)". Alice changes the name of the list to "Inbox".

The system rejects the update of the list and informs Alice that she already has that list of tasks in the system (i.e. a list with the same name).

B.6.3.4. Test story 4: Failed update (non-editable list)

Test objective: Attempt to update a non-editable list (Extension 8a).

Alice wants to update a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Sent" and informs the system that she wants to update some of its information.

The system rejects the update of the list and informs Alice that the list "Sent" is non-updatable.

B.6.4. Delete a list

B.6.4.1. Test story 1: Deletion of a list

Test objective: Delete a list of tasks (Main success scenario).

Alice wants to delete a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to delete it.

The system asks Alice to confirm the deletion. She confirms the deletion. The system informs Alice that the deletion of the list has been successful, and the list of lists now does not include "Personal" anymore.

B.6.4.2. Test story 2: Failed deletion (no editable lists)

Test objective: Attempt to delete a list when the system does not have any editable ones (Extension 2a).

Alice wants to delete a list of tasks from her system's account.

She logs into the system successfully. The system informs Alice that there are no editable lists (i.e. there is only the "Sent" and the "Inbox" lists).

B.6.4.3. Test story 3: Failed deletion (non-empty list)

Test objective: Attempt to delete a non-empty list (Extension 4a).

Alice wants to delete a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to delete it.

The system rejects the deletion of the list and informs Alice that the list "personal" still has some tasks in it.

B.6.4.4. Test story 4: Failed deletion (deletion-locked list)

Test objective: Attempt to delete a delete-locked list (Extension 4b).

Alice wants to delete a list of tasks from her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Sent" and informs the system that she wants to delete it.

The system rejects the deletion of the list and informs Alice that the list "Sent" is delete-locked.

B.6.5. Set default list

B.6.5.1. Test story 1: Setting of the default list

Test objective: Set the default list (Main success scenario).

Alice wants to set the default list of tasks for her system's account.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to set it as her default list. The system asks Alice to confirm the setting.

She confirms the setting. The system informs Alice that the setting of the list as the default one has been successful, and the default list is now "Personal".

B.6.5.2. Test story 2: Failed setting of the default list (default list already set)

Test objective: Attempt to set the default a list when the system already has one (Extension 2a).

Alice wants to set the default list of tasks for her system's account.

She logs into the system successfully. The system informs Alice that the default list is already set.

B.6.6. Unset default list

B.6.6.1. Test story 1: Unsetting of the default list

Test objective: Unset the default list (Main success scenario).

Alice wants to unset the default list of tasks for her system's account.

She logs into the system successfully. The system informs Alice that the default list has been successfully unset. Now the system does not have a default list.

B.6.6.2. Test story 2: Failed unsetting of the default list (no default list)

Test objective: Attempt to unset the default a list when the system does not have one (Extension 2a).

Alice wants to unset the default list of tasks for her system's account.

She logs into the system successfully. The system informs Alice that there is no default list to unset.

B.6.7. Share a list with some contacts

B.6.7.1. Test story 1: Sharing of a list with contacts

Test objective: Share a list with some of the user's contacts (Main success scenario).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to share it with some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to share the list of tasks with him. The system asks Alice to confirm the sharing.

She confirms the sharing. The system informs Alice that the sharing of the chosen list of tasks with the chosen contacts has been successful.

Now the list is marked as shared with those contacts (with acceptance pending status) and the system assigns it to all of them, in this case only to Bob.

B.6.7.2. Test story 2: Failed sharing of a list of tasks (no unrestricted lists)

Test objective: Attempt to share a list of tasks when the system does not have any unrestricted ones (Extension 2a).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unrestricted lists of tasks to share.

B.6.7.3. Test story 3: Failed sharing of a list of tasks (no contacts)

Test objective: Attempt to share a list of tasks when she does not have any contact in the system (Extension 2b).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no contacts to share any lists of tasks with.

B.6.7.4. Test story 4: Failed sharing of a list of tasks (chosen contacts already have the list)

Test objective: Attempt to share a list of tasks with some of the user's contacts, when some of them already have the list shared previously (Extension 8a).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to share it with some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to share the list of tasks with him. The system asks Alice to confirm the sharing.

She confirms the sharing. The system rejects the sharing and informs Alice that the chosen list of tasks was already shared with some of the chosen contacts, in this case with "Bob".

B.6.8. Share a list with some groups

B.6.8.1. Test story 1: Sharing of a list with groups

Test objective: Share a list with some of the user's groups (Main success scenario).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to share it with some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are "Friends" and "Coworkers".

Alice chooses the group of contacts "Friends" and informs the system that she wants to share the list of tasks with them. The system asks Alice to confirm the sharing.

She confirms the sharing. The system informs Alice that the sharing of the chosen list of tasks with the chosen groups of contacts has been successful.

Now the list is marked as shared with those contacts in the groups (with acceptance pending status) and the system assigns it to all of them, in this case to Bob and Carol.

B.6.8.2. Test story 2: Failed sharing of a list of tasks (no unrestricted lists)

Test objective: Attempt to share a list of tasks when the system does not have any unrestricted ones (Extension 2a).

Alice wants to share a list of tasks from the system with some of her contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no unrestricted lists of tasks to share.

B.6.8.3. Test story 3: Failed sharing of a list of tasks (no groups)

Test objective: Attempt to share a list of tasks when she does not have any groups of contacts in the system (Extension 2b).

Alice wants to share a list of tasks from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system informs Alice that there are no groups of contacts to share any lists of tasks with.

B.6.8.4. Test story 4: Failed sharing of a list of tasks (empty group)

Test objective: Attempt to share a list of tasks with some groups of contacts that are empty (Extension 6a).

Alice wants to share a list of tasks from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to share it with some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends”, “Coworkers” and “Unknown”.

Alice chooses the group of contacts “Unknown” and informs the system that she wants to share the list of tasks with them. The system asks Alice to confirm the sharing.

She confirms the sharing. The system rejects the sharing and informs Alice that some of the chosen groups of contacts did not contain any contact in it, in this case the group “Unknown”.

B.6.8.5. Test story 5: Failed sharing of a list of tasks (already shared)

Test objective: Attempt at sharing a list of tasks with some of the user's contacts belonging to groups, when some of them already have the list of tasks shared previously (Extension 8a).

Alice wants to share a list of tasks from the system with some of her groups of contacts that also use the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to share it with some of her contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are "Friends" and "Coworkers".

Alice chooses the group of contacts "Friends" and informs the system that she wants to share the list of tasks with them. The system asks Alice to confirm the sharing.

She confirms the sharing. The system rejects the sharing and informs Alice that the chosen list of tasks was already shared with some of the contacts belonging to the chosen groups, in this case "Bob".

B.6.9. Publish a list for some contacts

B.6.9.1. Test story 1: Publication of a list of tasks for some contacts

Test objective: Publish a list of tasks for some of the user's contacts (Main success scenario).

Alice wants to publish a list of tasks from the system for some of her contacts that also use it.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to publish it for some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to publish the list of tasks for him. The system asks Alice to confirm the publication.

She confirms the publication. The system informs Alice that the publication of the chosen list of tasks for the chosen contacts has been successful.

Now the list is marked as published for those contacts and the system will let them all read the list, in this case only to Bob.

B.6.9.2. Test story 2: Failed publication of a list of tasks for some contacts (no contacts)

Test objective: Attempt to publish a list of tasks when she does not have any contact in the system (Extension 2a).

Alice wants to publish a list of tasks from the system for some of her contacts that also use it.

She logs into the system successfully. The system informs Alice that there are no contacts to publish any lists of tasks for.

B.6.9.3. Test story 3: Failed publication of a list for some contacts (chosen contacts already have the list published for them)

Test objective: Attempt to publish a list of tasks for some of the user's contacts, when some of them already have access to the list, previously granted (Extension 8a).

Alice wants to publish a list of tasks from the system for some of her contacts that also use it.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are "Inbox", "Personal" and "Sent".

Alice chooses the list "Personal" and informs the system that she wants to publish it for some of her contacts.

The system provides Alice with a list of all her contacts that use the system. These contacts are "Bob" and "Carol".

Alice chooses the contact "Bob" and informs the system that she wants to publish the list of tasks for him. The system asks Alice to confirm the publication.

She confirms the publication. The system informs Alice that the publication of the chosen list of tasks for the chosen contacts has been successful.

The system rejects the publication and informs Alice that the chosen list of tasks was already published for some of the chosen contacts, in this case "Bob".

B.6.10. Publish a list for some groups

B.6.10.1. Test story 1: Publication of a list of tasks for some groups

Test objective: Publish a list of tasks for some of the user's groups (Main success scenario).

Alice wants to publish a list of tasks from the system for some of her groups of contacts that also use it.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to publish it for some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends” and “Coworkers”.

Alice chooses the group of contacts “Friends” and informs the system that she wants to publish the list of tasks for them. The system asks Alice to confirm the publication. She confirms the publication.

The system informs Alice that the publication of the chosen list of tasks for the chosen groups of contacts has been successful.

Now the list is marked as published for those contacts in the groups, and the system will let them all read the list, in this case to Bob and Carol.

