



IT2901 INFORMATICS PROJECT II

---

# LUMA

---

Christian FORFANG  
Einar UVSLØKK  
Johannes HARESTAD

Per Ove RINGDAL  
Simen NATVIG  
Sondre FRISVOLD

May 15, 2011

## **Abstract**

This document is the project report established in course IT2901, Informatics Project II at Norwegian University of Science and Technology, during the spring of 2011. The assigned task was to develop a new and improved version of the Luma LDAP client. The given Luma client, version 2.4, has several weaknesses that are addressed and improved in the new version of Luma, version 3.0, developed in this project. Luma 2.4 is using the deprecated Qt 3 application framework. In Luma 3.0 this framework is replaced with the Qt 4 application framework, and implemented utilizing the Model View Controller architectural design pattern. The user interface is also significantly improved and adapted to the common practices in modern applications. Another important result is that Luma 3.0 is cross platform, with support for Windows/Linux/Unix/Mac OS.

This report defines the requirements to the new version; describes the work that is carried out; and documents the design and implementation decisions that are taken.

Luma 3.0 satisfies all the requirements. This report along with the source code presents a solution to the problem of this assignment.

# Contents

<b>Contents</b>	<b>7</b>
<b>List of Figures</b>	<b>9</b>
<b>List of Tables</b>	<b>11</b>
<b>1 Introduction</b>	<b>11</b>
1.1 Project purpose . . . . .	11
1.2 Challenges and problems . . . . .	11
1.3 The customer . . . . .	11
1.4 The group . . . . .	12
1.4.1 Group structure . . . . .	13
Project Manager . . . . .	13
Quality Assurance Control . . . . .	13
Testing . . . . .	13
Back-end and LDAP . . . . .	13
UI and Repository . . . . .	14
Report and Documentation . . . . .	14
1.5 Structure of the report . . . . .	15
<b>2 Pre studies</b>	<b>16</b>
2.1 Technologies . . . . .	16
2.2 Development methods . . . . .	17
2.2.1 Alternative development methods . . . . .	17
2.2.2 Our development method . . . . .	18
2.3 Libraries and Tools . . . . .	18
2.4 Alternative solutions . . . . .	20
2.4.1 JXplorer . . . . .	21
2.4.2 Argonne's LDAP browser/editor . . . . .	21
2.4.3 LDAP Browser from Symlabs . . . . .	21
2.4.4 Summary . . . . .	22
<b>3 Problems with Luma 2.4</b>	<b>23</b>
3.1 Problems . . . . .	23
3.2 Bugs . . . . .	24
3.3 Shortcomings . . . . .	25
3.4 Plugin system . . . . .	26
3.4.1 Plugin implementation . . . . .	27
3.4.2 Plugin system implementation . . . . .	27

<b>4</b>	<b>Requirements</b>	<b>29</b>
4.1	Functional requirements . . . . .	29
4.2	Non-functional requirements . . . . .	30
4.3	Requirements summary . . . . .	31
4.4	Use cases . . . . .	31
<b>5</b>	<b>Planning</b>	<b>37</b>
5.1	Milestones . . . . .	37
5.2	Risk analysis . . . . .	37
5.3	Architecture . . . . .	38
5.4	Product backlog . . . . .	38
5.5	Gantt and Task list . . . . .	41
<b>6</b>	<b>Sprint overviews</b>	<b>44</b>
6.1	Sprint 1 . . . . .	44
6.1.1	Review . . . . .	45
6.2	Sprint 2 . . . . .	46
6.2.1	Review . . . . .	47
6.3	Sprint 3 . . . . .	48
6.3.1	Review . . . . .	48
6.4	Sprint 4 . . . . .	49
6.4.1	Review . . . . .	50
6.5	Sprint 5 . . . . .	51
6.5.1	Review . . . . .	51
6.6	Sprint 6 . . . . .	53
6.6.1	Review . . . . .	54
<b>7</b>	<b>Development</b>	<b>55</b>
7.1	Main Window . . . . .	55
7.1.1	Sprint 1 . . . . .	55
	Design . . . . .	55
	Implementation . . . . .	56
7.1.2	Sprint 4 . . . . .	57
	Design . . . . .	57
	Implementation . . . . .	58
7.1.3	Sprint 5 . . . . .	58
	Implementation . . . . .	58
7.1.4	Sprint 6 . . . . .	60
	Implementation . . . . .	60
7.2	Settings system . . . . .	61
7.2.1	Sprint 1 . . . . .	61
	Design . . . . .	61
	Implementation . . . . .	62
7.2.2	Sprint 2 . . . . .	62
7.2.3	Sprint 5 . . . . .	63

## CONTENTS

---

	Design . . . . .	63
	Implementation . . . . .	64
7.3	Server dialog . . . . .	65
7.3.1	Sprint 1 . . . . .	65
7.3.2	Sprint 2 . . . . .	66
7.3.3	Sprint 4 . . . . .	66
7.3.4	Sprint 6 . . . . .	67
7.4	Internationalization . . . . .	68
7.4.1	Sprint 1 . . . . .	68
7.4.2	Sprint 2 . . . . .	69
	Implementation . . . . .	69
7.4.3	Sprint 3 . . . . .	69
7.4.4	Sprint 6 . . . . .	69
7.5	Plugin system . . . . .	71
7.5.1	Sprint 1 . . . . .	71
	The reimplementaion of the <code>PluginLoader</code> . . . . .	71
	The reimplementaion of configuring plugins . . . . .	72
7.5.2	Sprint 2 . . . . .	72
	The implementation . . . . .	73
7.5.3	Sprint 3 . . . . .	73
	Implementing plugin system as in Luma 2.4 . . . . .	73
	A new and better implementation . . . . .	74
7.5.4	Sprint 4 . . . . .	74
7.6	Browser plugin . . . . .	75
7.6.1	Sprint 2 . . . . .	76
7.6.2	Sprint 3 . . . . .	77
7.6.3	Sprint 4 . . . . .	78
	Entry view . . . . .	81
	HTML templates . . . . .	82
7.6.4	Sprint 5 . . . . .	83
7.6.5	Sprint 6 . . . . .	84
7.6.6	Performance comparison with Luma 2.4 . . . . .	84
7.7	Template plugin . . . . .	86
7.7.1	Sprint4 . . . . .	86
7.7.2	Sprint5 . . . . .	86
7.7.3	Sprint6 . . . . .	86
7.8	Search Plugin . . . . .	88
7.8.1	Sprint 4 . . . . .	88
	Design . . . . .	88
	Implementation . . . . .	88
	Testing . . . . .	89
7.8.2	Sprint 5 . . . . .	90
	Design . . . . .	90
	Implementation . . . . .	90

7.8.3	Sprint 6 . . . . .	92
	Implementation . . . . .	92
7.8.4	Summary . . . . .	93
7.9	Logging . . . . .	95
7.10	Back-end . . . . .	97
7.10.1	Sprint 6 . . . . .	97
7.11	Installation . . . . .	99
7.11.1	Sprint 3 . . . . .	99
	Settings, storage and file-paths . . . . .	99
	Implementation . . . . .	100
7.11.2	Sprint 6 . . . . .	100
7.12	Luma Tool Chain . . . . .	102
7.12.1	Sprint 3 . . . . .	102
	Compiling UI files to Python code . . . . .	102
7.12.2	Sprint 4 . . . . .	102
	Updating the Luma Project file . . . . .	102
	Updating the Luma Resource file . . . . .	103
7.12.3	Sprint 5 . . . . .	103
7.12.4	Sprint 6 . . . . .	104
	Kickstarting Luma translation . . . . .	104
7.12.5	Summary . . . . .	104
<b>8</b>	<b>Testing</b>	<b>105</b>
8.1	White-box . . . . .	105
8.1.1	Unit-test . . . . .	105
8.2	Black-box . . . . .	105
8.2.1	Integration testing . . . . .	105
8.2.2	System testing . . . . .	106
8.3	Tests features . . . . .	106
8.4	Functional requirements testing . . . . .	107
<b>9</b>	<b>Evaluation</b>	<b>112</b>
9.1	Challenges summary . . . . .	112
9.2	Group structure . . . . .	113
9.3	Customer relations . . . . .	113
9.4	Development method . . . . .	113
9.5	Tools and utilities . . . . .	114
9.6	Social gatherings . . . . .	114
9.7	Further work . . . . .	114
9.7.1	Search plugin . . . . .	114
9.7.2	More plugins . . . . .	115
9.7.3	Installation packages . . . . .	115

## CONTENTS

---

<b>10 Conclusion</b>	<b>116</b>
10.1 Comparison of Luma 2.4 and Luma 3.0 . . . . .	116
10.1.1 Comparing functional features . . . . .	116
10.1.2 Comparing non-functional features . . . . .	117
<b>Bibliography</b>	<b>119</b>
<b>Appendix A Official meetings</b>	<b>120</b>
A.1 Meetings with customer . . . . .	120
A.2 Meetings with supervisor . . . . .	121
<b>Appendix B Crash course in LDAP</b>	<b>122</b>
B.1 What is LDAP . . . . .	122
B.2 Directories and directory services . . . . .	122
B.3 The LDAP information model . . . . .	122
<b>Appendix C Cross-platform GUI development</b>	<b>124</b>
C.1 Human Interface Guidelines . . . . .	124
C.2 The menu bar . . . . .	124
C.2.1 Windows and the X Window System . . . . .	124
C.2.2 Mac OS . . . . .	125
C.3 Icons . . . . .	125
<b>Appendix D Cross-platform Python deployment</b>	<b>127</b>
D.1 Available tools . . . . .	127
D.1.1 Distutils . . . . .	128
D.1.2 Setuptools . . . . .	128
D.1.3 Py2exe . . . . .	128
D.1.4 Py2app . . . . .	129
D.1.5 PyInstaller . . . . .	130
<b>Appendix E Luma User documentation</b>	<b>131</b>
E.1 Available formats . . . . .	131

# List of Figures

2.1	The JXplorer interface . . . . .	21
2.2	The Symlabs LDAP browser interface . . . . .	22
3.1	Sequence diagram for expanding user object entries in the Addressbook plugin with Luma 2.4. . . . .	24
3.2	Sequence diagram for improved solution to expanding user object entries in the Addressbook plugin. . . . .	25
3.3	Browsing an entry on an LDAP enabled server with Luma 2.4 . . . . .	26
4.1	Use case diagram . . . . .	36
5.1	Luma architecture . . . . .	38
5.2	Gantt diagram . . . . .	43
7.1	Mockup sketch for the application main window . . . . .	55
7.2	The menu bar structure after sprint 1 . . . . .	56
7.3	Conceptual architecture compared to actual implementation . . . . .	57
7.4	Mockup sketch for the application main window with tabs to handle various plugin views. . . . .	57
7.5	The menu bar structure after sprint 4. . . . .	58
7.6	The menu bar structure as shown when running on Mac OSX. . . . .	58
7.7	The main window after sprint 4 . . . . .	59
7.8	The Luma 3.0 Welcome tab . . . . .	59
7.9	The final menu bar. . . . .	60
7.10	Settings dialog mockup 1. . . . .	61
7.11	Settings dialog mockup 2, with tabbed layout. . . . .	61
7.12	The new settings dialog at the end of sprint 1. . . . .	62
7.13	The new Luma settings system architecture . . . . .	63
7.14	The settingsdialog with plugin configuration. . . . .	64
7.15	The serverdialog after sprint 6 . . . . .	67
7.16	The browser plugin in Luma 2.4 . . . . .	75
7.17	The browser plugin as of sprint 2 . . . . .	77
7.18	Classdiagram for the browser plugin as of sprint 2 . . . . .	78
7.19	The browser plugin as of sprint 3 . . . . .	79
7.20	Failure to connect to a server in Luma 2.4 . . . . .	80
7.21	Error when trying to connect (open) the server named no-ext-server after sprint 3 . . . . .	81
7.22	The same error as in figure 7.21 with sprint 4 . . . . .	82
7.23	Entry view . . . . .	83
7.24	The browser plugin after sprint 6 . . . . .	85
7.25	The new template plugin . . . . .	87
7.26	Search plugin redesign mockup (old design to the left) . . . . .	88
7.27	The search form after sprint 4. . . . .	89



## LIST OF FIGURES

---

7.28	The search form as seen in Luma 2.4. . . . .	89
7.29	The Luma 3.0 5 Search plugin, search-form design. . . . .	90
7.30	The Luma 3.0 5 Search plugin, search-form with autocomplete enabled. . .	90
7.31	The Luma 3.0 5 filterbuilder design. . . . .	91
7.32	The filterbuilder with syntax highlighting enabled . . . . .	91
7.33	The new Luma result view with filter support. . . . .	92
7.34	The Luma 2.4 search plugin. . . . .	93
7.35	The new Luma search plugin with support for multiple result views. . . . .	93
7.36	The loggerwidget . . . . .	95
7.37	Logging to the console when the <code>-verbose</code> flag is used . . . . .	96
7.38	Users has the ability to use a temporary password which is not saved to disk	97
7.39	Windows graphical exe install. . . . .	101
7.40	Windows graphical msi install. . . . .	101
7.41	Linux graphical RPM install. . . . .	101
8.1	Testing structure . . . . .	106
B.1	LDAP directory tree . . . . .	123

# List of Tables

1.1	Challenges that will be tested at the end . . . . .	12
1.2	Team member responsibilities . . . . .	13
2.1	List of comparison criteria. . . . .	20
2.2	Summary of the compared alterantive solutions . . . . .	22
3.1	Plugin attributes and methods . . . . .	27
4.1	Prioritization of the functional and non-functional requirements . . . . .	31
4.2	Use case 1 . . . . .	32
4.3	Use case 2 . . . . .	32
4.4	Use case 3 . . . . .	33
4.5	Use case 4 . . . . .	33
4.6	Use case 5 . . . . .	34
4.7	Use case 6 . . . . .	34
4.8	Use case 7 . . . . .	35
4.9	Use case 8 . . . . .	35
4.10	Use case 9 . . . . .	35
4.11	Use case 10 . . . . .	36
5.1	Risk analysis . . . . .	37
5.2	Product backlog . . . . .	38
5.3	Tasklist . . . . .	41
6.1	Sprint 1 backlog . . . . .	44
6.2	Sprint 2 backlog . . . . .	46
6.3	Sprint 3 backlog . . . . .	48
6.4	Sprint 4 backlog . . . . .	49
6.5	Sprint 5 backlog . . . . .	51
6.6	Sprint 6 backlog . . . . .	53
7.1	Browser performance-comparison results. . . . .	85
8.1	Unit test . . . . .	105
8.2	Module integration . . . . .	106
8.3	The most important test features . . . . .	107
8.4	Functional requirement test 1.1 . . . . .	107
8.5	Functional requirement test 1.2 . . . . .	107
8.6	Functional requirement test 1.3 . . . . .	108
8.7	Functional requirement test 2.1 . . . . .	108
8.8	Functional requirement test 3.1 . . . . .	108
8.9	Functional requirement test 4.1 . . . . .	109
8.10	Functional requirement test 5.1 . . . . .	109
8.11	Functional requirement test 6.1 . . . . .	109

## LIST OF TABLES

---

8.12	Functional requirement test 7.1 . . . . .	110
8.13	Functional requirement test 8.1 . . . . .	110
8.14	Functional requirement test 9.1 . . . . .	110
8.15	Functional requirement test 10.1 . . . . .	111
A.1	Overview: meetings with customer . . . . .	120
A.2	Overview: meetings with supervisor . . . . .	121
C.1	The layout of the menubar on Windows and X Window Systems. . . . .	125
C.2	The layout of the menubar on Mac OS. . . . .	125
E.1	Output formats supported by reStructuredText. . . . .	131

# 1

## Introduction

### 1.1 Project purpose

---

The project purpose is to create a working prototype of Luma using the Qt 4 application framework and utilizing the Model View Controller (MVC) architectural design pattern. Luma is a graphical utility for accessing and managing data stored on LDAP servers distributed under GNU General Public License [1]. Luma, in its current state is in version 2.4 and is using version 3 of the Qt application framework. This application framework is deprecated and has license issues with open source applications using it on Windows. Qt 4 does not have these license issues and converting Luma to Qt 4 means that it gives the opportunity to make it cross-platform and conform to the MVC pattern.