B.6.10.2. Test story 2: Failed publication of a list of tasks for some groups (no groups)

Test objective: Attempt to publish a list of tasks when she does not have any groups of contacts in the system (Extension 2a).

Alice wants to publish a list of tasks from the system for some of her groups of contacts that also use it.

She logs into the system successfully. The system informs Alice that there are no groups of contacts to publish any lists of tasks for.

B.6.10.3. Test story 3: Failed publication of a list of tasks for some groups (empty group)

Test objective: Attempt to publish a list of tasks for some of the user’s groups of contacts which are empty (Extension 6a).

Alice wants to publish a list of tasks from the system for some of her groups of contacts that also use it.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to publish it for some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends”, “Coworkers” and “Unknown”.

Alice chooses the group of contacts “Unknown” and informs the system that she wants to publish the list of tasks for them. The system asks Alice to confirm the publication. She confirms the publication.

The system rejects the publication and informs Alice that some of the chosen groups of contacts did not contain any contact in it, in this case the group “Unknown”.

B.6.10.4. Test story 4: Failed publication of a list of tasks for some groups (already published)

Test objective: Attempt to publish a list of tasks for some of the user’s groups of contacts, when some of them already have access to the list, previously granted (Extension 8a).

Alice wants to publish a list of tasks from the system for some of her groups of contacts that also use it.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to publish it for some of her groups of contacts.

The system provides Alice with a list of all her groups of contacts that use the system. These groups of contacts are “Friends” and “Coworkers”.

Alice chooses the group of contacts “Friends” and informs the system that she wants to publish the list of tasks for them. The system asks Alice to confirm the publication. She confirms the publication.

The system rejects the publication and informs Alice that the chosen list of tasks was already shared with some of the contacts belonging to the chosen groups, in this case “Bob”.

B.6.11. Publish a list for anyone

B.6.11.1. Test story 1: Publication of a list of tasks for some groups

Test objective: Publish a list of tasks for some of the user’s groups (Main success scenario).

Alice wants to publish a list of tasks from the system for anyone.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to publish it for anyone.

The system asks Alice to confirm the publication. She confirms the publication. The system informs Alice that the publication of the chosen list of tasks for anyone has been successful.

Now the list is marked as published for anyone, and the system will let them all read the list.

B.6.11.2. Test story 2: Failed publication of a list for anyone (already published)

Test objective: Attempt to publish a list of tasks for anyone, when it is already public (Extension 8a).

Alice wants to publish a list of tasks from the system for anyone.

She logs into the system successfully. The system provides Alice with a list of her currently existing lists of tasks which are “Inbox”, “Personal” and “Sent”.

Alice chooses the list “Personal” and informs the system that she wants to publish it for anyone. The system asks Alice to confirm the publication.

She confirms the publication. The system rejects the publication and informs Alice that the chosen list of tasks was already published for anyone.

B.6.12. Unpublish a list for some contacts

B.6.12.1. Test story 1: Unpublication of a list for some contacts

Test objective: Restrict previously granted reading access to a list of tasks to some of the user’s contacts (Main success scenario).

This test story was omitted and written directly in CSTL as a test case.

B.6.12.2. Test story 2: Failed unpublication of a list for some contacts (Not public for them)

Test objective: Attempt to unpublish for some of the user’s contacts a list of tasks which was not previously published for them (Extension).

This test story was omitted and written directly in CSTL as a test case.

B.6.13. Unpublish a list for some groups

Test stories related to this test case were not written, because test cases were not needed, as this use case did not make it into the presented conceptual schema (chapter 8).

B.6.14. Unpublish a list for anyone

B.6.14.1. Test story 1: Unpublication of a list for anyone

Test objective: Unpublish for anonymous access a previously list published (Main success scenario).

This test story was omitted and written directly in CSTL as a test case.

B.6.14.2. Test story 2: Failed unpublication of a list for anyone (Not public for anyone)

Test objective: Attempt to unpublish for anonymous access a list which already not allowed anonymous access (Extension).

This test story was omitted and written directly in CSTL as a test case.

B.6.15. Accept a shared list

B.6.15.1. Test story 1: Acceptance of a shared list

Test objective: Accept a shared list of tasks of some of the user's contacts (Main success scenario).

This test story was omitted and written directly in CSTL as a test case.

B.6.15.2. Test story 2: Failed acceptance of a shared list (Not a shared list)

Test objective: Attempt to accept a shared list that is not shared (Extension 9a).

This test story was omitted and written directly in CSTL as a test case.

B.6.16. Reject a shared list

B.6.16.1. Test story 1: Rejection of a shared list

Test objective: Reject a shared list of tasks of some of the user's contacts (Main success scenario).

This test story was omitted and written directly in CSTL as a test case.

B.6.16.2. Test story 2: Failed rejection of a shared list (Not a shared list)

Test objective: Attempt to reject a shared list that is not shared (Extension 9a).

This test story was omitted and written directly in CSTL as a test case.

B.6.17. Archive lists

Test stories related to this test case were not written, because test cases were not needed, as this use case did not make it into the presented conceptual schema (chapter 8).

B.6.18. Unarchive lists

Test stories related to this test case were not written, because test cases were not needed, as this use case did not make it into the presented conceptual schema (chapter 8).

B.7. Location

B.7.1. Create a location

B.7.1.1. Test story 1: Creation of a location

Test objective: Create a location (Main success scenario).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice asks the system to create a new location. She introduces the position that she wants the location to represent.

The system asks Alice to choose a name to identify the location. She chooses “Shopping mall” as a name.

The system confirms the creation of the location with name “Shopping mall”, and the representation of locations now includes it.

B.7.1.2. Test story 2: Creation of a location, longer alternative

Test objective: Create a location using its physical address (Extension 3a).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice asks the system to create a new location. She introduces the address that she wants the location to represent, which is “Barcelona”.

The system returns various matches with possible candidates, such as “Barcelona, Spain” and “Barcelona, Brazil”. Alice chooses “Barcelona, Spain”.

The system asks Alice to choose a name to identify the location. She chooses “Hometown” as a name.

The system confirms the creation of the location with name “Hometown”, and the representation of locations now includes it.

B.7.1.3. Test story 3: Creation of a location using the street address

Test objective: Create a location using its physical street address (Extension 3a.2a).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice asks the system to create a new location. She introduces the address that she wants the location to represent, which is “Barcelona plaça de Catalunya”.

The system asks Alice to choose a name to identify the location. She chooses “Hometown main square” as a name.

The system confirms the creation of the location with name “Hometown main square”, and the representation of locations now includes it.

B.7.1.4. Test story 4: Failed creation of a location using the street address

Test objective: Attempt to create a location using its physical street address (Extension 3a.2b).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice asks the system to create a new location. She introduces the address that she wants the location to represent, which is "Trinsic".

The system informs Alice that it could not find "Trinsic" and rejects the creation of the location.

B.7.1.5. Test story 5: Failed creation (duplicated name)

Test objective: Attempt to create a location with a duplicated name (Extension 6a).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are "Alice's home", "Alice's office" and "Post office".

Alice asks the system to create a new location. She introduces the position that she wants the location to represent.

The system asks Alice to choose a name to identify the location. She chooses "Alice's home" as a name.

The system rejects the creation of the new location and informs Alice that she already has that same location (i.e. a location with the same).

B.7.1.6. Test story 6: Failed creation (blank name)

Test objective: Attempt to create a location with a duplicated name (Extension 6a).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are "Alice's home", "Alice's office" and "Post office".

Alice asks the system to create a new location. She introduces the position that she wants the location to represent.

The system asks Alice to choose a name to identify the location. She does not introduce any name but asks the system to proceed.

The system rejects the creation of the new location and informs Alice that every location must at least have a name to identify it.

B.7.1.7. Test story 7: Failed creation (name too long)

Test objective: Attempt to create a location with a too long name (Extension 6a).

Alice wants to add a location to the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice asks the system to create a new location. She introduces the position that she wants the location to represent.

The system asks Alice to choose a name to identify the location. She chooses a 1000 word description of the place as a name.

The system rejects the creation of the new location and informs Alice that the name is too long.

B.7.2. Read a location

B.7.2.1. Test story 1: Reading of a location

Test objective: Reading a location (Main success scenario).

Alice wants to read a location from the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice chooses the location “Post office” and informs the system that she wants to read more information about it.

The system provides Alice with further information about the location, like the position on a map with the exact location of the post office and which tasks are related to the position.

B.7.2.2. Test story 2: Failed read (no locations)

Test objective: Attempt to read a location when the system has none (Extension 2a).

Alice wants to read a location from the system.

She logs into the system successfully. The system informs Alice that there are no locations.

B.7.3. Update a location

B.7.3.1. Test story 1: Modification of a location

Test objective: Update location (Main success scenario).

Alice wants to update a location from the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice chooses the location “Post office” and informs the system that she wants to update some of its information.

The system provides Alice with further information about the location, like the position on a map with the exact location of the post office and which tasks are related to the position.

Alice changes the name of the location to “Post office headquarters”.