### 1.2 Challenges and problems

---

As stated in the purpose of the project, the current state of Luma is outdated and this comes with many problems. Currently the MVC pattern is not implemented. What would be the model code, is entangled with the view. Our first challenge will therefore be to separate the model and view, and properly implement the MVC pattern.

The second challenge is that Luma should continue to have plugin support. This challenge will be to understand how plugin support works, and how Luma has implemented this before. Probably there will be problems here, changing some of the current plugins from Qt 3 to Qt 4 without rewriting too much. In addition to make a change to Qt 4, we might consider to change functionality and make them more user friendly.

Currently Luma does not run on Windows due to license issues with Qt 3. When writing our new solution, we should make it fully cross-platform. It should be easy to run on all platforms, and should not depend on any OS specific requirements. Writing such a large software that is fully cross-platform is something that none of the group members have previous experience with.

### 1.3 The customer

---

Our customer is Bjørn Ove Grøtan. He is one of the developers on the old Luma. Luma is not owned by a company or a person, it is released under the GNU General Public License [10] and is written solely by voluntarily contributors, like Bjørn Ove Grøtan.

Table 1.1: Challenges that will be tested at the end

Challenges summary
Implement MVC using Qt 4
Continue plugin support
Cross-platform

Bjørn Ove Grøtan is a former NTNU student, and has been in charge of the Luma project for some years. The Luma project was started by German student, Wido Depping, as a student project. The first beta release was made available in November 2003 [16]. Bjørn Ove Grøtan started using Luma in its early stages, and later took over most of the work on maintaining and developing it. Today he has a full-time job, and the time for developing Luma is very limited.

### 1.4 The group

---

Our group consists of six ambitious students, on their last semester on Bachelor in Informatics (BIT) at the Norwegian University of Science and Technology (NTNU). All members are experienced in system design and programming from previous courses at NTNU.

- **Christian Forfang**  
Has previous experience with programming languages Java, C++ and Python, as well as basic design patterns (including MVC) and modeling.
- **Einar Uvsløkk**  
Experienced with Java, Python, Linux and Vim. Some experience with open source gui toolkits like GTK and design patterns, including MVC.
- **Johannes Harestad**  
Knowledge and experience with PHP, MySql and Java. Some experience with MVC patterns, Python, and Scrum.
- **Per Ove Ringdal**  
Previous experience with Java, Python, Linux and SVN. Some experience with MVC from Java Swing.
- **Simen Natvig**  
Experienced with programming Java, Python, C++ and C. Knows basic design patterns and modeling.
- **Sondre Frisvold**  
Experienced with Java, MySql, and to some degree Python. Knowledge of the basics regarding MVC patterns.

### 1.4.1 Group structure

Although all team members will have responsibilities to all parts and aspects of the project, we have come to the conclusion that some sort of structure should be in place from the start. This way we can save some time, should parts of the project suddenly need some attention. The group is structured as shown in table 1.2.

Table 1.2: Team member responsibilities

Project responsibility	Team member
Project Manager	Johannes Harestad
Quality Assurance Control	Christian Forfang
Testing	Per Ove Ringdal
Back-end and LDAP	Simen Natvig
UI and Repository	Einar Uvsløkk
Report and Documentation	Sondre Frisvold

#### Project Manager

The project manager is in charge of the project progression. He must see to it that requirements and priorities are in context and realistic throughout the project lifetime. The project manager will also be responsible for keeping in touch with both the customer and the supervisor.

#### Quality Assurance Control

The Quality Assurance Control (QA) manager is responsible for code review. He must see to it that coding conventions are being followed by other team members, and keep an eye on patterns being followed.

#### Testing

The testing manager is responsible for the code being tested properly. He must see to it that unit tests are written and performed against relevant pieces of the code base.

#### Back-end and LDAP

The back-end and LDAP manager is responsible for maintaining the project's LDAP server used throughout the project lifetime. He is also responsible for making sure that the back-end layer is properly implemented towards the LDAP protocol, and provides a good abstraction of LDAP for the model layer. He should be knowledgeable about the LDAP technology and be able to help the other group members regarding LDAP issues and questions.

## CHAPTER 1. INTRODUCTION

---

### **UI and Repository**

The UI manager is responsible for making sure that the user interface is consistent, usable and effective.

The repository manager has overall responsibility for the code base in the repository being in good shape. He must see to it that the trunk is not broken, branching guidelines are followed and marking stable trunks as tags throughout the project lifetime. He should also ensure that backups of the repository is performed regularly, in case of unforeseen server issues.

### **Report and Documentation**

The report and documentation manager is responsible for the main report progression. He must see to it that each process in the project is documented, and ensure that this documentation finds its way into the report. He also has the overall responsibility for the structure of the project report.

### 1.5 Structure of the report

---

The project report consists of ten chapters. It describes, from start to finish, the complete development process. It also contains a number of appendices, which further describes our study and work.

**Chapter 1** Introduces the problem, the customer and the project group.

**Chapter 2** Describes the pre-studies performed in preparation for the project development. Here we discuss various technologies, development libraries and tools, and development methods we have considered using for this project. We also present a comparison of alternative solutions to our problem, describing both pros and cons with these solutions.

**Chapter 3** Describes our analysis of the Luma 2.4. We present problems and shortcomings with Luma 2.4, as well as a describing the state of the features we are to bring forth in Luma 3.0 .

**Chapter 4** Presents the functional and non-functional requirements for the project. It also contains the use case scenarios for the application.

**Chapter 5** Presents the project planning, including the project milestones, gantt diagram and task-list.

**Chapter 6** Presents an overview for each sprint in the development period.

**Chapter 7** Presents a detailed description of each feature we have been working on during the project development period. The chapter is divided into a section for each, feature, which again is divided into subsection for which sprint it was worked on in.

**Chapter 8** Presents the various testing done for the project.

**Chapter 9** Presents our evaluation of the work we have done, and we discuss challenges and problems that occurred during the project as well as further work that can be done on Luma 3.0 .

**Chapter 10** Ends the main part of the application with a conclusion of what we have delivered.



# 2

## Pre studies

Luma 2.4 is open source and released under the GNU General Public License version 2. In order to make use of the existing code base in our development work, we are required to release our forthcoming work under the same or a compatible open source license. As a result of this, we must also choose technology, development tools and libraries that conform to this license[11].

In the following subsections we explain what technologies, libraries and tools we will use during our development work, why we have chosen them and alternatives to our choices.

### 2.1 Technologies

---

- **Python (2.x)**

One of the requirements for the application is to be runnable on all major platforms (Windows/Linux/Unix/Mac OS). Because Luma 2.4 is written in Python, to ensure that previous contributors are able to continue future contributions, we are required to continue using the Python programming language in our development work. Python is an interpreted, cross-platform, high-level programming language, with support for object-oriented programming. The only major decision we have made, concerning the chosen programming language, is what version to use. Because some of the required libraries we are planning on using is not available with Python 3, we've decided to use the latest release of the Python 2.x branch [21].

- **LDAP(Lightweight Directory Access Protocol)**

LDAP is an application protocol for reading and editing directories over an IP network. For a crash course introduction to LDAP see appendix B

- **Qt 4**

Qt, originally developed by Norwegian company Trolltech which was later acquired by Nokia, is a cross-platform application framework mainly used for developing software with graphical user interfaces (GUIs). In addition to being a widget toolkit (for GUIs) it also includes good internationalization support and other non-GUI features like database access, XML parsing, thread management, network support and a cross-platform API for file handling. While it is developed in C++, it can be used in other languages (like Python) through language-bindings.

## 2.2 Development methods

---

In the following sections we will first describe different development methods we have considered and second why we chose our.

### 2.2.1 Alternative development methods

- **Extreme Programming**

XP in contrast to other development models, the team is always open to change the specifications during the development process. Through the process the teams get constant feedback from the customer and the team must always bear in mind that they have to start over at any minute. Another thing is that when extreme programming is used, the coding starts at day one. So no planning other than the code is done. But XP has several flaws, examples being lack of overall design specification and problems with unstable requirements.

- **Waterfall**

Waterfall follows a strict pattern;

1. Requirement
2. Software design
3. Integration
4. Testing
5. Deployment
6. Maintenance

When one phase is completed, the team starts on the next item. The idea of Waterfall is that the process shall be seen as a steadily flow downwards, like a actual Waterfall. Changes and reviews can be done before moving to the next issue. One of the major drawbacks is the fact that if for instance the customer is unclear on the requirements in the beginning of the project this can backfire later on. Worst case scenario can be that the project has to start from square one again.

- **RUP**

RUP, Rational Unified Process, is an iterative software development process framework, unlike other development methods, RUP is a framework of adaptable processes. Because of this RUP is highly adaptable and can easily be molded into what the team really needs for solving their problem. One key element to RUP is that the development team is expected to adapt and change the processes delivered by RUP to better serve their needs.

- **Scrum**

Scrum is an iterative, incremental methodology for project management used in agile software development. It has great support for dynamic requirement changes and customer integration without losing control of the project progression. In Scrum

you break big problems down into smaller ones. Each of these smaller problems are taken care of in iterations, which Scrum calls *sprints*. For each *sprint* the team creates a part of the software that is shippable. This means that for each *sprint* a piece of working software is incremented to the final product. The *sprint backlog* is the list of features to be implemented for each *sprint*. This list is made in the beginning of a *sprint* and is made out of a bigger list that goes for the whole project, called the *product backlog*. This is a high-level list of the 'what' to be done sorted by importance. The list is maintained throughout the entire project. So for each *sprint* the team looks at the *product backlog* and divides the high-level problems into smaller ones. The team consists of a *Scrum master*, a *Product owner* and *Developers*. The *Scrum master* maintains the process, the *product owner* represents the stakeholders and the developers develop the product. It is up to the team how long each sprint should last, but typically between two and four weeks. During a sprint, the team meets each day. A daily Scrum meeting is held, where the *Scrum master* and the developers is standing and answers the *Scrum master* on three questions; what have you done, what are you doing today, and do you have any problems accomplishing these goals? It is up to the *Scrum master* to maintain the backlogs (product and sprint), and estimate time.

### 2.2.2 Our development method

We have decided to use an adaptation of Scrum as our development method. This is because it is hard to gather all the requirements at the beginning of the project, when more project tasks can be added. We also had to learn the new and old technology as we tried to understand how the old solution worked so we could extend and re-implement it. We figured it was easier to concentrate on parts, instead of doing like Waterfall where you spend weeks of studying and planning to gather all the requirements at once. Because our project should result in a plugin based, multilingual, cross-platform application, a lot of issues most likely would arise concerning the testing phases of the development work. This is where sprints come in to the picture. In Scrum you produce a shippable part of the product, and test it for each sprint. Because of this we think that with a sprint based, and prioritized, development method these issues are more likely to be met and solved with less problems than with a traditional Waterfall model. Another reason for choosing Scrum is the fact that we wanted to have regular meetings with our customer. The customer is therefor a big part of the implementation and can influence the requirements and general design of the solution. Because the group members have different courses it is hard to have daily Scrum stand-up meetings. For this reason we chose not to have a Scrum master.

## 2.3 Libraries and Tools

---

- **Python-ldap**

To be able to access LDAP servers through Python we have chosen to use Python-ldap. This library wraps the OpenLDAP 2.x C-libraries, and provides an object-oriented API to access LDAP directory servers from Python programs. Python-ldap

is distributed under a GPL compatible Python-style license. There is no other open source LDAP libraries available for the Python programming language[15].

- **PyQt4**

To be able to use the Qt 4 framework in Python we have chosen to use PyQt4. PyQt4 is a set of bindings for Qt 4 developed by Riverbank Computing, and available on all major platforms. PyQt4 is available under various licenses, including the GPL, version 2 and 3. Because the Qt framework is licensed under the GNU Lesser General Public License (LGPL), Nokia, the owner of the Qt framework, also wanted to provide Python bindings under the same license as Qt. This resulted in they releasing their own Python bindings, PySide (LGPL), in 2009 as Riverbank Computing was unwilling to also make the PyQt4 bindings available under the LGPL.

Because PyQt4 is more mature, and earlier Luma versions are using PyQt3, as well as being distributed under the GPL, we have chosen to continue using PyQt in this project[20].

- **Qt 4 Designer**

Qt Designer is a part of the Qt 4 framework, and is an program for developing GUI for applications using Qt 4. The Qt 4 Designer enables us to create GUIs in *a what you see is what you get* manner. This eliminates much of the time we would ordinary spend on writing GUI code manually, and instead we can focus on the code for the application logic.

- **Apache Subversion**

Subversion (SVN) is a software versioning and a revision control system, founded and sponsored by CollabNet Inc since 2000 [3]. As an alternative to SVN we have considered using Git [8], which opposed to SVN is a distributed revision control system. Because all team members have been working with SVN on different projects in the past, we have decided to use SVN for revision control, eliminating the need for additional training required for first time Git users.

We will still, during the early phases of the project, keep Git as an open alternative. If SVN prove to be difficult when it comes to merging pieces of code that multiple team members have been working on, we will reconsider switching to Git, since it has better support for this.

- **Eclipse (w/PyDev and Subclipse plugins)**

Eclipse is an Open Source Integrated Development Environment (IDE) [7], whom all of team members are familiar with. By extending the IDE with plugins for Python (PyDev) [19] and SVN (Subclipse) [24], it will provide the team with a familiar development environment.

- **L<sup>A</sup>T<sub>E</sub>X**

We have chosen to use the document markup language/preparation system, L<sup>A</sup>T<sub>E</sub>X, for the project report [14]. Even though few of the team members have any real experience with L<sup>A</sup>T<sub>E</sub>X, we consider the *value* of learning it, superior to the *effort* required learning it.

- **Dia**

For modeling and diagrams we use Dia, a free and open source program [6].

- **Redmine**

Redmine is a free and open source, web-based project management and bug-tracking tool, which we will use to keep track of time spent on the project, assigning and register new issues, team communication, documents and as a wiki [22].

## 2.4 Alternative solutions

---

A quick search on Google, using the query `LDAP client`, revealed a lot of alternative solutions to Luma. In order to create a summary on alternative solutions, we followed the criteria outlined in table 2.1 when comparing them with Luma.

Table 2.1: List of comparison criteria.

Cross-platform	It should be able to run on all major platforms (Windows/-Linux/Unix/Mac OS)
Standalone client	It should be a standalone application, not a plugin or a web-based solution
Plugin support	It should be possible to add application functionality through plugins
Browse and edit	It should be possible to both browse and edit LDAP entries
User friendly UI	It should provide an intuitive and easy-to-use user interface
Multilingual	It should support language translations
Up to date technology	It should use up to date technology
Open source	It should be distributed under an Open Source Definition compliant license <sup>1</sup>

Few of the many alternative solutions that we found, managed to comply to all of our criteria. Most of the solutions did not meet the criteria open source, cross-platform, or standalone client. We will now present the clients that best conformed to the given criteria, and conclude the comparison with a summary in table 2.2.

### 2.4.1 JXplorer

JXplorer is maybe the biggest competitor client solution. This client meets all of our criteria, with the last version released 22<sup>th</sup> May, 2010. JXplorer claims to have the world's finest LDAP browser<sup>2</sup> and is shown in figure 2.1. The client is developed by a fairly large team of contributors. The project is also sponsored by tools and technologies including *BitRock*, *IntelliJ IDEA* and *Sourceforge.net*.

---

<sup>1</sup>The Open Source Initiative maintains both the approval process and the Open Source Definition. A list of approved licenses can be found at <http://www.opensource.org/licenses/index.html>

<sup>2</sup>So far...

## 2.4. ALTERNATIVE SOLUTIONS

Our solution will differ in the technology, and license used. JXplorer uses Java, and got a license based on BSD. Based on the current version of Luma, JXplorer seems to have a better UI when it comes to effectiveness and usability [12].

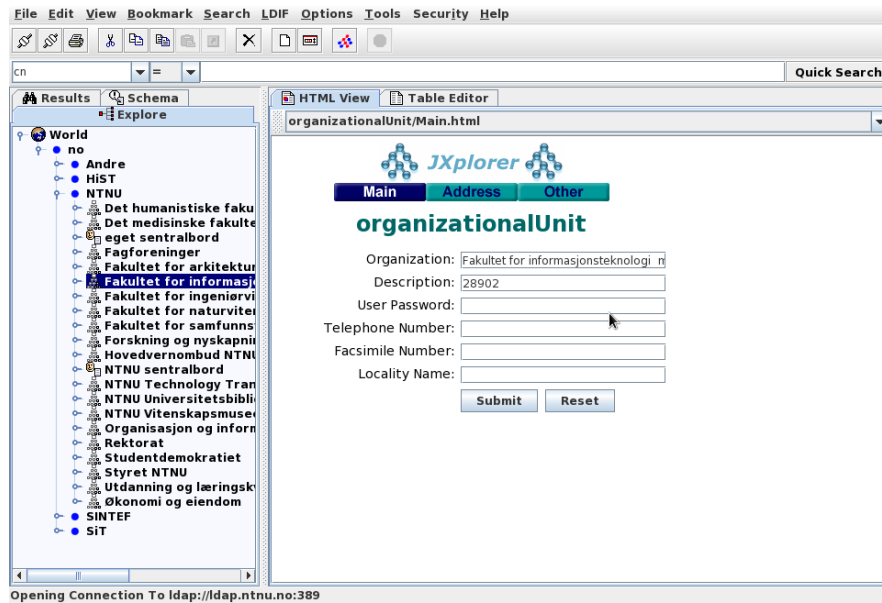


Figure 2.1: The JXplorer interface

### 2.4.2 Argonne's LDAP browser/editor

A LDAP client with a graphical user interface, written in Java and developed by Argonne national laboratory. It is not free, but it is cross platform. The documentation does not say anything about plugin support, so this criteria is missing too. Our solution will have the advantage of being free and have plugin-support [5].

### 2.4.3 LDAP Browser from Symblabs

Another client written in Java, developed by Symblabs. The user interface looks intuitive and clean, seen in Figure 2.2. With no support for plugins and no multilingual support, the client from Symblabs differs from our product [25].

### 2.4.4 Summary

Table 2.2 summarizes the clients we have compared. Even though we have only included a selection of clients that best conform to our given criteria, there exists other, platform-dependent solutions as well.

## CHAPTER 2. PRE STUDIES

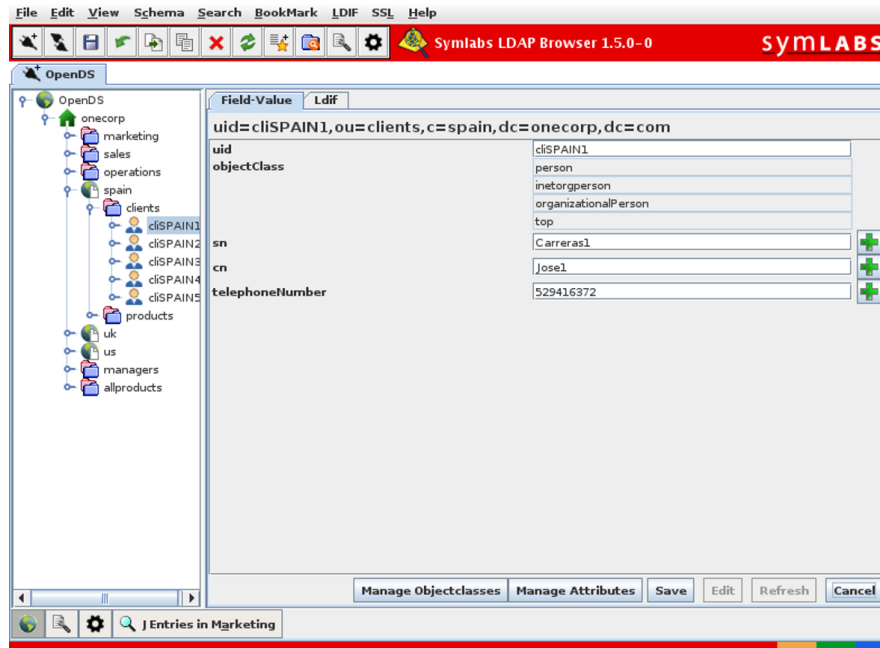


Figure 2.2: The Symlabs LDAP browser interface

Table 2.2: Summary of the compared alterantive solutions

Criteria	JXplorer	Argonne	Symlabs
Cross-platform	X	X	X
Standalone client	X	X	X
Plugins	X	?	
Browse/Edit	X	X	X
Userfriendly interface	X	?	X
Multilingual	X		
Up to date technology	X	X	X
Open Source	X		
	8/8	4/8	5/8

# 3

## Problems with Luma 2.4

The last stable release of Luma was released on February 27<sup>th</sup> 2008, as version 2.4. Throughout this paper, we will refer to this version as Luma 2.4. The code base from this release is what we will be working with, during the project lifetime.

As part of our pre studies and development-preparations, we did a comprehensive examination of Luma 2.4, looking at both code quality, design patterns, functionality and usability. Depending on the state of this code base we are supposed to port, improve, and/or write new code. Examining Luma 2.4 revealed a number of issues, concerning both implementation and usability. Primarily, the overall application architecture is somewhat diffuse, and lacking some solid *framework* foundation. The various problems, bugs and shortcomings in Luma 2.4 will be discussed in the following section.

### 3.1 Problems

---

When large data sets<sup>1</sup> is returned from an entry expansion on a LDAP enabled server, the application interface of Luma 2.4 becomes unresponsive for several seconds, as the GUI is being updated. Even though this expansion is a generally expensive operation, we do believe that there is room for improvements here. In order to reduce the amount of time the application is being unresponsive, we need to implement and optimize code wherever possible. Hopefully, and most likely, it is possible to do these expensive operations even *without* leaving the application unresponsive.

Furthermore, smarter solutions is needed to handle undo actions and cancel operations in the application. Especially in places with the same amount of entry expansion as discussed above. As an example, let's take a look at the current implementation of the Addressbook plugin<sup>2</sup>.

#### **Example**

*Suppose a user has selected to work against a server containing a lot of entries with user objects. When a server is selected a routine, `getUserEntries()` is executed to get all the user-object entries from the server. Then the user selects one of these entries, to look at a specific user object. In Luma 2.4 if the user now decides to go back, or cancel,*

---

<sup>1</sup>i.e. data sets containing < 10 000 entries, with each entry containing attributes and corresponding attribute values.

<sup>2</sup>The Addressbook plugin is used for managing user entries on a LDAP enabled server.



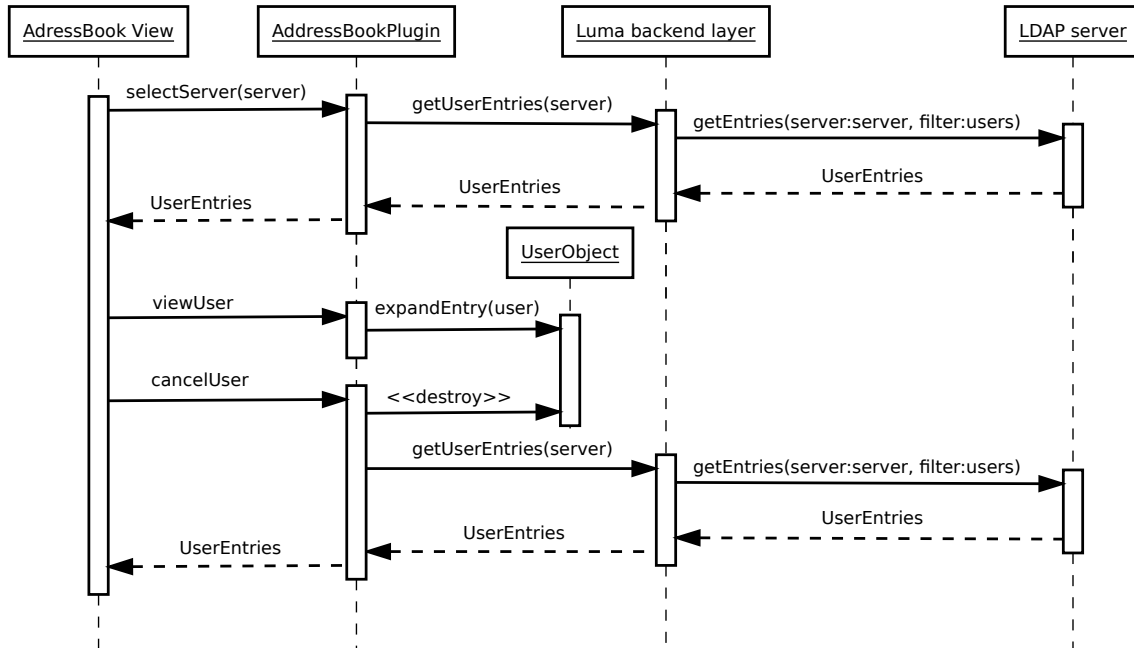


Figure 3.1: Sequence diagram for expanding user object entries in the Addressbook plugin with Luma 2.4.

*the current operations on the user object, the same expensive `getUserEntries()` routine is automatically executed prior to returning to the previous expanded user-objects overview.*

Figure 3.1 shows a conceptual sequence diagram, illustrating the execution flow of the previous example, as done in Luma 2.4. We believe that the second execution of the `getUserEntries()` routine is very expensive, completely unnecessary, and should be avoided in the reimplementation. Figure 3.2 illustrates a possible improvement. Here the last `getUserEntries()` execution is removed, as the user object entries already is present after the first execution.

### 3.2 Bugs

As in most software projects with large code bases, there exist documented and undocumented bugs in the Luma 2.4 implementation. Documented bugs can be found on the official bug tracker[17].

Perhaps the most noteworthy implementation bug present in this release, is the extensive use of `QWidget.setText(None)`, which raises a lot of unnecessary exceptions at runtime. We also experience that messages dialogs, meant to inform the user about errors and warnings, are implemented wrong. It is quite alarming, and almost a paradox, that messages meant to inform about illegal operations is throwing exceptions them self.

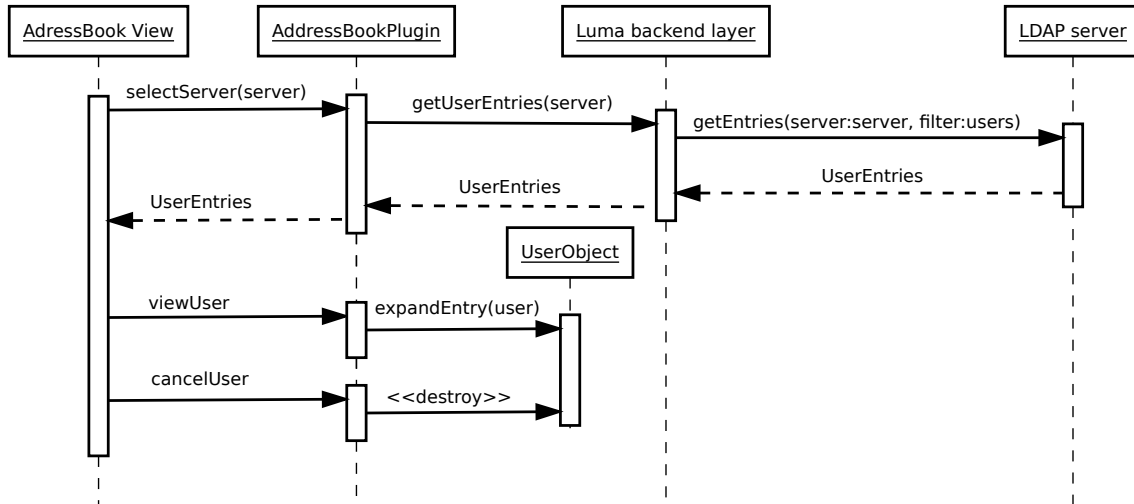


Figure 3.2: Sequence diagram for improved solution to expanding user object entries in the Addressbook plugin.

### 3.3 Shortcomings

Luma 2.4 has several shortcomings which are subject for improvements. The menu bar, for instance, does not follow any standard naming conventions, and the grouping of menu actions could be done in a more intuitive fashion. There exist Human Interface Guidelines (HIG) for most popular platforms, where, among other things, the menu bar has its own platform best-practices. The menu bar, with its challenges, is discussed in appendix C.

Another shortcoming is that no complete translation of the application is present. Parts of the application remain in the default language<sup>3</sup> when another language is selected. Related to this shortcoming is the need for reloading the whole application upon switching to a new language.

#### Example

*Suppose a user has navigated to one part of the application and, in order to better understand the given options, decides to switch to another language. While doing this in Luma 2.4 the application will be set back to its initial on-start state, leaving the user in need to navigate back to where he or she was.*

This behavior affects the application work flow, and is both unnecessary and inconvenient for the user. We believe a complete translation of the application can be done at runtime without changing application state. The overall feedback to the user is also subject for improvements. Even though feedback is given in some situations, it is completely absent in other situations.