The system asks Alice to confirm the modification. She confirms the modification.

The system informs Alice that the modification of the location has been successful, and the list of locations now includes the modified version of the name description.

B.7.3.2. Test story 2: Failed update (no locations)

Test objective: Attempt to update a location when the system has none (Extension 2a).

Alice wants to update a location from the system.

She logs into the system successfully. The system informs Alice that there are no locations.

B.7.3.3. Test story 3: Failed update (duplicated name)

Test objective: Attempt to update a location with a duplicated name (Extension 8a).

Alice wants to update a location from the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice chooses the task “Alice’s home” and informs the system that she wants to update some of its information.

The system provides Alice with further information about the location, like the position on a map with the exact location of the post office and which tasks are related to the position.

Alice changes the name of the location to “Alice’s office”.

The system asks Alice to confirm the modification. She confirms the modification.

The system rejects the update of the location and informs Alice that she already has that same location (i.e. a location with the same).

B.7.4. Delete a location

B.7.4.1. Test story 1: Deletion of a location

Test objective: Delete a location (Main success scenario).

Alice wants to delete a location from the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice chooses the location “Post office” and informs the system that she wants to delete some of its information.

The system asks Alice to confirm the deletion. She confirms the deletion.

The system informs Alice that the removal of the location has been successful, and the list of locations does not include the deleted location anymore.

B.7.4.2. Test story 2: Failed deletion (no locations)

Test objective: Attempt to delete a location when the system has none (Extension 2a).

Alice wants to delete a location from the system.

She logs into the system successfully. The system informs Alice that there are no locations.

B.7.5. Set a default a location

B.7.5.1. Test story 1: Set the default location

Test objective: Set a location as the default one (Main success scenario).

Alice wants to delete a location from the system.

Alice wants to set a default location to the system. She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice chooses the location “Alice’s office” and informs the system that she wants to set it as default.

The system asks Alice to confirm the action. She confirms it.

The system informs Alice that the location “Alice’s office” has been successfully set as the default location.

B.7.5.2. Test story 2: Failed setting of the default location (no locations)

Test objective: Attempt to set a location as the default one when the system has none (Extension 2a).

Alice wants to delete a location from the system.

She logs into the system successfully. The system informs Alice that there are no locations.

B.7.5.3. Test story 3: Failed setting of the default location (already has one)

Test objective: Attempt to set a location as the default one when the system already has one (Extension 2b).

Alice wants to delete a location from the system.

She logs into the system successfully. The system informs Alice that she already has a default location along with its name, in this case “Alice’s office”.

B.7.6. Unset a default a location

B.7.6.1. Test story 1: Unset the default location

Test objective: Unset a location as the default one (Main success scenario).

Alice wants to unset the default location from the system.

She logs into the system successfully. The system provides Alice with a list of her currently existing locations which are “Alice’s home”, “Alice’s office” and “Post office”.

Alice informs the system that she wants to unset its default location. The system asks Alice to confirm the action. She confirms it.

The system informs Alice that there is no longer a default location.

B.7.6.2. Test story 2: Failed unsetting of the default location (does not have one)

Test objective: Attempt to unset the default location when the system does not have any set (Extension 2a).

Alice wants to unset the default location from the system.

She logs into the system successfully. The system informs Alice that she does not have any location set as default.

C. Conceptual schema code

The code presented here is roughly ordered according to the chosen subsets. Nevertheless, its version is final.

C.1. Subset 1: Basic use cases (i)

Algorithm C.1 Account class

```
model RememberTheMilk enum Language {Norwegian, Catalan,
                                     Spanish, English}

class Account attributes
    username: String
    password: String
        creationDate: Integer
    isLoggedIn: Boolean
    email : String
    -- Assoc
        -- Task [*]
        -- ReminderS[*]
        -- LanguageSettings[0..1]
        -- List [*]
        -- List[0..1] //DefaultList
        -- Account[*] //contacts
        -- Group[1] //owner
        -- Group[1] //member
end

context Account inv usernameAccountIdentifier:
    Account.allInstances->isUnique(username)

context Account inv usernameMaxLength:
    Account.allInstances->forAll(a| a.username.size() <= 50
                                 and a.username.size() >= 2)

context Account inv passwordMaxLength:
    Account.allInstances->forAll(a| a.password.size() <= 50
                                 and a.password.size() >= 4)
```

Algorithm C.2 Create account

```
event CreateAccount
  attributes
    username: String
    password: String
    creationDate: Integer
    isLoggedIn: Boolean[0..1]
  operations
    effect()
end

context CreateAccount ini inv usernameAccountIdentifier:
    Account.allInstances()->forAll(a|
                                a.username <> self.username)

context CreateAccount ini inv usernameMaxLength:
    self.username.size() <= 50 and self.username.size() >= 2

context CreateAccount ini inv passwordMaxLength:
    self.password.size() <= 50 and self.password.size() >= 4

context CreateAccount::effect()
  post: Account.allInstances()->exists(a | a.oclIsNew() and
                        a.username = self.username and
                        a.password = self.password and
                        a.creationDate = self.creationDate and
                        a.isLoggedIn = false)
```

Algorithm C.3 Log into account

```
event LogIntoAccount
    attributes
        username: String
        password: String
        isLoggedIn: Boolean[0..1]
    operations
        effect()
end

context LogIntoAccount ini inv usernameAndPasswordMatch:
    Account.allInstances()->exists(a|
                                a.username = self.username and
                                a.password = self.password)

context   LogIntoAccount::effect()
  post:   Account.allInstances()->exists(a |
                                  a.username = self.username and
                                  a.password = self.password and
                                  a.isLoggedIn = true)
```

Algorithm C.4 Task class

```
class Task
  attributes
    descriptor: String
    creationDate: Integer
    dueDate: Integer
    completed: Boolean
    postponedTimes: Integer
    sender: Account[0..1]
    -- Assoc
       --Account[1..*]
       --Note[*]
       --Prio[0..1]
       --List[0..1]
end

context Task inv descriptorMaxLength:
    Task.allInstances->forAll(a| a.descriptor.size() <= 200
                                 and a.descriptor.size() >= 4)

context Task inv taskIdentifier:
    not Account.allInstances->exists(a, b | a <> b and
                a.task.descriptor = b.task.descriptor and
                a.task.dueDate = b.task.dueDate and
                a.username = b.username)

association BelongsToAccount between
    Task[*]
    Account[1..*]
end
```

Algorithm C.5 Create task

```
event CreateTask
    attributes
        user: Account
        descriptor: String
        creationDate: Integer
        dueDate : Integer
    operations
        effect()
end

context CreateTask ini inv descriptorMaxLength:
    self.descriptor.size() <= 200 and
                                    self.descriptor.size() >= 4

context CreateTask ini inv taskIdentifier:
    not Account.allInstances->exists(a |
       a.task.descriptor->includes(self.descriptor) and
       a.task.dueDate->includes(self.dueDate) and
       a.username = self.user.username)

context CreateTask ini inv loggedUser:
    self.user.isLoggedIn = true

context  CreateTask::effect()
  post:  Task.allInstances()->exists(t | t.oclIsNew() and
                        t.descriptor = self.descriptor and
                        t.creationDate = self.creationDate and
                        t.dueDate = self.dueDate and
                        t.completed = false and
                        t.postponedTimes = 0
                        t.account->includes(self.user))
```

Algorithm C.6 Delete task

```
event DeleteTask
    attributes
        user: Account
        descriptor: String
        dueDate : Integer
        confirmation : Boolean
    operations
        effect()
end

context DeleteTask ini inv priorTaskExistence:
    Account.allInstances()->exists(a |
            a.task.descriptor->includes(self.descriptor) and
            a.task.dueDate->includes(self.dueDate) and
            a.username = self.user.username)

context DeleteTask ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteTask ini inv confirmedDeletion:
    self.confirmation = true

context  DeleteTask::effect()
  post:  not Task.allInstances()->exists(t |
                            t.descriptor = self.descriptor and
                            t.dueDate = self.dueDate and
                            t.account->includes(self.user))
```

C.2. Subset 2: Basic use cases (ii)

Algorithm C.7 Update account

```
event UpdateAccount
  attributes
    user: Account
    email: String
  operations
    effect()
end

context UpdateAccount::effect()
  post: Account.allInstances()->exists(a | a = self.user and
                                       a.isLoggedIn = true and
                                       a.email = self.email)

context UpdateAccount ini inv invalidEmail:
    self.email.size() >= 5

context UpdateAccount ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.8 Log out of an account

```
event LogOutAccount
  attributes
    user: Account
  operations
    effect()
end

context LogOutAccount ini inv loggedUser:
    self.user.isLoggedIn = true

context LogOutAccount::effect()
  post: Account.allInstances()->exists(a | a = self.user and
                                     a.isLoggedIn = false)
```

Algorithm C.9 Update task