<sup>3</sup>The default language is English, the same as the development language

## CHAPTER 3. PROBLEMS WITH LUMA 2.4

### Example

While browsing an entry on a LDAP enabled server, and changing or adding attribute values, all changes must be saved before another entry can be accessed. In Luma 2.4 the user is given no feedback, other than cryptic error output in the log, if he tries to switch to another entry with unsaved changes.

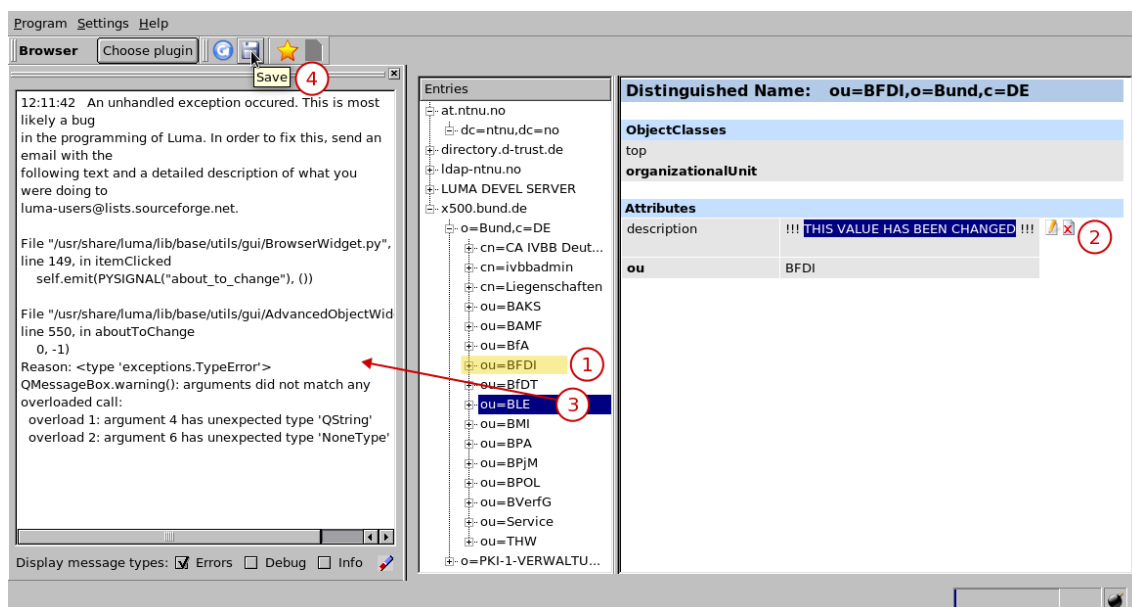


Figure 3.3: Browsing an entry on an LDAP enabled server with Luma 2.4

Figure 3.3 show the browser plugin in a screen shot from Luma 2.4, illustrating the lack of proper feedback when a user performs an illegal operation. **1.** a server entry is selected; **2.** an attribute value is edited; **3.** another entry is selected, resulting in no feedback, but the error message in the logger window; and **4.** changes need to be saved in order to complete step 3. This example is also related to the message dialog implementation bugs, discussed above, and is something that must be fixed when we start our development work.

## 3.4 Plugin system

An important feature of Luma 2.4 is the support for plugins. A plugin is simply a set of software components that adds specific capabilities to a larger software application. This way, anyone can create their own plugins to give Luma new functionality. Luma 2.4 comes by default with many plugins, like the address book and administrator plugins.

This section describes how a plugin has to be implemented, and how Luma 2.4, as the host application, implements the support for such a plugin.

### 3.4.1 Plugin implementation

The first step in creating your own plugin, is to create a new folder for the plugin. This folder must be put in the plugins locations.

```
/lib/luma/plugins/plugin-name
```

In Luma 2.4 there is no rules for how this folder must be structured. The only requirement is to have certain attributes and methods in the `__init__.py` file for the plugin folder. These attributes are described in table 3.1.

Table 3.1: Plugin attributes and methods

Attributes and methods	Description
<code>lumaPlugin</code>	boolean variable
<code>pluginName</code>	variable that sets the plugin name
<code>pluginUserString</code>	variable for what plugin name the user will see
<code>version</code>	variable for the version number
<code>author</code>	variable for author name
<code>getIcon(iconPath)</code>	method used to get the plugin's icon
<code>getPluginWidget(parent)</code>	the plugin itself, this widget will be loaded into the main window
<code>getPluginSettingsWidget(parent)</code>	method used to get the plugin's settings widget used when setting up the plugin
<code>postprocess()</code>	unknown, no plugins have implemented this method that we know of

The plugin does not need to implement these attributes and methods, it can simply initialize all attributes to `None` and put a `pass` statement in all the methods, and still be a valid plugin. Because of this, it is possible to create plugins that do not have any widgets, views or functionality.

After creating a folder and the `__init__.py` file as described, Luma 2.4 will look at it as a complete plugin. Furthermore it is up to the author how he wishes to implement the plugin. The author may create as many modules as he wishes, inside the plugin-folder.

### 3.4.2 Plugin system implementation

Luma 2.4 implements the plugin system to take use of plugins described above with the following features:

1. Let the user choose what plugins he wants to have available.
2. If the plugin allows it, let the user configure it.
3. Load and use a plugin.

## CHAPTER 3. PROBLEMS WITH LUMA 2.4

---

4. Unload and detach a plugin.

The implementation of Luma 2.4 plugin system is not very modular. More will be explained and compared in section 7.5 when we explain the reimplementation of the Luma plugin system.

# 4

## Requirements

The final solution should fulfill a number of functional and non-functional requirements. Together these requirements specify the overall goals of the project, and the aim will be, given sufficient time, to complete all of them.

In accordance with an agile software development methodology, the requirements will initially be given as user stories which will later be supplemented by use cases. The user stories will be supplemented with use cases before the development starts. User stories are useful in a time constrained project like this since they can be written very quickly and provide a good overview of required functionality. This means that we can start working on the features earlier as no time is wasted on the definition of detailed requirements at an early stage [2]. If more documentation of the requirements are needed, this will be provided later. One of the main reasons for using user stories is to keep the startup phase of the project easy and straight forward. In that way the focus is maintained on the software's intended functionality. It also facilitates an iterative development starting with the highest prioritized feature first.

### 4.1 Functional requirements

---

The following user stories were defined:

- **FR1:** As an user of Luma, I want to be able to maintain a list of LDAP servers and associated configuration needed to connect to them.
- **FR2:** As an user of an LDAP server, I want to be able to connect to an LDAP server and browse its content.
- **FR3:** As a maintainer of the content on an LDAP server, I want to be able to delete entries on an LDAP server.
- **FR4:** As a maintainer of the content on an LDAP server, I want to be able to add new entries to an LDAP server.
- **FR5:** As a maintainer of the content on an LDAP server, I want to be able to edit entries on an LDAP server.
- **FR6:** As an user of an LDAP server, I want to be able to search for entries on the server.

- **FR7:** As an user of an LDAP server, I want to be able to export entries from LDAP servers to the local system.
- **FR8:** As an user of Luma, I want to be able to add custom-made plugins created by me or others.
- **FR9:** As an user of Luma, I want to be able to see details of errors that have occurred in a log.
- **FR10:** As an user of Luma, I want to be able to specify template-objects.

### 4.2 Non-functional requirements

---

The following user stories were defined:

- **NFR1:** As the product-owner I want Luma to be implemented using using PyQt4.
- **NFR2:** As the product-owner I want Luma to utilize the Model-View-Controller-pattern.
- **NFR3:** As the product-owner, I want Luma to be runnable on all major platforms (Windows/Linux/Unix/Mac OS).
- **NFR4:** As the product-owner, I want Luma to be GPL (open source).
- **NFR5:** As the product-owner, I want Luma to be written in understandable code so that other people can continue extending it later.
- **NFR6:** As an user of Luma, I want it to be reasonably easy to use.
- **NFR7:** As an user of Luma, I want it to have good documentation.
- **NFR8:** As an user of Luma, I want it to be easily translatable to my language (multilingual).

### 4.3 Requirements summary

---

In table 4.1 the functional and non-functional requirements are listed and prioritized. The priorities should be interpreted in relation to each other.

Table 4.1: Prioritization of the functional and non-functional requirements

Name	Priority / Importance	Dependencies
FR1	High	None
FR2	High	FR1
FR3	Medium	FR2
FR4	Medium	FR2
FR5	Medium	FR2
FR6	Medium	FR2
FR7	Medium	FR2
FR8	Low	None
FR9	Low	None
FR10	Low	FR4
NFR1	High	None
NFR2	High	None
NFR3	High	None
NFR4	High	None
NFR5	Medium	None
NFR6	Medium	None
NFR7	Low	None
NFR8	High	None

### 4.4 Use cases

---

The use cases below describes the main functions of Luma 3.0 . The use cases clarify how the software is supposed to work, and define the foundation for the testing to be performed at a later stage [27].



## CHAPTER 4. REQUIREMENTS

---

Table 4.2: Use case 1

Adding a server - FR1

Precondition	Luma is installed.
Postcondition	The server and its configuration is saved by Luma.
Main flow	
1.	The user starts Luma.
2.	The user opens the server list and clicks add.
3.	The users types inn a nickname for the server and clicks ok. (The server is added to the list.)
4.	The user clicks on the nickname in the server list and gets an overview of the current (default) settings.
5.	The user changes the default settings to the correct ones for the server he wishes to add.
6.	The user selects "Ok" and the dialog saves and exits.
Alternative flow	
3a.	The user types inn an existing server-name and the process is canceled and the user notified.
6a.	The user clicks "Cancel" and after confirming the selection the dialog exits without saving changes.

Table 4.3: Use case 2

Browsing a server - FR2

Precondition	Luma is configured with a server.
Postcondition	The users is able to browse the contents of the server.
Main flow	
1.	The user starts Luma.
2.	The user selects the "Browse" functionality and then the server to browse.
3.	The user is shown the content of the server and is able to navigate through its content.
Alternative flow	
3a.	The user is notified the server can't be connected to.

Table 4.4: Use case 3  
Deleting an entry - FR3

Precondition	Luma is configured with a server to which the users has "delete"-privileges.
Postcondition	The entry chosen for deletion by the user is deleted from the server.
Main flow	
1.	The user browse to the item he wishes to delete.
2.	The users right-clicks the item to get a context menu of available operations.
3.	The users selects "Delete" and is asked to confirm the choice.
4.	The item is deleted from the server.
Alternative flow	
4a.	The user does not have permission to delete the entry and is notified of this.

Table 4.5: Use case 4  
Adding a new entry - FR4

Precondition	Luma if configured with a server to which the user has permission to add new items.
Postcondition	The new entry is saved on the server.
Main flow	
1.	The user browses to the position of the new item.
2.	The user right-clicks the item that shall be the parent of the new item and chooses "add new entry".
3.	A dialog-box pops up asking the user for the details of the new item.
4.	The users confirms the dialog box and the new item is added.
Alternative flow	
4b.	The user cancels the dialog box and no new item is added.
4b.	The user does not have permission to add a new item (at this position) so the item is not added.

## CHAPTER 4. REQUIREMENTS

---

Table 4.6: Use case 5  
Editing an entry - FR5

Precondition	Luma is configured with a server to which the users has "edit"-privileges.
Postcondition	The entry is changed according to the users need.
Main flow	
1.	The user locates the entry to edit through Luma and selects it.
2.	The entry is shown in its own part of the window.
3.	The user clicks the edit-icon for the field he wishes to change.
4.	A dialog box pops up asking what the field should be changed to.
5.	The user clicks "save" and the edited entry is saved on the server.
Alternative flow	
5a.	The user doesn't have permission to change the item and is notified of this.
5b.	The user clicks "Cancel" and the changes are discarded after confirming the choice.

Table 4.7: Use case 6  
Searching for entries - FR6

Precondition	Luma is configured with a server.
Postcondition	The user manages to locate the searched-for entry.
Main flow	
1.	The user starts the "Search"-plugin.
2.	The user selects the server he wishes to search on, as well as the correct baseDN.
3.	The user inputs the filter for the research and clicks "Search".
4.	A list is updated with the found entries.
5.	The user double-clicks on an item to open and view it.
6.	The search is finished.
Alternative flow	
3a.	The user selects to use a wizard in order to specify what to search for.
4a.	No items are found so the list is empty.

Table 4.8: Use case 7  
Exporting entries - FR7

Precondition	Luma is configured with a server.
Postcondition	The entry is export to a file.
Main flow	
1.	The users browses to the correct item.
2.	The user right-clicks on the item and chooses to export it
3.	The user is asked what to export (single entry, entry+children) and where the file should be saved and its format.
4.	The entries are exported to a file.

Table 4.9: Use case 8  
Adding new plugins - FR8

Precondition	Luma is installed. The new plugin is in a format acceptable by Luma.
Postcondition	The plugin is accessible from within Luma.
Main flow	
1.	The user moves the folder with the new plugin to Luma's designated folder for plugins.
2.	The user launches Luma and opens the plugin manager.
3.	The user sees the new plugin in the list of plugins and checks the check-box next to "activate".
4.	The plugin is accessible from Luma's list of plugins.

Table 4.10: Use case 9  
Logging - FR9

Precondition	Luma is running.
Postcondition	The user sees a log of logged events.
Main flow	
1.	The user selects "Show logger" from the main-menu.
2.	The log-window appear on the screen.
3.	The user sets the different log-levels he want to see.
4.	The log-window is updated to only show the log-items with the levels selected.

## CHAPTER 4. REQUIREMENTS

Table 4.11: Use case 10  
Template objects - FR10

Precondition	Luma is installed.
Postcondition	The new template-object is usable when adding new items.
Main flow	
1.	The users enters the template-plugin.
2.	The user chooses to create a new template.
3.	The user adds a new attribute to the template.
4.	The user adds a new objectClass to the template.
5.	The user chooses to save the new template.
6.	The template is saved by Luma.
Alternative flow	
2a.	The name is already used so the user is asks for a new one.
5a.	The user chooses to discard the changes to the template. The changes are not saved.