```
event UpdateTask
  attributes
    user: Account
    descriptorOld: String
    dueDate : Integer
    descriptorNew: String
  operations
    effect()
end

context  UpdateTask::effect()
  post:   Task.allInstances()->exists(t |
                        t.descriptor = self.descriptorNew and
                        t.dueDate = self.dueDate and
                        t.account->includes(self.user))

context UpdateTask ini inv taskIdentifier:
    Account.allInstances->exists(a |
          a.task.descriptor->includes(self.descriptorOld) and
          a.task.dueDate->includes(self.dueDate) and
          a.username = self.user.username)

context UpdateTask ini inv descriptorMaxLength:
    self.descriptorNew.size() <= 200 and
                                self.descriptorNew.size() >= 4
```

Algorithm C.10 Delete account

```
event DeleteAccount
  attributes
    user: Account
    username: String
    password: String
    confirmation: Boolean
  operations
    effect()
end

context DeleteAccount::effect()
  post: not Account.allInstances()->exists(a | a = self.user)

context DeleteAccount ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteAccount ini inv userAndUsernameAndPasswordMatch:
    Account.allInstances()->exists(a|
                                a.username = self.username and
                                a.password = self.password and
                                a = self.user)

context DeleteAccount ini inv confirmedDeletion:
    self.confirmation = true
```

C.3. Subset 3: Performance use cases (i)

Algorithm C.11 Note class

```
class Note
  attributes
    text: String
    -- Assoc
        --Task[1]
end

context Note inv textMinLength:
    Note.allInstances->forAll(a| a.text.size() > 0)

association BelongsToTask between
    Task[1]
    Note[*]
end
```

Algorithm C.12 Create note

```
event CreateNote
  attributes
    user: Account
    text: String
        task: Task
  operations
    effect()
end

context CreateNote ini inv textMinLength:
    self.text.size() > 0

context CreateNote::effect() post:
  Note.allInstances()->exists(n | n.oclIsNew() and
                          n.text = self.text and
                          n.task = self.task and
                          n.task.account->includes(self.user))

context CreateNote ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.13 Update note

```
event UpdateNote
  attributes
    user: Account
    textNew: String
        task: Task
    note: Note
  operations
    effect()
end

context UpdateNote ini inv textMinLength:
    self.textNew.size() > 0

context UpdateNote::effect()
  post: Note.allInstances()->exists(n | n.text = self.textNew
                          and n.task = self.task and
                          n.task.account->includes(self.user))
```

Algorithm C.14 Delete note

```
event DeleteNote
  attributes
    user: Account
        task: Task
    note: Note
        confirmation: Boolean
  operations
    effect()
end

context DeleteNote::effect()
  post: not Note.allInstances()->exists(n |
                          n = self.note and
                          n.task = self.task and
                          n.task.account->includes(self.user))

context DeleteNote ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteNote ini inv confirmedDeletion:
    self.confirmation = true
```

Algorithm C.15 Complete task

```
event CompleteTask
  attributes
        user: Account
        task: Task
    operations
        effect()
end

context CompleteTask::effect()
  post: Task.allInstances()->exists(t |
                                t = self.task and
                                t.completed = true and
                                t.account->includes(self.user))

context CompleteTask ini inv loggedUser:
    self.user.isLoggedIn = true

context CompleteTask ini inv notAlreadyCompleted:
    self.task.completed = false
```

Algorithm C.16 Postpone task

```
event PostponeTask
  attributes
    user: Account
    task: Task
    today: Integer
  operations
    effect()
end

context PostponeTask::effect()
  post: Task.allInstances()->exists(t | t = self.task and
                          t.dueDate >= self.today and
                          t.account->includes(self.user) and
                          t.completed = false and
                          t.postponedTimes > 0)

context PostponeTask ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.17 Duplicate task

```
event DuplicateTask
  attributes
    user: Account
    task: Task
  operations
    effect()
end

context DuplicateTask::effect()
  post: Task.allInstances()->exists(t |
      t.descriptor = self.task.descriptor.concat(' copy') and
      t.account->includes(self.user) and
      t.completed = false)

context DuplicateTask ini inv loggedUser:
    self.user.isLoggedIn = true
```

C.4. Subset 4: Performance use cases (ii)

Algorithm C.18 Uncomplete task

```
event UncompleteTask
  attributes
    user: Account
    task: Task
  operations
    effect()
end

context UncompleteTask::effect()
  post: Task.allInstances()->exists(t | t = self.task and
                                t.completed = false and
                                t.account->includes(self.user))

context UncompleteTask ini inv loggedUser:
    self.user.isLoggedIn = true

context UncompleteTask ini inv alreadyCompleted:
    self.task.completed = true
```

Algorithm C.19 Priority class

```
class Prio
  attributes
    level: Integer
    -- Assoc
       --Task[*]
end

association PrioOfTask between
    Task[*]
    Prio[0..1]
end
```

Algorithm C.20 Create priority

```
event CreatePrio
  attributes
    user: Account
    level: Integer
    task: Task
  operations
    effect()
end

context CreatePrio ini inv noPrioYet:
    self.task.prio->isEmpty()

context CreatePrio::effect()
  post: Prio.allInstances()->exists(p | p.oclIsNew() and
                          p.level = self.level and
                          p.task->includes(self.task) and
                          p.task.account->includes(self.user))


context CreatePrio ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.21 Update priority (i)

```
event UpdatePrio
    attributes
        user: Account
        prioNew: Integer[0..1]
        increase: Boolean[0..1]
        decrease: Boolean[0..1]
        task: Task
    operations
        effect()
    end
```

Algorithm C.22 Update priority (ii)

```
context UpdatePrio::effect()
  post: Prio.allInstances()->exists(p |
    -- first clause
    ((self.increase.isUndefined() and self.increase.isUndefined())
       or (not (self.increase=true) and not (self.increase=true)))
       implies p.level = self.prioNew
    and -- second clause
    ((self.increase=true) and not (self.decrease=true)
       and p.level@pre < 3) -- 3 is the max. prio. level
       implies p.level = p.level@pre +1
    and -- third clause
    ((self.decrease=true) and not (self.increase=true)
       and p.level@pre > 1) -- 1 is the min. prio. level
       implies p.level+1 = p.level@pre
    and -- fourth clause, unconditional
    p.task->includes(self.task)
    and -- fifth clause, unconditional
    p.task.account->includes(self.user))
```

Algorithm C.23 Update priority (iii)

```
context UpdatePrio ini inv alreadyPrio:
    self.task.prio->notEmpty()

context UpdatePrio ini inv loggedUser:
    self.user.isLoggedIn = true

context UpdatePrio ini inv notIncreaseAndDecrease:
    not ((self.increase=true) and (self.decrease=true))

context UpdatePrio ini inv notOutOfUpperBounds:
    Prio.allInstances()->exists(p | ((self.increase=true)
          and not (self.decrease=true) implies p.level < 3)
          and p.task->includes(self.task)
          and p.task.account->includes(self.user))

context UpdatePrio ini inv notOutOfLowerBounds:
    Prio.allInstances()->exists(p | (not (self.increase=true)
                and (self.decrease=true) implies p.level > 1)
                and p.task->includes(self.task) and
                    p.task.account->includes(self.user))
```

Algorithm C.24 Delete priority

```
event DeletePrio
  attributes
    user: Account
    task: Task
    confirmation: Boolean
  operations
    effect()
end

context DeletePrio::effect()
  post: not Prio.allInstances()->exists(p | p.level < 0 and
                          p.task->includes(self.task) and
                          p.task.account->includes(self.user))

context DeletePrio ini inv loggedUser:
    self.user.isLoggedIn = true

context DeletePrio ini inv confirmedDeletion:
    self.confirmation = true

context DeletePrio ini inv alreadyPrio:
    self.task.prio->notEmpty()
```

Algorithm C.25 Reminder schedule class

```
class RemindS
  attributes
    previousTime: Integer [0..1]
    regularTime: Integer [0..1]
    methode: String
    -- Assoc
        --Account[1]
end

association ReminderOfAccount between
    Account[1]
    RemindS[*]
end
```

Algorithm C.26 Create reminder schedule

```
event CreateReminder
  attributes
    user: Account
    methode: String
    previousTime: Integer[0..1]
    regularTime: Integer[0..1]
operations
    effect()
end

context CreateReminder::effect()
  post: RemindS.allInstances()->exists(r | r.oclIsNew() and
                                  r.methode = self.methode and
                                  r.account = self.user)

context CreateReminder ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.27 Update reminder schedule

```
event UpdateReminder
  attributes
    user: Account
    methodeOld: String
    methodeNew: String[0..1]
    previousTime: Integer[0..1]
    regularTime: Integer[0..1]
operations
    effect()
end

context UpdateReminder ini inv alreadyThatReminder:
    RemindS.allInstances()->exists(rs |
                              self.methodeOld = rs.methode and
                              rs.account = self.user)

context UpdateReminder::effect()
  post: RemindS.allInstances()->exists(rs |
                    self.previousTime >= 0 implies
                        rs.previousTime = self.previousTime and
                    self.regularTime >= 0 implies
                        rs.regularTime = self.regularTime and
                    self.methodeNew.size() > 0 implies
                         rs.methode = self.methodeNew and
                    rs.account = self.user)

context UpdateReminder ini inv loggedUser:
    self.user.isLoggedIn = true
```

Algorithm C.28 Delete reminder class