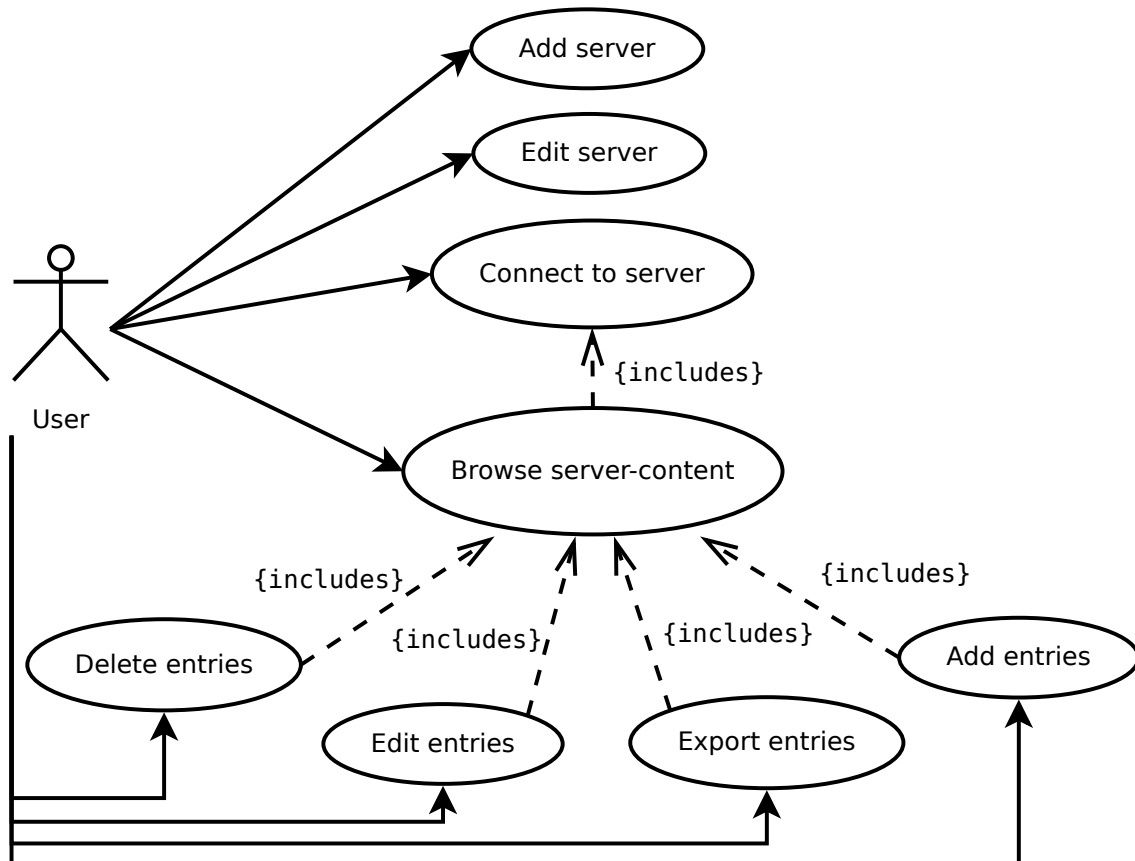


Figure 4.1: Use case diagram

# 5

## Planning

### 5.1 Milestones

---

These are the milestones of the project:

1. Preliminary report 31.01.2011
2. Mid-semester report 28.02.1011
3. Pre-final report 04.04.2011
4. Final report 15.05.2011
5. Software finalized 15.05.2011
6. Presentation of projects 23.05.2011

### 5.2 Risk analysis

---

Through risk analysis the group was able to plan remedial actions and be prepared for possible risks. Table 5.1 shows our risk analysis.

Table 5.1: Risk analysis

Description	L <sup>1</sup>	I <sup>2</sup>	R <sup>3</sup>	Preventative action	Remedial action
Misunderstood features	6	9	54	Customer contact	Rework feature
Insufficient (Py)Qt-knowledge	7	7	49	Prioritize study	Dedicate to study
Insufficient LDAP-knowledge	7	7	49	Prioritize study	Dedicate to area
Inadequate work-process	5	9	45	Clear working process	Adjust and adapt
Internal disagreements	6	7	42	Plan well ahead	Conflict resolution
Customer issues	4	8	32	Keep regular contact	Conflict resolution
Optimistic (sprint) schedule	5	5	25	Good time-estimates	Cut features
Data loss	3	8	24	Regular backups	Data retrival
Illness	3	8	24	Good health	Cut features

5.3 Architecture

Figure 5.1 describes the top level architecture for Luma 3.0 . Inside the **Luma**-package we have a **Plugins**-package where the various plugins will be placed and loaded from. The browser and template plugins are provided as examples. The **Luma**-package is responsible for starting and displaying the program, and the **Backend**-package will be used by the plugins and Luma for communicating with the LDAP-server. The plugins themselves will be designed using the Model View Controller-architecture and extend **QWidget** so they can be easily embedded into the main window of the Luma application.

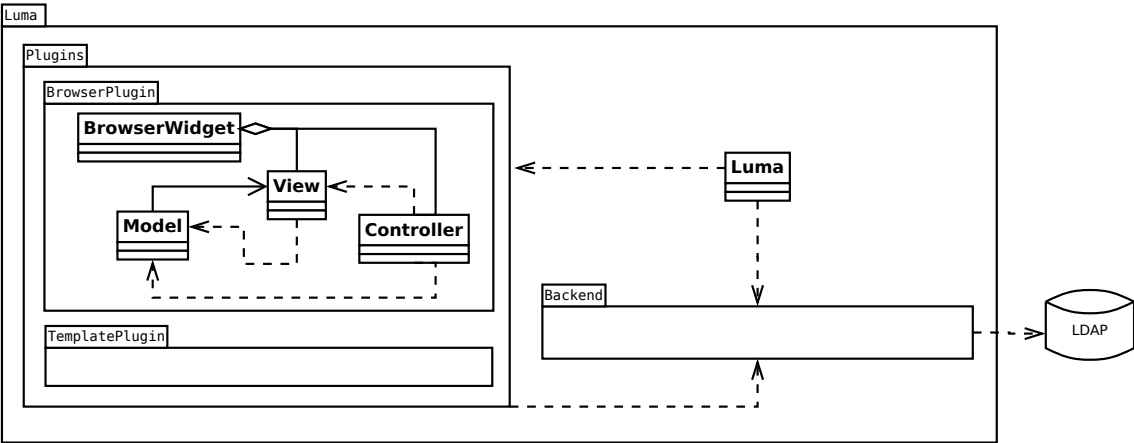


Figure 5.1: Luma architecture

5.4 Product backlog

Table 5.2: Product backlog

Priority	ID	Description	Part of Sprint
	MW	<b>Main Window</b>	
High	MW1	Port the old MainWindow from Qt3 to Qt4 with MVC	1
Low	MW2	About, credits and license dialog.	1
Medium	MW3	Widget to host plugins	1
Medium	MW4	Host the plugin toolbar	2,3
High	MW5	Tabs for plugins	3,4

... continued on next page

<sup>1</sup>Likelihood on a scale from 1-9 (9 being most likely).  
<sup>2</sup>Importance on a scale from 1-9 (9 being most important).  
<sup>3</sup>Risk is the product of Likelihood and Importance ( $L * I$ )

## 5.4. PRODUCT BACKLOG

Priority	ID	Description	Part of Sprint
	SD	<b>Server dialog</b>	
High	SD1	Server list - write from/to files (convert old code)	1
High	SD2	Dialog for server list and configuration of servers	1
Low	SD3	<i>Test connection</i> -functionality	6
	SD4	Usability improvements	4
	SD5	<i>Save</i> -functionality (OK, Save, Cancel)	2
	PS	<b>Plugin support</b>	
High	PS1	Find and import plugins	1
Medium	PS2	Dialog for plugin configuration	1,2,3,4
Low	PS3	Plugin toolbar	2
	L	<b>Language/Translation</b>	
Med	L1	Locate/load files dynamically	1
Med	L2	Dynamic translation for all widgets	4
Low	L3	Port old/create new Qt translation files	
	O	<b>Other</b>	
Med	O1	Use Qt's resource system for images	1
Low	O2	Replace deprecations	6
	BP	<b>Browser plugin</b>	
High	BP1	View content on server	2
Med	BP2	Add items	3,5,6
Med	BP4	Edit items	3,6
Med	BP5	Remove items	4
Low	BP6	Set filter	2
Low	BP7	Set limit	2
Low	BP8	Export items to file	4
Low	BP9	Counting entries recieved on search	4
Low	BP10	Custom html views	4
	LO	<b>Logging</b>	
High	LO1	New backend-system	1
Med	LO2	Loglevel filtering in the loggerwidget	2
High	LO3	Convert to new system	1
	CS	<b>Configurations and settings</b>	
High	CS1	Read/write to files	2
High	CS2	Settings system (backend) that GUI can use	1,2
Med	CS3	Settings dialog (GUI)	1
Low	CS4	Generic settings wrapper for plugins	4

... continued on next page



## CHAPTER 5. PLANNING

Priority	ID	Description	Part of Sprint
	I	<b>Installation</b>	
High	I1	Prepare the application filepath usage	1,4
High	I2	Write cross-platform install script	1,3
Med	I3	Document installation process for Linux distribution packagers	6
Low	I4	Create frozen application bundles for Windows and Mac OSX	
	T	<b>Template</b>	
High	T1	GUI and design	4
High	T2	Read/write templitelist from/to file	4
Med	T3	Basic functionality	4,5
Med	T4	Attributes changed on removal and adding of objectclasses	5
Low	T5	Non-must attributes can be changed to custom-must	5
	B	<b>Backend</b>	
Med	B1	Rewrite backend to separate out GUI-code	6
High	B2	Rework threading to avoid segfault on closed plugins	6
Med	B3	Provide async API to LumaConnection (Qt signal+slots)	6
Low	B4	Add ability to set temporary password for servers	6
	S	<b>Search</b>	
High	S1	Redesign UI (more space efficient and intuitive)	4
High	S2	Validate search filters	4
High	S3	Use search filters	4
High	S4	Handle errors and exceptions (User feedback)	5
High	S5	Retrieve search results	5
High	S6	Display search results	5
Med	S7	Build search filters	5
Med	S8	Save search filters	5
Med	S9	Remember search settings	5
Low	S10	Additional filtering on retrieved search results	5
	LD	<b>Luma Documentation</b>	
High	LD1	Setup documentation tools for the project	6
Med	LD2	Write userguide	6

... continued on next page

## 5.5. GANTT AND TASK LIST

Priority	ID	Description	Part of Sprint
Med	LD3	Write/document installation process	6
Med	LD4	Write/document instructions for developers	6
	LTC	<b>Luma Tool Chain</b>	
Low	LTC1	Create tools for handling application resources	3,4,5,6

### 5.5 Gantt and Task list

In figure 5.2 the project lifetime is described as a gantt diagram. Superficial tasks are described in figure 5.3.

#### Task list explanation

The *Duration* column in figure 5.3 describes the duration time in **d**ays or **h**ours. The *Work* column describes the amount of work this task is expected to include. For example, the expected amount of work per team member each week is 20 hours. With 5 work days a week this yields

$$\frac{20\text{hours}}{5\text{days}} = 4\text{hours/day}$$

One sprint has a duration of 2 weeks =  $10d_{\text{duration}}$ , when excluding the weekends. With a team consisting of 6 members this results in a total amount of work per sprint

$$10d_{\text{duration}} * 6 = 60d_{\text{work}}$$

Table 5.3: Tasklist

WBS	Name	Start	Finish	Work	Duration
1	<b>Project Start</b>	<b>Jan 10</b>	<b>Jan 28</b>	<b>90d</b>	<b>15d</b>
1.1	Establish group	Jan 10	Jan 10	1d	1d
1.2	Pre studies	Jan 10	Jan 28	15d	15d
1.3	Customer meeting	Jan 21	Jan 21	3d	2h
1.4	Writing requirements	Jan 21	Jan 28	33d	5d 2h
1.5	Planning	Jan 11	Jan 28	28d	14d
1.6	Tool setup	Jan 17	Jan 28	10d	10d
2	<b>Sprint 1</b>	<b>Jan 31</b>	<b>Feb 11</b>	<b>60d</b>	<b>10d</b>
2.1	Planning	Jan 31	Feb 1	9d	2d
2.2	Implementation	Feb 1	Feb 10	24d	8d
2.3	Testing	Feb 2	Feb 11	21d	8d
2.4	Evaluation	Feb 4	Feb 11	6d	6d

... continued on next page

## CHAPTER 5. PLANNING

---

WBS	Name	Start	Finish	Work	Duration
3	<b>Sprint 2</b>	<b>Feb 14</b>	<b>Feb 25</b>	<b>60d</b>	<b>10d</b>
3.1	Planning	Feb 14	Feb 15	9d	2d
3.2	Implementation	Feb 15	Feb 24	24d	8d
3.3	Testing	Feb 16	Feb 25	21d	8d
3.4	Evaluation	Feb 18	Feb 25	6d	6d
4	<b>Sprint 3</b>	<b>Feb 28</b>	<b>Mar 11</b>	<b>60d</b>	<b>10d</b>
4.1	Planning	Feb 28	Mar 1	9d	2d
4.2	Implementation	Mar 1	Mar 10	24d	8d
4.3	Testing	Mar 2	Mar 11	21d	8d
4.4	Evaluation	Mar 4	Mar 11	6d	6d
5	<b>Sprint 4</b>	<b>Mar 14</b>	<b>Mar 25</b>	<b>60d</b>	<b>10d</b>
5.1	Planning	Mar 14	Mar 15	9d	2d
5.2	Implementation	Mar 15	Mar 24	24d	8d
5.3	Testing	Mar 16	Mar 25	21d	8d
5.4	Evaluation	Mar 18	Mar 25	6d	6d
6	<b>Sprint 5</b>	<b>Mar 28</b>	<b>Apr 8</b>	<b>60d</b>	<b>10d</b>
6.1	Planning	Mar 28	Mar 29	9d	2d
6.2	Implementation	Mar 29	Apr 7	24d	8d
6.3	Testing	Mar 30	Apr 8	21d	8d
6.4	Evaluation	Apr 1	Apr 8	6d	6d
7	<b>Sprint 6</b>	<b>Apr 26</b>	<b>May 13</b>	<b>90d</b>	<b>10d</b>
7.1	Planning	Apr 26	Apr 27	9d	2d
7.2	Implementation	Apr 27	May 12	36d	12d
7.3	Testing	Apr 28	May 13	34d	12d
7.4	Evaluation	Apr 29	May 13	11d	11d
8	(MS) Report: Preliminary version	Jan 31	Jan 31	N/A	N/A
9	(MS) Report: Mid-semester version	Feb 28	Jan 31	N/A	N/A
10	(MS) Report: Pre-final	Apr 8	Apr 8	N/A	N/A
11	(MS) Report: Final	May 15	May 15	N/A	N/A

## 5.5. GANTT AND TASK LIST

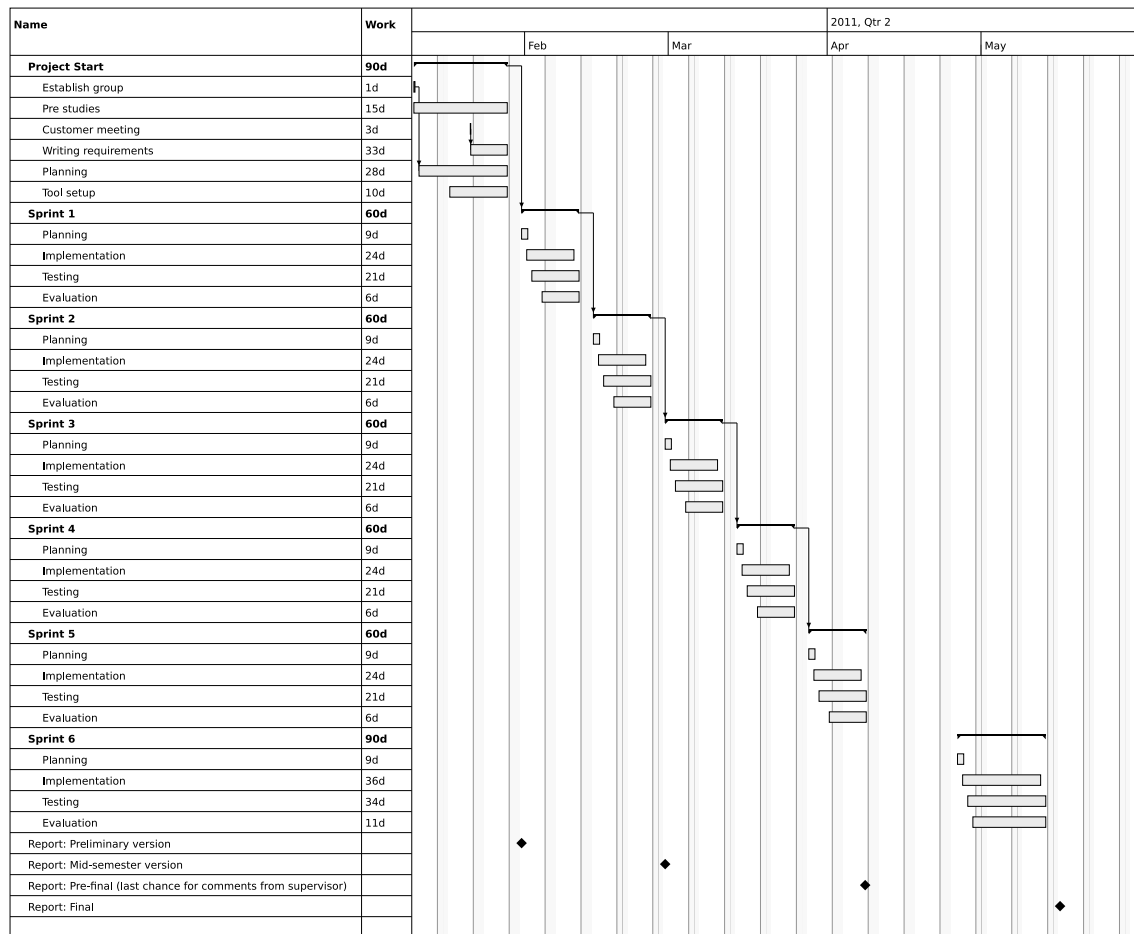


Figure 5.2: Gantt diagram

# 6

## Sprint overviews

This chapter includes agendas and reviews for each sprint in the project development. Further details of our work for each sprint can be found in chapter 7, this is just an introduction. We would also like to mention that before we started our first sprint we spent 258.5 hours on pre studies and planning. The total time spent on the entire project is 1674.5 hours.

### 6.1 Sprint 1

---

**Start:** January 31<sup>st</sup>

**End:** February 11<sup>th</sup> (13<sup>th</sup>)

**Hours spent:** 261.2

**Agenda:** The goal for sprint 1 is to get the main window designed and to implement the basics for making it runnable. We also want to get started on the implementation of the plugin list GUI. In addition we want to get the core functionality implemented for reading, editing and saving of the server list and the basics of the server dialog running.

Table 6.1: Sprint 1 backlog

ID	Description
SD1	<b>Server list - write/read to/from file</b> Go through and understand old code Refactor/improve where neccessary Write tests for reading and writing serverlist
SD2	<b>Dialog for server list and configuration of servers</b> GUI of server dialog Connect with the server-object from the list
PS1	<b>Find and import plugins</b> PluginLoader class
PS2	<b>Dialog for plugin configuration</b> Design Model

... continued on next page

ID	Description
MW1	<b>Port the old MainWindow from Qt3 to Qt4 with MVC</b> Design Implementation
MW3	<b>Widget to host plugins</b> Implement a QStackedWidget
MW2	<b>About, credits and license dialog.</b> Redesign Implementation
L1	<b>Locate/load files dynamically</b> Automatic building of the language selection in the menubar on startup Implement a dedicated language translation files handler
CS2	<b>Settings system (backend) that GUI can use</b> Global settings wrapper for QSettings
CS3	<b>Settings dialog (GUI)</b> Design Implementation
LO1	<b>New backend-system</b> Read up on Python's logging module Design + implementation
LO3	<b>Convert to new system</b> Replace old logging to use the new system

### 6.1.1 Review

Sprint 1 was our first real test of the project group. The group functioned very well from the beginning. Due to our pre-studies and the assignments of responsibilities, we got a good start in the project development during the sprint. We were able to finish most of the features in the sprint backlog, and we even got time to get started on extra features. The GUI for the main window was up and running halfway through the sprint. The development on this feature is discussed in more detail in section 7.1. The functionality for maintaining the application server list was also successfully implemented. This is discussed in more detail in section 7.3. The work with the plugin list turned out to be too dependent on the back-end part of the plugin support system, and the work on this feature was therefore continued in sprint 2 in section 6.2. In addition to the planned features, we got started on multilingual support for the application. We implemented most of this feature's core functionality during sprint 1. This feature is discussed in more detail in section 7.4.

The biggest problem encountered was at the end of the sprint, when all of the code parts were to be merged together. We concluded that the problems with merging most likely would become a bigger issue when our code base got bigger during future sprints. A decision was made to switch from SVN to Git for our repository and code version control system, because of the superior merge support in the Git system, compared to SVN.

## CHAPTER 6. SPRINT OVERVIEWS

---

### 6.2 Sprint 2

---

**Start:** February 14<sup>th</sup>

**End:** February 25<sup>th</sup> (27<sup>th</sup>)

**Hours spent:** 252

**Agenda:** We are starting implementation of the browser plugin and want to view contents on servers at the end of the sprint. We also want to start implementing the settings system, continue on the plugin system and start documenting how to make new plugins. The logging system should also be implemented in this sprint.

Table 6.2: Sprint 2 backlog

ID	Description
SD1	<b>Server list - write/read to/from file</b> Improving saving serverlist
SD2	<b>Dialog for server list and configuration of servers</b> Fix delete button focus problem
BP1	<b>View content on server</b> Tree model and view LDAPTreeItem-Family Context menu(right-click Reload Items
BP4	<b>Edit item</b> Using advanceObjectView to edit entries
BP6	<b>Set filter</b> Gui to specify the filter Implement so the filter is used Have the item show a filter is used
PS2	<b>Dialog for plugin configuration</b> Improve settings on plugins Reload plugins after new settings in MainWin
PS3	<b>Plugin toolbar</b> Convert old toolbar to QT4
MW1	<b>Port the old MainWindow from Qt3 to Qt4 with MVC</b> GUI to specify the limit Implement so the limit is used Have the item show a limit is used
MW3	<b>Widget to host plugins</b> Change between different plugins

... continued on next page

ID	Description
MW4	<b>Host the plugin toolbar</b> Functionality to go back and forth between plugins
LTC1	<b>Create tools for handling application resources</b> Implementing wrapper for various qt and PyQt tools target resources
CS2	<b>Settings system (backend) that GUI use</b> Switch from ConfigParser to QSettings
CS1	<b>Read/write to files</b> Use Qsetting for application settings On startup locate the user config
L02	<b>Loglevel filtering in the loggerwidget</b> Design the loggerwidget Connect with logger system so logs are shown
SD5	<b>Save-functionality (Ok, Save, Cancel)</b> Save internal state on <i>Save</i> -clicked Return the saved state on <i>Cancel</i> -clicked after save
Work planned, but not finished	
L	<b>Innstallation</b> Look into old installation, make a new for cross-plattform
L3	<b>Port old/create new QT translation files</b>

### 6.2.1 Review

In sprint 2 we became quite comfortable with the Qt-libraries and GUI design in Qt Designer. We discovered that Qt has more features than GUI, like QSettings, QtXML and Qt's resource system. In sprint 1 we were concerned about not finding platform-independent solutions, specially when looking at how Luma 2.4 takes care of setting and configuration files. The solution was already there for us in Qt 4, the `QSettings` class. This will be discuses more in section 7.2. Most of our work in sprint 2 was on the browser plugin and therefore we got more work done on the plugin than we had expected, more about the browser plugin in section 7.6. The plugin support system was continued and can be read about in section 7.5.



### 6.3 Sprint 3

---

**Start:** February 28<sup>th</sup>

**End:** March 11<sup>th</sup> (13<sup>th</sup>)

**Hours spent:** 129

**Agenda:** Continuing work on the browser plugin and begin on the entry-view showing entries on the LDAP servers. Improving the plugin system. Begin research on the template plugin, and start looking at ways to deploy and distribute the application.

Table 6.3: Sprint 3 backlog

ID	Description
MW4	<b>Host the plugin toolbar</b> Improve functionality to various plugins
PS2	<b>Dialog for plugin configuration</b> Make it work with corrupted settings
MW5	<b>Tabs and plugins</b> Design and planning of tabs
I2	<b>Write cross-plattform install script</b> Look at aviable deployment tools Create a basic installation script
O1	<b>Other</b> Using grc in browser plugin
BP1	<b>View content on server</b> Viewing LDAP objects
BP2	<b>Add items</b> Editing LDAP items
BP4	<b>Edit items</b> Deleting attributes from ldap objects
<b>Work planned, but not finished</b>	
T	<b>Template</b> Ask customer for more information about this plugin Study old implementation

#### 6.3.1 Review

Sprint 3 was a mix of new planning and improving work from sprint 2. The work on browser plugin was continued and we begun on the entry-view to show LDAP entries in the GUI. This is further described in section 7.6.2. We were done with the plugin-support-system half way through sprint 3, but decided to redo much of the design and this will be discussed in section 7.5.3. The deployment and distribution part was begun, this is shown in section 7.11.1. However, we did not have time to start on the template plugin.

## 6.4 Sprint 4

**Start:** March 14<sup>th</sup>

**End:** March 25<sup>th</sup> (27<sup>th</sup>)

**Hours spent:** 204.5

**Agenda:** Browser plugin should get the functionality to add and delete LDAPserver entries. We want to change the plugin system to have plugins in tabs so we can change between these without closing them. The installation and platform deployment should be continued. Since we did not start on the template plugin in sprint 3, we decide to start and get most of the template plugin working.

Table 6.4: Sprint 4 backlog

ID	Description
T1	<b>Host the plugin toolbar</b> Make the GUI for plugin and all dialogs
T2	<b>Read/write templitelist from/to file</b>
T3	<b>Basic functionlity</b> Add and delete templates Add and delete objectclasses Add and delete attributes Icons in attribute tableview
MW5	<b>Tabs for plugins</b> Implement tabs for plugins Make a welcome tab Improve icons
PS2	<b>Dialog for plugin configuration</b> Make tabs(about/settings)
S1	<b>Redesing UI(more space effcent and intuitive)</b>
S2	<b>Validate search filters</b> Implement basic filter validaton before the search opearation
S3	<b>Use search filters</b> Implement the search routine (sync/async)
I1	<b>Prepare the application filepath usage</b> Refactor codebase to use realive import statements(see: PEP 328)
CS4	<b>Generic settings wrapper for plugins</b> Implement a safe way for plugins to access the config file
L2	<b>Dynamic translation for all widgets</b> Construct a convention for defining strings not created in the Qt4 Designer, that should be translated

... continued on next page

## CHAPTER 6. SPRINT OVERVIEWS

---

ID	Description
BP8	<b>Export items to file</b> Implement right-click menu with options for exporting selections Implement a export dialog where user can approve the selected items
BP10	<b>Custom HTML views</b> HTML-tempalte views
SD4	<b>Usability improvements(serverdialog</b> Confirmation on cancel Choose SSL-port automatically
BP5	<b>Remove items</b> Selectabe from tree or edit-view Confirm deletion with user Update model

### 6.4.1 Review

Out of the two functionalities we planned to add to browser only delete was implemented, but instead of add, export was implemented. More about the browser plugin in sprint 4 in section 7.6.3. The new design plans for the plugin support from sprint 3 were all implemented in sprint 4 and is described in section 7.5.4. We did not get more done on the installation and deployment part of the project this sprint. Most of the functionality of the template plugin was done this sprint, more is written about this in section 7.7.1. We also got time to get started on the search plugin in the end of this sprint and got a head start for the next sprint, more about the search plugin in section 7.8.

## 6.5 Sprint 5

**Start:** March 28<sup>th</sup>

**End:** April 8<sup>th</sup> (13<sup>th</sup>)

**Hours spent:** 195

**Agenda:** In this sprint we want to get the most important parts of the code finished, but mostly we will concentrate on the report that is going to be delivered. We want to finish the three plugins, browser, template and search, at the end of this sprint.

Table 6.5: Sprint 5 backlog

ID	Description
T3	<b>Basic functionality</b> Add and delete objectclasses Add and delete attributes
T4	<b>Attributes changed on removal and adding of objectclasses</b> Automatically add must attributes when objectclass is added Automatically remove attributes that is not supported
S4	<b>Handle errors and exceptions(User feedback)</b> Notify the user when things go bad
S5	<b>Retreive search results</b> Implement the search result handler
S6	<b>Display search results</b> Implement the model view for search results
S7	<b>Build search filters</b> Implement(improve) the filter wizard from Luma 2.4
S8	<b>Save search filters</b> Implement a way to save filters to be used later
S9	<b>Remember search settings</b> Implement a setting widget for the search plugin
S10	<b>Additional filtering on retrieved search results</b> Implement a filter box for the search result
BP2	<b>Add items</b> Adding entry using selection as template
BP10	<b>Custom html views</b> Selection default view

### 6.5.1 Review

In this sprint we focused mostly on documentation and refining the report, but still got much done in the coding part of the project. The browser plugin finally got the ability to add items and some major work in the back-end part, more on this in section 7.6.4.

## CHAPTER 6. SPRINT OVERVIEWS

---

The template plugin got its last functionalities in sprint 5 described in section 7.7.2 Also the search plugin got finished and had all its functionalities at the end of sprint 5 and is talked about in section 7.8.2.

## 6.6 Sprint 6

**Start:** April 26<sup>th</sup>

**End:** May 15<sup>th</sup> (13<sup>th</sup>)

**Hours spent:** 373,75

**Agenda:** This is the final sprint and everything should be done.

Table 6.6: Sprint 6 backlog

ID	Description
LD1	<b>Setup documentation tools for the project</b> Choose tools that ease the documentaio process Capabel of generating the document in a number of formats
LD2	<b>Write userguide</b> Document various aspects of the application relevant to the user
LD3	<b>Write/document installation process</b> Document requirements and library dependencies Document various ways to install and run the application
SD3	<b>“Test connection”-functionality“</b> Take currently selected server and tr to bind it, report status
	<b>Debugging</b> Everything
T5	<b>Non-must attributes can be changed to custom-must</b>
BP4	<b>Edit items</b> Different editor for binary/password/rdn Add attribute wizard
BP2	<b>Add items</b> Add entry using a template object
B1	<b>Rewrite backend to separate out GUI-code</b> Refactor classes
B2	<b>Rework threaing to avoid segfault on closed plugins</b> Retain references while threads running
B3	<b>Provide async API to LumaConnection(Qt signal + slots</b> Use QThreads for threading and implement the needed signals
B4	<b>Ability to set temporary password for servers</b> Add menu item pluss input-dialog
I3	<b>Document installation process for Linux distributions</b>
Work planned, but not finished	
I4	<b>Create frozen application bundles for Windows and Mac OSX</b>

### 6.6.1 Review

By this sprint we had discovered a weakness in the back-end that potentially cause the program to crash. This was remedied and the opportunity to change the back-end was also used to add some new plugin developer features. In addition there were small changes and additions to the browser and the search plugins, these are mentioned respectively in section 7.6.5 and section 7.8.3.