```
event DeleteReminder
  attributes
    user: Account
        rem: RemindS
    confirmation: Boolean
operations
    effect()
end

context DeleteReminder::effect()
  post:  not RemindS.allInstances()->exists(rs |
                                        rs = self.rem and
                                            rs.account = self.user)

context DeleteReminder ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteReminder ini inv confirmedDeletion:
    self.confirmation = true

context DeleteReminder ini inv alreadyReminder:
    RemindS.allInstances()->exists(rs | rs = self.rem and
                                        rs.account = self.user)
```

Algorithm C.29 LanguageS class

```
class LanguageSettings
  attributes
    lang: Language[0..1]
    -- Assoc
        --Account[*]
end


context LanguageSettings inv uniqueLang:
  --singleton pattern applied to each language
    not LanguageSettings.allInstances()->exists(i1, i2 |
                                          i1.lang = i2.lang
                                          and i1 <> i2)

context LanguageSettings inv uniqueLangName:
  --singleton pattern applied to each language
    LanguageSettings.allInstances()->isUnique(lang)

association LanguageOfAccount between
    Account[*]
    LanguageSettings[0..1]
end

class Catalan < LanguageSettings
  attributes
end

class Norwegian < LanguageSettings
  attributes
end

class Spanish < LanguageSettings
  attributes
end

class English < LanguageSettings
  attributes
end
```

Algorithm C.30 Change language

```
event ChangeLanguage
  attributes
    user: Account[0..1]
    lang: Language
operations
    effect()
end

context ChangeLanguage ini inv loggedUserAccount:
    not self.user.isUndefined() implies
                                    (self.user.isLoggedIn=true)

context ChangeLanguage::effect()
  post:  LanguageSettings.allInstances()->exists(s |
                                          s.lang = self.lang)
```

C.5. Subset 5: Performance use cases (iii)

Algorithm C.31 List class

```
class List
  attributes
    name : String
    isEditable : Boolean
    isPublic: Boolean
    -- Assoc
        --Account[*]
        --Task[*]
end

association ListOfAccount between
    Account[1..*]
    List[*]
end

association ListHasTasks between
    List[*]
    Task[*]
end

context List inv uniqueNameUser:
    not List.allInstances()->exists(a, b | a <> b and
            a.name = b.name and
            a.account->intersection(b.account)->notEmpty())

--a task can only be in one list per user
context List inv uniqueTaskListUser:
       not List.allInstances()->exists(la, lb | la <> lb and
           la.task->exists(ta | lb.task->includes(ta)) and
           la.account->exists(aa | lb.account->includes(aa)))
```

Algorithm C.32 Create list

```
event CreateList
  attributes
    user: Account
    name: String
    editable: Boolean
operations
    effect()
end

context CreateList::effect()
  post: List.allInstances()->exists(l | l.oclIsNew() and
                          l.name = self.name and
                          l.account->includes(self.user) and
                          l.isEditable = self.editable)

context CreateList ini inv loggedUser:
    self.user.isLoggedIn = true

context CreateList ini inv uniqueNameUser:
    not List.allInstances()->exists(a | a.name = self.name
                          and a.account->includes(self.user)
```

Algorithm C.33 Update list

```
event UpdateList
  attributes
    user: Account
    nameOld: String
    nameNew: String
    confirmation: Boolean
operations
    effect()
end

context UpdateList::effect()
  post: List.allInstances()->exists(l |
                              l.name = self.nameNew and
                              l.account->includes(self.user))

context UpdateList ini inv loggedUser:
    self.user.isLoggedIn = true

context UpdateList ini inv confirmed:
    self.confirmation = true

context UpdateList ini inv editable:
    List.allInstances()->exists(a | a.name = self.nameOld
                           and a.account->includes(self.user)
                           and a.isEditable=true)

context UpdateList ini inv oldListExists:
    List.allInstances()->exists(a |
                             a.name = self.nameOld and
                              a.account->includes(self.user))

context UpdateList ini inv newListDoesNotExist:
    not List.allInstances()->exists(a |
                              a.name = self.nameNew and
                              a.account->includes(self.user))
```

Algorithm C.34 Delete list

```
event DeleteList
  attributes
    user: Account
    name: String
    confirmation: Boolean
operations
   effect()
end

context DeleteList::effect()
  post: not List.allInstances()->exists(l |
                          l.name = self.name and
                          l.account->includes(self.user) and
                          l.isEditable = true)

context DeleteList ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteList ini inv confirmed:
    self.confirmation = true

context DeleteList ini inv existsEmptyAndEditable:
    List.allInstances()->exists(a | a.name = self.name and
                          a.account->includes(self.user) and
                          a.isEditable=true and
                          a.task->size() = 0)
```

Algorithm C.35 Set default list

```
association DefaultListOfAccount between
    Account[0..1] role ownerAccount
    List[0..1] role defaultList
end

event SetDefaultList
  attributes
    user: Account
    list: List
operations
    effect()
end

context SetDefaultList::effect()
  post: Account.allInstances()->exists(a | a = self.user and
                                    a.defaultList = self.list)

context SetDefaultList ini inv loggedUser:
    self.user.isLoggedIn = true

context SetDefaultList ini inv noDefaultListYet:
    not Account.allInstances()->exists(a | a = self.user and
                              a.defaultList.isDefined() = true)

context SetDefaultList ini inv listOwnedByUser:
    List.allInstances()->exists(a | a = self.list and
                                a.account->includes(self.user))
```

Algorithm C.36 Unset default list

```
event UnsetDefaultList
attributes
    user: Account
operations
    effect()
end

context UnsetDefaultList::effect()
  post: not Account.allInstances()->exists(a | a = self.user
                                and a.defaultList.isDefined())

context UnsetDefaultList ini inv loggedUser:
    self.user.isLoggedIn = true

context UnsetDefaultList ini inv alreadyDefaultList:
Account.allInstances()->exists(a | a = self.user and
                            a.defaultList.isDefined() = true)
```

Algorithm C.37 Move task to list

```
event MoveTaskToList
        attributes
            user: Account
        list: List
        task: Task
        operations
            effect()
        end

context  MoveTaskToList::effect()
  post: Task.allInstances()->exists(t | t = self.task and
                          t.account->includes(self.user) and
                          t.list->includes(self.list))

context MoveTaskToList ini inv existsListAndEditable:
    List.allInstances()->exists(a | a = self.list and
a.isEditable=true)

context MoveTaskToList ini inv existsTask:
    Task.allInstances()->exists(t | t = self.task)

context MoveTaskToList ini inv loggedUser:
    self.user.isLoggedIn = true

context MoveTaskToList ini inv ownsList:
    self.list.account->includes(self.user)

context MoveTaskToList ini inv ownsTask:
    self.task.account->includes(self.user)

context MoveTaskToList ini inv fromEditableList:
--if the list where the task belongs for that user (owner)
--is defined then the list must be editable
    self.task.list->any(l |
          l.account->includes(self.user)).isDefined() implies
              self.task.list->any(l |
                          l.account->includes(self.user)).
                                  isEditable=true
context MoveTaskToList ini inv toEditableList:
    self.list.isEditable = true
```

Algorithm C.38 Add contact

```
association AreContacts between
    Account[*] role contactor
    Account[*] role contacted
end

event AddContact
  attributes
    user: Account
    contact: Account
operations
    effect()
end

context  AddContact::effect()
  post: Account.allInstances()->exists(a, b |
                                    a = self.user and
                                    b = self.contact and
                                    a.contacted->includes(b))

context AddContact ini inv loggedUser:
    self.user.isLoggedIn = true

context AddContact ini inv notAlreadyAContact:
    not Account.allInstances()->exists(a | a = self.user and
                          a.contacted->includes(self.contact))

context AddContact ini inv reciprocity:
    Account.allInstances()->forAll(a, b | a <> b and
                              a.contacted->includes(b) implies
                                  b.contactor->includes(a))
```

Algorithm C.39 Delete contact

```
event DeleteContact
  attributes
    user: Account
    contact: Account
operations
    effect()
end

context DeleteContact::effect()
  post: not Account.allInstances()->exists(a, b |
                                    a = self.user and
                                    b = self.contact and
                                    a.contacted->includes(b))

context DeleteContact ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteContact ini inv beenContact:
    Account.allInstances()->exists(a | a = self.user and
                          a.contacted->includes(self.contact))

context DeleteContact ini inv reciprocity:
    Account.allInstances()->forAll(a, b | a <> b and
                               a.contacted->includes(b) implies
                               b.contactor->includes(a))
```

C.6. Subset 6: Excitement use cases (i)

Algorithm C.40 Publish list for anyone

```
event PublishListForAnyone
  attributes
    user: Account
    list: List
operations
    effect()
end

context PublishListForAnyone::effect()
  post: List.allInstances()->exists(l | l = self.list and
                                l.isPublic = true and
                                l.account->includes(self.user))

        context PublishListForAnyone ini inv loggedUser:
            self.user.isLoggedIn = true

    context PublishListForAnyone ini inv listOwnedByUser:
            self.list.account->includes(self.user)

context PublishListForAnyone ini inv listNotAlreadyPublic:
    not List.allInstances()->exists(l | l = self.list and
l.isPublic = true and
                                l.account->includes(self.user))
```

Algorithm C.41 Unpublish list for anyone

```
event UnpublishListForAnyone
  attributes
    user: Account
    list: List
operations
    effect()
end

context UnpublishListForAnyone::effect()
  post: not List.allInstances()->exists(l |
                                l = self.list and
                                l.isPublic = true and
                                l.account->includes(self.user))

context UnpublishListForAnyone ini inv loggedUser:
    self.user.isLoggedIn = true

context UnpublishListForAnyone ini inv listOwnedByUser:
    self.list.account->includes(self.user)

context UnpublishListForAnyone ini inv listAlreadyPublic:
    List.allInstances()->exists(l | l = self.list and
                               l.isPublic = true and
                               l.account->includes(self.user))
```

Algorithm C.42 Send task to contact

```
event SendTaskToContact
  attributes
    user: Account
    contact: Account[1..*]
    task: Task
operations
    effect()
end

context SendTaskToContact::effect()
  post: Task.allInstances()->exists(t | t = self.task and
        self.contact->forAll(b | t.account->includes(b)) and
        t.sender = self.user and
        t.completed = false)
context SendTaskToContact ini inv loggedUser:
    self.user.isLoggedIn = true

context SendTaskToContact ini inv notAlreadyCompleted:
    self.task.completed = false

context SendTaskToContact ini inv taskInEditableList:
    self.task.list->select(l |
         l.account->includes(self.user))->notEmpty() implies
             self.task.list->any(l |
                              l.account->includes(self.user)).
                                                  isEditable=true

context SendTaskToContact ini inv contactsOfUser:
    self.contact->forAll(b |
                            self.user.contacted->includes(b))
```

Algorithm C.43 Publish list for contacts

```
association CanBeReadBy between
    List[*] role listRO
    Account[*] role reader
end

event PublishListForContact
  attributes
    user: Account
    contact: Account[1..*]
    list: List
operations
    effect()
end

context PublishListForContact::effect()
  post: List.allInstances()->exists(l | l = self.list and
            l.account->includes(self.user) and
            self.contact->forAll(b | l.reader->includes(b)))

context PublishListForContact ini inv loggedUser:
    self.user.isLoggedIn = true

context PublishListForContact ini inv
                        listNotAlreadyPublishedForSomeOfThem:
    not List.allInstances()->exists(l | l = self.list and
            l.account->includes(self.user) and
           self.contact->forAll(b | l.reader->includes(b)))
context PublishListForContact ini inv listOwnedByUser:
    self.list.account->includes(self.user)

context PublishListForContact ini inv contactsOfUser:
    self.contact->forAll(b| self.user.contacted->includes(b))
```

Algorithm C.44 Unpublish list for contacts

```
event UnpublishListForContact
  attributes
    user: Account
    contact: Account[1..*]
    list: List
operations
    effect()
end

context UnpublishListForContact::effect()
  post: not List.allInstances()->exists(l |
            l = self.list and
            l.account->includes(self.user) and
            self.contact->forAll(b | l.reader->includes(b)))

context UnpublishListForContact ini inv loggedUser:
    self.user.isLoggedIn = true

context UnpublishListForContact ini inv
                              listAlreadyPublishedForSomeOfThem:
    List.allInstances()->exists(l | l = self.list and
            l.account->includes(self.user) and
            self.contact->forAll(b | l.reader->includes(b)))

context UnpublishListForContact ini inv listOwnedByUser:
    self.list.account->includes(self.user)

context UnpublishListForContact ini inv contactsOfUser:
    self.contact->forAll(b| self.user.contacted->includes(b))
```

Algorithm C.45 Share list with contacts

```
association AcceptancePending between
    List[*] role listAP
    Account[*] role candidate
end

event ShareListWithContact
  attributes
    user: Account
    contact: Account[1..*]
    list: List
operations
    effect()
end

context ShareListWithContact::effect()
  post: List.allInstances()->exists(l | l = self.list and
          l.account->includes(self.user) and
          self.contact->forAll(b | l.candidate->includes(b)))

context ShareListWithContact ini inv loggedUser:
    self.user.isLoggedIn = true

context ShareListWithContact ini inv
                            listNotAlreadySharedForSomeOfThem:
    not List.allInstances()->exists(l | l = self.list and
           l.account->includes(self.user) and
          self.contact->forAll(b | l.account->includes(b) or
l.candidate->includes(b)))

context ShareListWithContact ini inv listOwnedByUser:
   self.list.account->includes(self.user)

context ShareListWithContact ini inv editableList:
    self.list.isEditable = true

context ShareListWithContact ini inv contactsOfUser:
    self.contact->forAll(b |
self.user.contacted->includes(b))
```

Algorithm C.46 Accept shared list

```
event AcceptSharedList
  attributes
    user: Account
    list: List
operations
    effect()
end

context AcceptSharedList::effect()
  post: List.allInstances()->exists(l | l = self.list and
                          l.account->includes(self.user) and
                          not l.candidate->includes(self.user))

context AcceptSharedList ini inv loggedUser:
    self.user.isLoggedIn = true

context AcceptSharedList ini inv listPendingAcceptance:
List.allInstances()->exists(l | l = self.list and
                            l.candidate->includes(self.user))
```

Algorithm C.47 Reject shared list

```
event RejectSharedList
  attributes
    user: Account
    list: List
operations
    effect()
end

context RejectSharedList::effect()
  post: List.allInstances()->exists(l | l = self.list and
                        not l.account->includes(self.user) and
                        not l.candidate->includes(self.user))

context RejectSharedList ini inv loggedUser:
    self.user.isLoggedIn = true

context RejectSharedList ini inv listPendingAcceptance:
    List.allInstances()->exists(l | l = self.list and
                              l.candidate->includes(self.user))
```

Algorithm C.48 Share task with contacts

```
event ShareTaskWithContact
attributes
    user: Account
    contact: Account[1..*]
    task: Task
operations
    effect()
end

context ShareTaskWithContact::effect()
  post: Task.allInstances()->exists(t | t = self.task and
            t.account->includes(self.user) and
            self.contact->forAll(b | t.account->includes(b)))

context ShareTaskWithContact ini inv loggedUser:
    self.user.isLoggedIn = true

context ShareTaskWithContact ini inv
                              taskNotAlreadySharedForSomeOfThem:
    not Task.allInstances()->exists(t | t = self.task and
            t.account->includes(self.user) and
            self.contact->forAll(b | t.account->includes(b)))

context ShareTaskWithContact ini inv taskOwnedByUser:
    self.task.account->includes(self.user)

context ShareTaskWithContact ini inv uncompleted:
    self.task.completed = false

context ShareTaskWithContact ini inv contactsOfUser:
    self.contact->forAll(b| self.user.contacted->includes(b))
```

Algorithm C.49 Update account

```
context  UpdateAccount::effect()
  post: Account.allInstances()->exists(a | a = self.user and
                                      a.isLoggedIn = true and
                                      a.email = self.email)
```

C.7. Subset 7: Excitement use cases (ii)

Algorithm C.50 Group class

```
class Group
attributes
    subject : String
    -- Assoc
        --Account[1] owner
            --Account[*] member
end

association isOwned between
    Account[1] role owner
    Group[*] role ownedGroup
end

association belongsToGroup between
    Account[*] role member
    Group[*] role memberGroup
end

context Group inv uniqueSubjectUser:
    not Group.allInstances()->exists(a, b | a <> b and
                                    a.subject = b.subject and
a.owner = b.owner)

context Group inv membersAreUserContacts:
    Group.allInstances()->forAll(g |
                            g.member->forAll(m |
                                g.owner.contacted->includes(m)))
```

Algorithm C.51 Create group

```
event CreateGroup
attributes
    user: Account
    subject: String
operations
    effect()
end

context CreateGroup::effect()
  post: Group.allInstances()->exists(g | g.oclIsNew() and
                                g.subject = self.subject and
                                g.owner = self.user)

context CreateGroup ini inv loggedUser:
    self.user.isLoggedIn = true

context CreateGroup ini inv uniqueGroupSubject:
    not Group.allInstances()->exists(g |
                                g.subject = self.subject and
                                g.owner = self.user)
```

Algorithm C.52 Delete group

```
event DeleteGroup
attributes
    user: Account
    subject: String
    confirmation: Boolean
operations
    effect()
end

context DeleteGroup::effect()
  post: not Group.allInstances()->exists(g |
                                  g.subject = self.subject and
                                          g.owner = self.user)

context DeleteGroup ini inv loggedUser:
    self.user.isLoggedIn = true

context DeleteGroup ini inv confirmed:
    self.confirmation = true

context DeleteGroup ini inv existsEmptyGroupOwnedByUser:
    Group.allInstances()->exists(g |
                                  g.subject = self.subject and
                                  g.owner = self.user and
                                  g.member->size() = 0)
```

D. Methods code

D.1. Subset 1: Basic use cases (i)

Algorithm D.1 Create account