In addition there were changes in the template plugin, server dialog and installation, written about in sections 7.7.3, 7.3.4 and 7.11.2.

# 7

## Development

### 7.1 Main Window

---

Implementing and designing the application main window was one of the main priorities for sprint 1. The design of the main window in Luma 2.4 included a few legacy Qt 3 widgets, which we had to get rid off. As one of the project requirements is to develop a cross-platform application, we have spent some time studying various best practices for the different platforms. Most of the result from this study can be found in appendix C.

#### 7.1.1 Sprint 1

##### Design

The main architecture of the main window was something we tried to bring forth and take advantage of in the new implementation. The main window consists of several widgets which is explained below, and illustrated in the mock-up-sketch we made at the beginning of sprint 1, see figure 7.1.

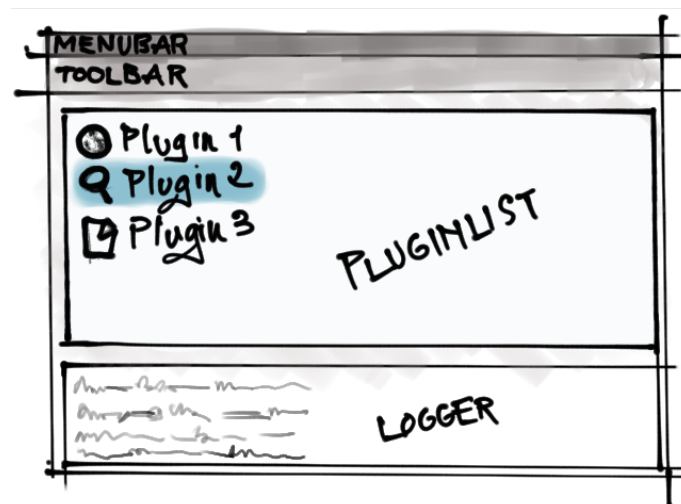


Figure 7.1: Mockup sketch for the application main window



- **Menu bar**  
A traditional application menu bar, providing common actions related to the application.
- **Plugin toolbar**  
A toolbar which provides easy access to installed plugins. Selecting a plugin, from the plugin toolbar, loads the selected plugin into the main view.
- **Main view**  
A stackable main view. This is where the each loaded plugin lives.
- **Logger window**  
This widget displays different kind of categorized messages that the application produces. The available categories is *error* messages, *debug* messages, and general application *info*.

### Implementation

**Menu bar** The original menu bar had a few shortcomings, as explained in chapter 3. In sprint 1 we tried to work out and fix these. As a result, the menu bar is completely restructured, with menu actions grouped together in a more logic and intuitive way. The result of the menu bar implementation after sprint 1 is illustrated in figure 7.2.

The new menu bar structure tries to follow best practices as described in the Human Interface Guidelines for the platforms running the X-Window system [9] [13] and the Windows platform [26]. Integrating a menu bar following the Mac OS Human Interface Guidelines, is something we eventually will postpone to a later sprint.

File	Edit	Help
<input checked="" type="checkbox"/> Show logger    Ctrl+L	Server List    Ctrl+Shift+S	About Luma    F12
Quit    Ctrl+Q	Reload Plugins    F5	
	Configure Plugins	
	Language    >	
	Settings	

Figure 7.2: The menu bar structure after sprint 1

**Main view** The main view contains a `QStackedWidget`, which is a Qt Widget designed to hold multiple child widgets, while only one widget being visible at once. Luma 2.4 used a Qt 3 specific version of this widget, and we needed to rewrite most of the code. The `QStackedWidget` will hold the plugin list and all open plugins.

**Plugin toolbar and logger window** Because the plugin toolbar and the logger window in Luma 2.4 were using mostly Qt 4 compatible widgets, the design and implementation of the GUI for these widgets are pretty much the same. The underlying code, however, introduces some improved functionality. Especially the code handling the logging is improved and discussed in section 7.9.

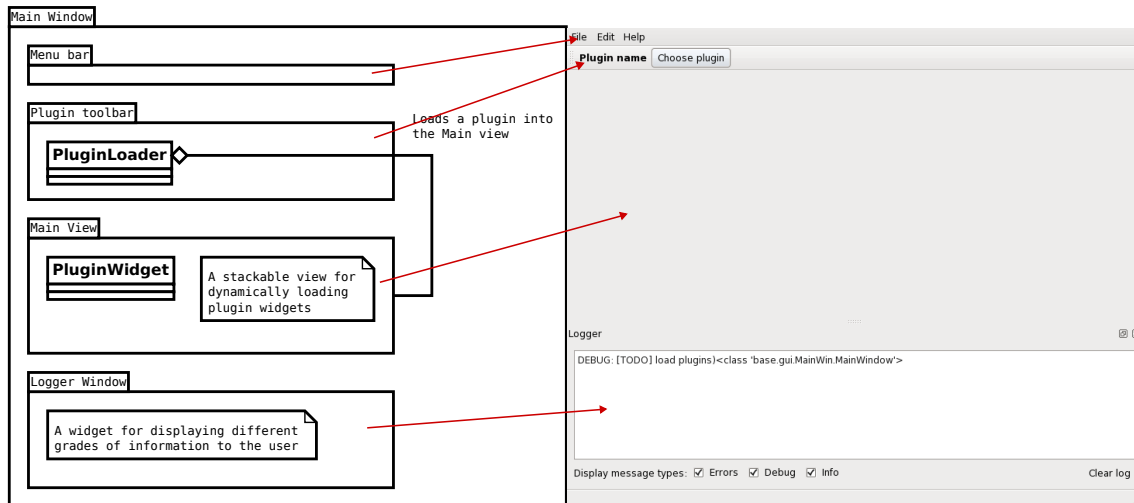


Figure 7.3: Conceptual architecture compared to actual implementation

### 7.1.2 Sprint 4

#### Design

**Main view** As a result of the changes and improvements being done on the application's plugin system, as described in section 7.5.4, we ended up changing the design of the application main view. This was primarily due to issues with memory usage and garbage collection of multiple running instances of plugins. A mock-up-sketch, illustrating the new tabbed layout for the main view is shown in figure 7.4.

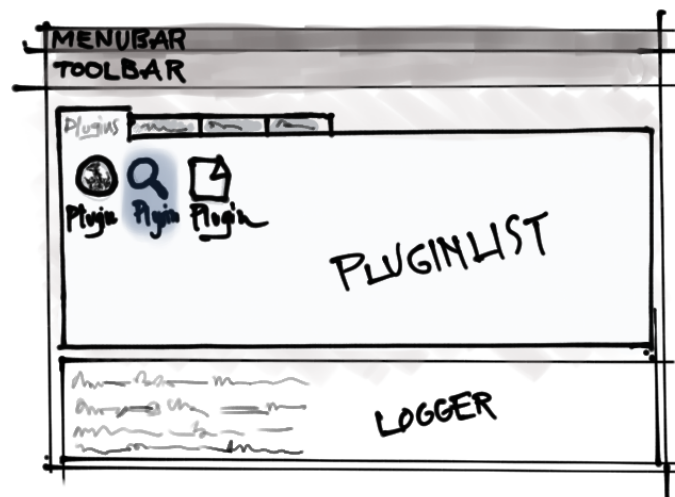


Figure 7.4: Mockup sketch for the application main window with tabs to handle various plugin views.

### Implementation

**Menu bar** During sprint 4 we managed to improve the layout and implementation of the application menu bar. The menu bar, seen in figure 7.5, now contains more menu action, enabling the user to control the main windows appearance better. We also managed to integrate the menu bar on Mac OS, shown in figure 7.6. This menu is following the best practices defined in the *Apple Human Interface Guidelines*[4]. Note however, that the application menu is displaying **Python** instead of **Luma**. In order to get the actual application name displayed, Luma must be deployed and installed as an Mac OS compatible *application bundle*.

File	Edit	View	Help
Quit    Ctrl+Q	Server List    Ctrl+Shift+S	Show Plugin List    Ctrl+P	About Luma    F12
	Reload Plugins    F5	Show Welcome Tab    Ctrl+Shift+W	
	Configure Plugins	<input checked="" type="checkbox"/> Statusbar	
	Language >	<input type="checkbox"/> Logger Window    Ctrl+L	
	Settings	<input type="checkbox"/> Fullscreen    F11	

Figure 7.5: The menu bar structure after sprint 4.

Python	Rediger	Vis
About Python	Server listen    ⌘S	Vis plugin listen    ⌘P
Preferences...    ⌘,	Reload plugins    F5	Vis velkomst tab    ⌘W
Services ▶	Språk ▶	<input checked="" type="checkbox"/> Statuslinje
Hide Python    ⌘H	Innstillinger	<input checked="" type="checkbox"/> Log vindu    ⌘L
Hide Others    ⌘⇧H		Fullskjerm    F11
Show All		
Quit Python    ⌘Q		

Figure 7.6: The menu bar structure as shown when running on Mac OSX.

**Main window** At the end of sprint 4 the application main window was near completion. The result is shown in figure 7.7. Compared to the main window of Luma 2.4, Luma 3.0 introduces a number of improvements. The most eye catching difference is the use of tabs to manage all active external widgets. This solutions improves and enable us to control the memory usage per active plugin.

### 7.1.3 Sprint 5

#### Implementation

**Main Window** In sprint 5 we added the *Welcome tab*. This is shown when Luma 3.0 is started for the first time. Figure 7.8 show what this tab looks like. The main purpose for *Welcome tab* was to provide new Luma users with a short *quickstart-guide*.

## 7.1. MAIN WINDOW

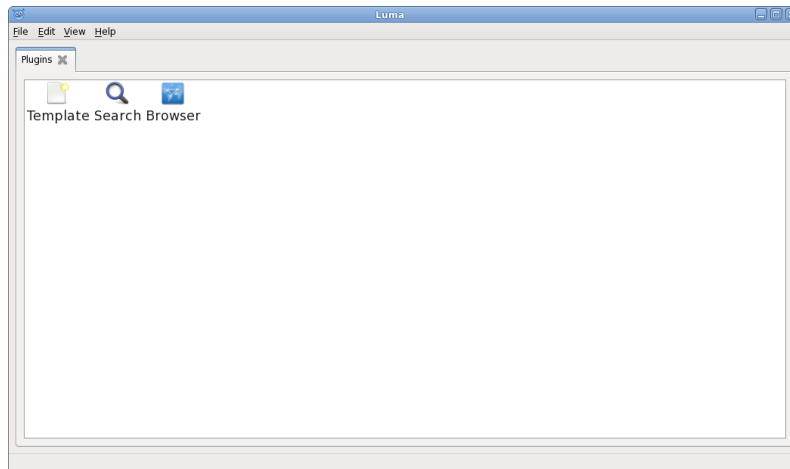


Figure 7.7: The main window after sprint 4

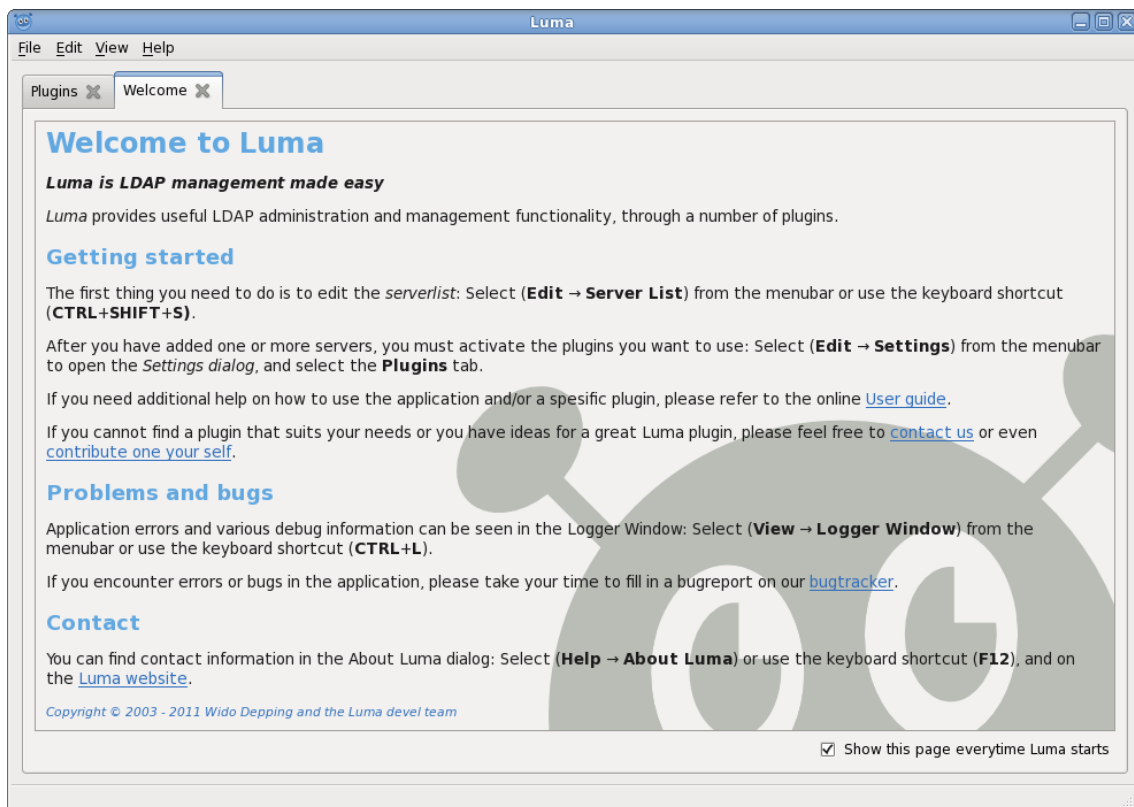


Figure 7.8: The Luma 3.0 Welcome tab

7.1.4 Sprint 6

Implementation

**Menubar** During sprint 6 we mostly examined the UI design, looking for missing key bindings, keyboard shortcuts, tab ordering, and other *none-crucial* aspects of the application. In order to improve the usability of Luma 3.0 , we ended up removing some of the menu bar actions. These were removed either because they were no longer needed, or because they duplicated some other action in the menu bar. For instance, we removed both the *Edit* → *Reload Plugins* and the *Edit* → *Configure Plugins*. The first because the application no longer needed to reload plugins, and the latter because the same dialog was accessible through the *Edit* → *Settings* menu bar action. We also updated the names for the various menu bar actions, resulting in the final menu bar implementation shown in figure 7.9.