```
method CreateAccount{
    res := new Account(username := self.username,
                       password := self.password,
                       creationDate := self.creationDate,
                       isLoggedIn := false,
                       email := '');
}
```

Algorithm D.2 Log into account

```
method LogIntoAccount{
    li := [Account.allInstances()->any(acc |
                       acc.username = self.username and
                       acc.password = self.password)];
    li.isLoggedIn := true;
}
```

Algorithm D.3 Create task

```
method CreateTask{
    res := new Task(descriptor := self.descriptor,
                    creationDate := self.creationDate,
                    dueDate := self.dueDate);
    res.account := self.user;
    res.completed := false;
    res.postponedTimes := 0;
    if self.list.isDefined() then
        res.list := self.list;
    endif
}
```

Algorithm D.4 Delete task

```
method DeleteTask{
    dt := [Task.allInstances()->any(t |
                    t.descriptor = self.descriptor and
                    t.dueDate = self.dueDate and
                    t.account->includes(self.user))];
    self.user.task := self.user.task->excluding(dt);
    if [dt.account->size() = 0] then
        delete dt;
    endif
}
```

D.2. Subset 2: Basic use cases (ii)

Algorithm D.5 Update account

```
method UpdateAccount{
    li := [Account.allInstances()->any(acc |
                acc = self.user and acc.isLoggedIn = true)];
    li.email := self.email;
}
```

Algorithm D.6 Log out of account

```
method LogOutAccount{
    li := [Account.allInstances()->any(acc |
                                        acc = self.user)];
    li.isLoggedIn := false;
}
```

Algorithm D.7 Update task

```
method UpdateTask{
    ut := [Task.allInstances()->any(t |
                        t.descriptor = self.descriptorOld and
                        t.dueDate = self.dueDate and
                        t.account->includes(self.user))];
    ut.descriptor := self.descriptorNew;
}
```

Algorithm D.8 Delete account

```
method DeleteAccount{
    ac := [Account.allInstances()->any(acc |
                acc = self.user and acc.isLoggedIn = true)];
    ac.task := '';
    while [Task.allInstances()->exists(t |
                                    t.account->size() = 0)] do
        delete [Task.allInstances()->any(t |
                                    t.account->size() = 0)];
    endwhile
    delete ac;
}
```

D.3. Subset 3: Performance use cases (i)

Algorithm D.9 Create note

```
method CreateNote{
    res := new Note(text := self.text);
    res.task := self.task;
}
```

Algorithm D.10 Update note

```
method UpdateNote{
    res := [Note.allInstances()->any(n | n = self.note)];
    res.text := self.textNew;
}
```

Algorithm D.11 Delete note

```
method DeleteNote{
    res := [Note.allInstances()->any(n | n = self.note)];
    delete res;
}
```

Algorithm D.12 Delete task postconditions

```
method CompleteTask{
    res := [Task.allInstances()->any(t | t = self.task)];
    res.completed := true;
}
```

Algorithm D.13 Postpone task

```
method PostponeTask{
    res := [Task.allInstances()->any(t | t = self.task)];
    res.postponedTimes := [res.postponedTimes +1];
    if [res.dueDate < self.today] then
        res.dueDate := self.today;
    else
        res.dueDate := [res.dueDate +1];
    endif
}
```

Algorithm D.14 Duplicate task

```
method DuplicateTask{
    tmp := [Task.allInstances()->any(t | t = self.task)];
    res := new Task(
              descriptor := [tmp.descriptor.concat(' copy')],
              creationDate := tmp.creationDate,
              dueDate := tmp.dueDate);
    res.account := [tmp.account];
    res.postponedTimes := tmp.postponedTimes;
    res.completed := false;
    if [tmp.completed = true] then
        res.dueDate := 0;
    endif
}
```

D.4. Subset 4: Performance use cases (ii)

Algorithm D.15 Uncomplete task

```
method UncompleteTask{
    res := [Task.allInstances()->any(t | t = self.task)];
    res.completed := false;
}
```

Algorithm D.16 Create priority

```
method CreatePrio{
    res := new Prio(level := self.level);
    res.task := self.task;
}
```

Algorithm D.17 Update priority

```
method UpdatePrio{
    res := [Prio.allInstances()->any(p|
                                p.task->includes(self.task))];
    if self.increase.isUndefined() then
        self.increase := false;
    endif
    if self.decrease.isUndefined() then
        self.decrease := false;
    endif
    if [(self.increase xor self.decrease)] then
            if [(self.increase=true) and (res.level<3)] then
            res.level := [res.level +1];
        endif
    endif
    if [(self.increase xor self.decrease)] then
        if [(self.decrease=true) and (res.level>1)] then
            res.level := [res.level -1];
        endif
    endif
    if [not (self.increase xor self.decrease)] then
        res.level := self.prioNew;
    endif

    res.task := res.task->including(self.task);
}
```

Algorithm D.18 Delete priority

```
method DeletePrio{
    res := [Prio.allInstances()->any(p |
                                p.task->includes(self.task))];
    delete res;
}
```

Algorithm D.19 Create reminder

```
method CreateReminder{
    res := new RemindS(methode := self.methode);
    res.account := self.user;
    if self.previousTime>0 then
        res.previousTime := self.previousTime;
    endif
    if self.regularTime>0 then
        res.regularTime := self.regularTime;
    endif
}
```

Algorithm D.20 Update reminder

```
method UpdateReminder{

    res := [RemindS.allInstances()->any(rs |
                            rs.methode = self.methodeOld and
                            rs.account = self.user)];
        if self.previousTime>=0 then
        res.previousTime := self.previousTime;
    endif
    if self.regularTime>=0 then
        res.regularTime := self.regularTime;
    endif
    if self.methodeNew.size()>0 then
        res.methode := self.methodeNew;
    endif
}
```

Algorithm D.21 Delete reminder

```
method DeleteReminder{
    res := [RemindS.allInstances()->any(rs |
                                        rs = self.rem and
                                        rs.account = self.user)];
    delete res;
}
```

Algorithm D.22 Change language (i)

```
method ChangeLanguage{
  if [(not self.user.isUndefined())] then
    idiom := [LanguageSettings.allInstances()->any(i |
                              i.account->includes(self.user))];
      if [idiom.isUndefined()=false] then
        idiom.account := idiom.account->excluding(self.user);
      endif

      if [(self.lang=#Catalan)] then
        if [not (Catalan.allInstances()->exists(i |
                                    i.lang=self.lang))] then
          id := new Catalan(lang := self.lang);
          id.account := id.account->including(self.user);
        endif
        if [(Catalan.allInstances()->exists(i |
                                    i.lang=self.lang))] then
          id := [LanguageSettings.allInstances()->any(i |
                                        i.lang=self.lang)];
          id.account := id.account->including(self.user);
        endif
      endif
//the same code for English and Spanish as well

      if [(self.lang=#Norwegian)] then
        if [not (Norwegian.allInstances()->exists(i |
                                    i.lang=self.lang))] then
          id := new Norwegian(lang := self.lang);
          id.account := id.account->including(self.user);
        endif
        if [(Norwegian.allInstances()->exists(i |
                                    i.lang=self.lang))] then
          id := [LanguageSettings.allInstances()->any(i |
                                        i.lang=self.lang)];
          id.account := id.account->including(self.user);
        endif
      endif
    endif
```

Algorithm D.23 Change language (ii)

```
  if [(self.lang=#Catalan)] then
    if [not (Catalan.allInstances()->exists(i |
                                        i.lang=self.lang))] then
      id := new Catalan(lang := self.lang);
    endif
    if [(Catalan.allInstances()->exists(i |
                                        i.lang=self.lang))] then
      id := [LanguageSettings.allInstances()->any(i |
                                            i.lang=self.lang)];
    endif
  endif
//the same code for English and Spanish as well

  if [(self.lang=#Norwegian)] then
    if [not (Norwegian.allInstances()->exists(i |
                                        i.lang=self.lang))] then
      id := new Norwegian(lang := self.lang);
    endif
    if [(Norwegian.allInstances()->exists(i |
                                        i.lang=self.lang))] then
      id := [LanguageSettings.allInstances()->any(i |
                                            i.lang=self.lang)];
    endif
  endif
```

D.5. Subset 5: Performance use cases (iii)

Algorithm D.24 Create list

```
method CreateList{
    if [not (List.allInstances()->exists(li |
                    li.name = self.name and
                    li.account->includes(self.user)))] then
        res := new List(name := self.name);
        res.isEditable := self.editable;
        res.account := res.account->including(self.user);
        res.isPublic := false;
    endif
}
```

Algorithm D.25 Update list

```
method UpdateList{
    res := [List.allInstances()->any(li |
                        li.name = self.nameOld and
                        li.account->includes(self.user) and
                        li.isEditable=true)];
    res.name := self.nameNew;
}
```

Algorithm D.26 Delete list

```
method DeleteList{
    res := [List.allInstances()->any(li |
                        li.name = self.name and
                       li.account->includes(self.user) and
li.isEditable = true)];
    delete res;
}
```

Algorithm D.27 Move task to list

```
method MoveTaskToList{
    if [self.list.task->includes(self.task)=false] then
        self.task.list := self.list;
    endif
}
```

Algorithm D.28 Set default list

```
method SetDefaultList{
    a := [Account.allInstances()->any(a | a = self.user)];
    a.defaultList := self.list;
}
```

Algorithm D.29 Unset default list

```
method UnsetDefaultList{
    a := [Account.allInstances()->any(a | a = self.user)];
    delete a.defaultList;
}
```

Algorithm D.30 Add contact

```
method AddContact{
    a := [Account.allInstances()->any(a | a = self.user)];
    b := [Account.allInstances()->any(b | b = self.contact)];
    a.contacted := a.contacted->including(b);
}
```

Algorithm D.31 Delete contact

```
method DeleteContact{
    a := [Account.allInstances()->any(a | a = self.user)];
    b := [Account.allInstances()->any(b | b = self.contact)];
    a.contacted := a.contacted->excluding(b);
}
```

D.6. Subset 6: Excitement use cases (i)

Algorithm D.32 Publish list for anyone

```
method PublishListForAnyone{
    res := [List.allInstances()->any(l | l = self.list and
                                l.account->includes(self.user))];
    res.isPublic := true;
}
```

Algorithm D.33 Unpublish list for anyone

```
method UnpublishListForAnyone{
    res := [List.allInstances()->any(l | l = self.list and
                                l.isPublic = true and
                                l.account->includes(self.user))];
    res.isPublic := false;
}
```

Algorithm D.34 Send task to contact

```
method SendTaskToContact{
    taskK := [Task.allInstances()->any(t | t = self.task)];
    taskK.sender := self.user;
    if taskK.completed.isUndefined()=true then
        taskK.completed := false;
    endif

    Integer index := 1;
    while self.contact->size()>=index do
        taskK.account := taskK.account->including(
                        self.contact->asSequence()->at(index));
        index := index+1;
    endwhile

    user := [Account.allInstances()->any(a | a = self.user)];
    listT := [List.allInstances()->any(l |
                                l.account->includes(user) and
                                l.task->includes(taskK))];
    listSent := [List.allInstances()->any(l |
            l.account->includes(user) and l.name = 'Sent')];

    userListsAssignedToTask := [user.task->any(t |
                                            t=taskK).list];
    userListsAssignedToTask := [user.task->any(t |
                        t.descriptor=taskK.descriptor).list->
                            excluding(listT)];
    userListsAssignedToTask := [user.task->any(t |
                         t=taskK).list->including(listSent)];
}
```

Algorithm D.35 Publish list for contacts

```
method PublishListForContact{
    res := [List.allInstances()->any(l | l = self.list and
                                l.account->includes(self.user))];
    index := 1;
    while self.contact->size()>=index do
        self.list.reader := self.list.reader->including(
                        self.contact->asSequence()->at(index));
        index := index+1;
    endwhile
}
```

Algorithm D.36 Unpublish list for contacts

```
method UnpublishListForContact{
    res := [List.allInstances()->any(l | l = self.list and
                                l.account->includes(self.user))];
    index := 1;
    while self.contact->size()>=index do
        self.list.reader := self.list.reader->excluding(
                        self.contact->asSequence()->at(index));
        index := index+1;
    endwhile
}
```

Algorithm D.37 Share list with contacts

```
method ShareListWithContact{
    res := [List.allInstances()->any(l | l = self.list and
                                l.account->includes(self.user))];
    index := 1;
    while self.contact->size()>=index do
        self.list.candidate := self.list.candidate->including(
                        self.contact->asSequence()->at(index));
        index := index+1;
    endwhile
}
```

Algorithm D.38 Accept shared list

```
method AcceptSharedList{
    res := [List.allInstances()->any(l | l = self.list and
                                l.candidate->includes(self.user))];
    res.account := res.account->including(self.user);
    res.candidate := res.candidate->excluding(self.user);
}
```

Algorithm D.39 Reject shared list

```
method RejectSharedList{
    res := [List.allInstances()->any(l | l = self.list and
                                l.candidate->includes(self.user))];
    res.candidate := res.candidate->excluding(self.user);
}
```

Algorithm D.40 Share task with contacts

```
method ShareTaskWithContact{
    res := [Task.allInstances()->any(t | t = self.task and
                                  t.account->includes(self.user))];
    index := 1;
    while self.contact->size()>=index do
        self.task.account := self.task.account->including(
                        self.contact->asSequence()->at(index));
        index := index+1;
    endwhile
}
```

D.7. Subset 7: Excitement use cases (ii)

Algorithm D.41 Create group

```
method CreateGroup{
    res := new Group(subject := self.subject);
    res.owner := self.user;
}
```

Algorithm D.42 Delete group

```
method DeleteGroup{
    res := [Group.allInstances()->any(g |
                                      g.subject = self.subject and
                                     g.owner = self.user)];
    delete res;
}
```

Bibliography

[1] Brandt, D. Randall, "How service marketers can identify value-enhancing service elements", Journal of Services Marketing 2 (3), pp. 35–41 (1988). Available from *http://www.emeraldinsight.com/journals.htm?articleid=1652453&show=abstract*

[2] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, Vol. 20, No. 4, pp. 10-19 (1987). Available from *http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html*

[3] Brooks, Frederick P., "The Mythical Man-Month", Addisson-Wesley, (1975)

[4] Cadotte, E. R., Turgeon, N., "Dissatisfiers and satisfiers: suggestions from consumer complaints and compliments", Journal of Consumer Satisfaction, Dissatisfaction and Complaining Behavior 1, pp. 74–79 (1988). Available from *http://lilt.ilstu.edu/staylor/csdcb/articles/Volume1/Cadotte%20et%20al%201988.pdf*

[5] Case, K. E., Fair, R. C., "Principles of Economics" (5th ed.), Prentice-Hall (1999).

[6] Covey S., Merrill A. R., Merrill R. R., "First Things First: To Live, to Love, to Learn, to Leave a Legacy", Simon and Schuster, (1994)

[7] Griethuysen, J.J. van, "Concepts and terminology for the conceptual schema and the information base", ISO TC97/SC5/WG3 (1982)

[8] Gogolla, M., Bohling, J., Richters, M. "Validating UML and OCL Models in USE by Automatic Snapshot Generation", Software & Systems Modeling, 4(4), pp. 386-398 (2005)

[9] Hevner, A. R., March, S.T., Ram S., "Design Science in Information Systems Research", Research MIS Quarterly 28(1), pp. 75-105 (2004)

[10] IEEE. Standard for Software Verification and Validation, Std 1012-1998b (1998).

[11] Kano, N., Nobuhiku S., Fumio T., Shinichi T., "Attractive quality and must-be quality", Journal of the Japanese Society for Quality Control 14 (2) pp. 39-48 (1984). Available from *http://ci.nii.ac.jp/Detail/detail.do?LOCALID=ART0003570680&lang=en*

[12] Krogstie J., et al. "Integrating the understanding of quality in requirements specification and conceptual modeling", ACM SIGSOFT, Vol. 23 No. 1, pp. 86-91 (1998)

[13] Lindland, O.I., Sindre, G., Solvberg, A., "Understanding Quality in Conceptual Modeling", IEEE Software, 11(2), pp. 42-49 (1994)

[14] Silver, M., Markus, M.L., Beath C.M., "The Information Technology Interaction Model: A Foundation for the MBA Core Course", MIS Quarterly, Vol. 19, No. 3, pp. 361-390 (1995)

[15] Olivé, A., Cabot J., "A Research Agenda for Conceptual Schema-Centric Development", Conceptual modeling in Information Systems Engineering, Springer Verlag, pp. 319-334. Available from *http://jordicabot.com/papers/OliveCabotResearchAgenda.pdf*

[16] Olivé, A., "Conceptual Modeling of Information Systems", Springer, (2007)

[17] Ramírez, A. "Esquema Conceptual De Magento, Un Sistema De Comerç Electrònic", UPC (2011). Available from http://hdl.handle.net/2099.1/12294.

[18] Tort, A., "Development of the Conceptual Schema of a Bowling Game System by Applying TDCM", UPC (2011). Available from *http://hdl.handle.net/2117/11196*

[19] Tort, A., "Development of the Conceptual Schema of the osTicket System by Applying TDCM", UPC (2011). Available from *http://hdl.handle.net/2117/12369*

[20] Tort, A., Olivé, A., "An approach to testing conceptual schemas", Data & Knowledge Engineering, 69(6), pp. 598-618 (2010)

[21] Tort, A., Olivé, A., Sancho, M. R., "An Approach to Test-Driven Development of Conceptual Schemas", Data & Knowledge Engineering, 69(6), pp. 598-618 (2011)

[22] Tort, A., Olivé, A., Sancho, M. R., "The CSTL Processor: A Tool for Automated Conceptual Schema Testing", (2011). Available from *http://www.springerlink.com/content/q22162u475671614/fulltext.pdf*

[23] Venkitaraman, R.K, Jaworski, C., "Restructuring customer satisfaction measurement for better resource allocation decisions: an integrated approach", Fourth Annual Advanced Research Techniques Forum of the American Marketing Association, (1993)