File		Edit		View		Help	
Quit	Ctrl+Q	Server List	Ctrl+Shift+S	Plugin List	Ctrl+P	About Luma	F12
		Temporary Password		Welcome Tab	Ctrl+Shift+W		
		Language	>	<input checked="" type="checkbox"/> Statusbar			
		Settings		<input type="checkbox"/> Logger Window	Ctrl+L		
				<input type="checkbox"/> Fullscreen	F11		

Figure 7.9: The final menu bar.

## 7.2 Settings system

As Luma 2.4 was only running on UNIX-like systems, there was no need for a cross-platform back-end system, handling settings and configuration files. When developing for multiple platforms, we need a robust way to store and retrieve application settings and configurations in an platform independent way. As result we looked into various ways and tools specialized for configuration file parsing, configuration file storing, etc.

Python includes a default configuration parser module (`ConfigParser`), which also was used in Luma 2.4. This module is fairly straight forward and intuitive to use, but a lot of work still remains on explicitly handling the file paths on the different platforms. The Qt application framework also includes its own settings system, known as `QSettings`, which is part of the `QtCore` library. `QSettings` provides persistent platform-independent application settings. What this means, is that Qt is handling all file path issues, ensuring that platform best-practices is followed on the running platform.

### 7.2.1 Sprint 1

#### Design

In sprint 1 we started designing a brand new settings dialog for the application. A graphical tool for configuring the application behavior was something that was not present in Luma 2.4. We figured that the ability to access and control various application settings through a graphical dialog would be very convenient. This is also very common in modern desktop applications. We came up with a few ideas for how this settings dialog should look like, with two of the versions illustrated in figure 7.10 and 7.11.



Figure 7.10: Settings dialog mockup 1.

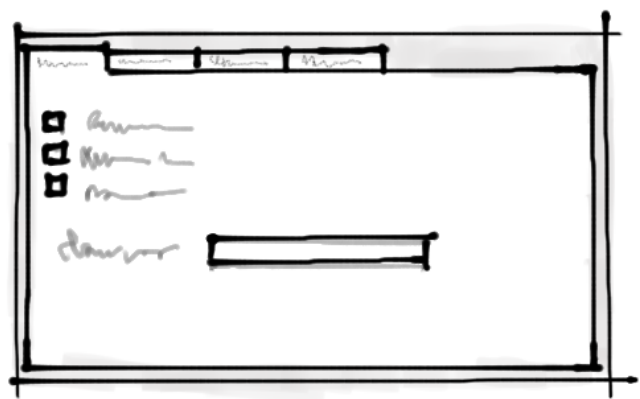


Figure 7.11: Settings dialog mockup 2, with tabbed layout.

### Implementation

We decided to implement the tabbed layout for the settings dialog. This meant that we also would be able to include the planned plugin configuration into this dialog, with its own dedicated plugins tab. At the end of sprint 1 we were able to finish the implementation of the skeleton GUI for the tabbed settings dialog. This version of the implementation is illustrated in figure 7.12. The implementation of the back-end functionality for the application settings system was postponed to sprint 2.

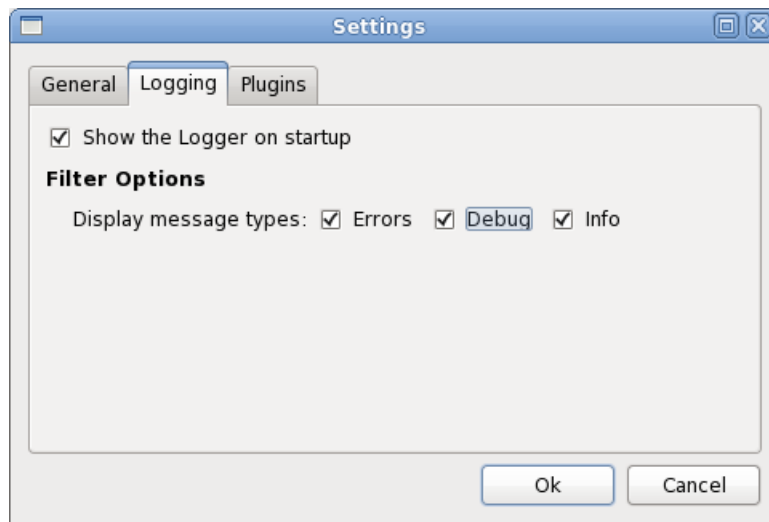


Figure 7.12: The new settings dialog at the end of sprint 1.

### 7.2.2 Sprint 2

In sprint 2 we implemented an extension of the `QSettings` class, which we simply called `Settings`, as illustrated in figure 7.13. The reason for doing it this way, instead of using the `QSettings` class directly, is to provide a transparent way to save and retrieve application settings for all parts of the application. The benefits are that settings sections and settings keys are managed in one place, leaving the rest of the application in no need of handling this functionality.

On UNIX and UNIX-like systems, the settings are saved in simple text-based ini-files, which are common practice on such systems. The location of the settings files will, on these systems, be `~/.config/luma/`. An example of the ini-style syntax is shown here:

```
[section1]
key1=valueA
key2=valueB

[section2]
key/with/multiple/levels = valueC
```

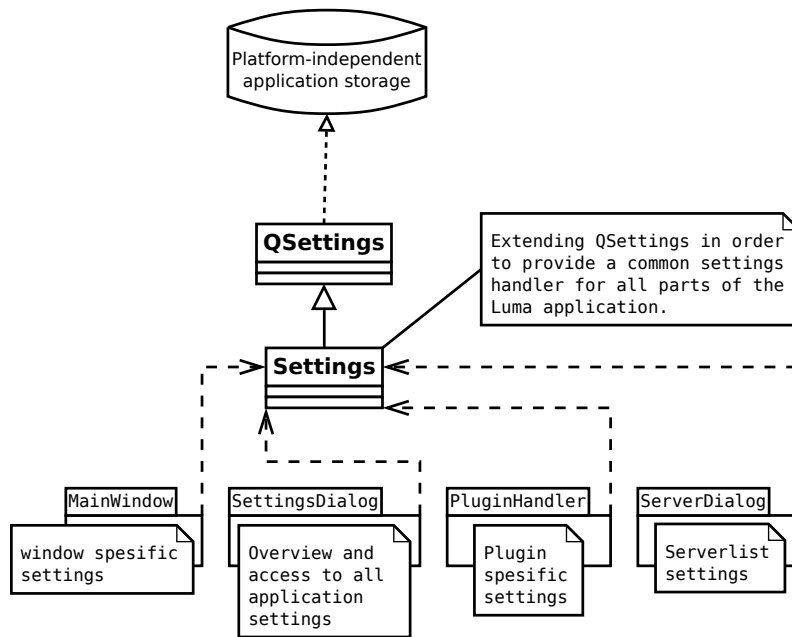


Figure 7.13: The new Luma settings system architecture

On Windows QSettings will write settings to the Windows-registry, and on Mac OSX the settings is saved in `.plist` files, located in `~/Library/Application Support/luma/`. We also can choose to store settings in ini-files on Windows and Mac OSX, as well as explicitly define the location to store the settings.

### 7.2.3 Sprint 5

#### Design

In order to simplify the handling of saving and retrieving application settings, we decided to change the initial design and implementation of the settings dialog. We experienced with the previous design was that it was confusing. Virtually all application settings were accessible from both the main window (the menu bar) as well as from the settings dialog. We therefore decided to put some constraints on what kind of settings we would provide in the settings dialog.

Functions the user could (more easily) control through the main window (primarily through options in the menu bar) would no longer be provided in the settings dialog. We only include application settings that the current Luma session did not directly depend upon. This change in design, meant we could keep the code cleaner and the user less confused.



### Implementation

In sprint 5 we finished the settings system implementation for plugins. We created a `PluginSettings` class that extended the `QSettings` class, much like the main `Settings` class, used by the core application. The idea was to create a transparent and persistent way for each plugin to access its own settings sections in the Luma configuration file.

```
class PluginSettings(QSettings):
    def __init__(self, pluginName):
        # pluginName is used as the prefix key in the settings file.

    def pluginValue(self, key, default):
        # Gets the value for the plugin setting defined by key.
        # default is the fallback value.

    def setPluginValue(self, key, value):
        # Sets a new value for the plugin setting defined my key.
```

In order to get each plugin to control reading and writing of its own settings, we took advantage of the Qt signal system. We implemented a custom signal, `onSettingsChanged`, which is emitted when the user clicks the OK button in the settings dialog. For this to work, all plugins that provide settings options need to provide a `writeSettings` method in its Settings widget. This method is then connected with the signal, and is called when the signal is emitted. Figure 7.14 shows the settings options provided by the search plugin.

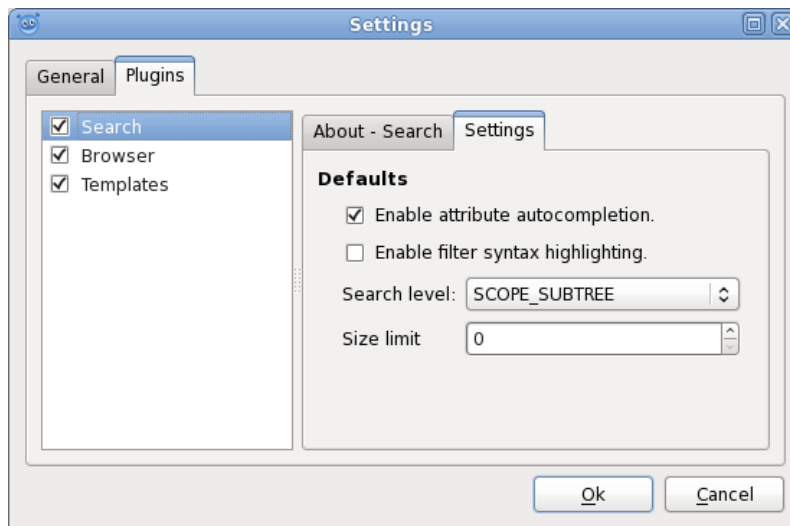


Figure 7.14: The settingsdialog with plugin configuration.

## 7.3 Server dialog

---

The server dialog (editor) was one of the first features we worked on. This was intentional for a couple of reasons: it would serve as a nice introduction to programming in Python and PyQt/Qt (and related tools like Qt Designer); and it would also make a soft introduction to Qt's MVC-framework, described in greater detail in section 7.6. It was also confined to one window, and thus we could focus on this feature alone. In addition, finalizing it early would be useful later in the project when we would need to manage and use different servers, e.g. implementing the browser plugin.

In terms of differences from Luma 2.4, we chose to use a different design inspired by *Vegar Westerlund*. This design had been added to the development-repository of Luma 3.0, but not incorporated into the stable branch. We felt this design was functionally better and easier to use than the old design.

The dialog was, as planned, functionally finished in sprint 1, but small tweaks were done in later sprints as well.

### 7.3.1 Sprint 1

The internal representation of servers in Luma 3.0 is with a **ServerObject**. This object defines all aspects of the servers needed for connecting to them, for instance an identifying name, hostname, port, whether to use *SSL* or *TLS*, etc. All **ServerObject** instances are then stored in a **ServerList** object which takes care of the persistent storage of the servers through the use of XML-files on disk.

As Luma 2.4 did not use the MVC-framework, we needed to redesign it completely. This meant creating a model by subclassing the **QAbstractItemModel**, as defined by the Qt 4 MVC framework (or in this case a partial implementation of the **QAbstractTableModel**), which would enable a view to access and save changed server data about the servers. A custom view would also have to be designed to display the data appropriately and provide the right editing-facilities for the different data-types.

The design of the view was done with the Qt 4 Designer program mentioned in section 2.3. Based on the available editor-widgets, we had to provide a mapping from the different columns (data-fields) in the model to the correct editor-widget, and also a list which displayed all the rows (the different servers) where the user could select which one to view or edit.

The mapping was done in two ways: For the majority of the fields we could use the Qt-provided **QDataWidgetMapper** to automatically fill out the editor-widgets with data from the model and save it back to it when requested. This worked for virtually all the fields except for the server's list of *baseDNs* the server had. For this case, we had to provide a custom delegate which defined how the model-data (in this case a Python-list) should be used in the view. The solution was to have the delegate to fill out a **QListWidget** with one item for each entry in the Python-list, and when saving take each item in the **QListWidget** and convert it back to a Python-list. As this was not and could not be done through the **QDataWidgetMapper** we had to write additional code which did the necessary conversion when the opened server changed (e.g. by the user selecting a different one).

For ease we found it necessary to change the format and internal representation of fields in the `ServerObject`. Luma 2.4 used a format where, for instance, the encryption method for the server was represented by the strings *Unencrypted*, *SSL* or *TLS*. This was changed to use a variant of enums (Python itself does not provide enums) in the form of class-variables, making it possible to represent this information on the form:

```
serverObject.encryptionMethod = ServerEncryptionMethod.SSL
```

or if unencrypted:

```
serverObject.encryptionMethod = ServerEncryptionMethod.Unencrypted
```

and check it with:

```
if serverObject.encryptionMethod == ServerEncryptionMethod.SSL:
    # Do something
elif serverObject.encryptionMethod == ServerEncryptionMethod.TLS:
    # Do something else
```

Also converted to this form was the authentication-method (`ServerAuthMethod.OPT`, where `OPT` could be `Simple`, `SASL_CRAM_MD5`, etc), and the decision whether to check the certificate (`ServerCheckCertificate.OPT` where `OPT` could be `Never`, `Allow`, `Try`, etc).

This change also meant defining a new format for saving the servers to file. We also created conversion-methods so that files using the previous format would be automatically converted.

### 7.3.2 Sprint 2

The server dialog in Luma 2.4 had the ability of reverting to a previously saved state while the dialog was open through the use of a *Save* button. The user could then do some changes, click save, do some erroneous changes and click cancel. The cancel would cause the erroneous changes to be canceled.

### 7.3.3 Sprint 4

The customer had expressed a wish for a confirmation-dialog on most potentially work/data-losing actions. One example is the "cancel"-button in the server dialog which disregards all changes made to servers since the saving. Up till now the confirmation-dialog was always displayed when cancel was clicked. We felt this was annoying when we opened the dialog just to check the settings and then immediately clicked cancel (to make sure we did not accidentally change something which was then saved by clicking *OK*).

We made some changes to the dialog to make it aware of changes and only display the confirmation when a cancellation would actually lose changes. This was done through listening to the model's `dataChanged`-signal and ensuring that the signal was only emitted when actual changes were made (this had to be checked by comparison).

Another change was made for usability-reasons. When using SSL-encryption, the port is often different from the standard LDAP-port. However, the user may forget to change

the port when setting up to connect with encryption, and get a not always so clear error message from the LDAP-server. In order to combat this, we changed the dialog to – when the user selected SSL – check if the port used is the standard LDAP-port and then ask the user if this should be changed to the common SSL-port instead. An alternative to this, which is done in other LDAP-clients, is to change the port automatically without notifying the user first, but we felt this would be a bad way of handling it as the user might very-well have a reason for using the port.

### 7.3.4 Sprint 6

We had noticed that other LDAP-application had the ability to test the connection while configuring the connection to the server. Mentioning this to the customer, he acknowledged that this would be a good addition to Luma. This implemented and provided no major challenge. It only was to take the currently selected server's current configuration and attempt a bind with it.

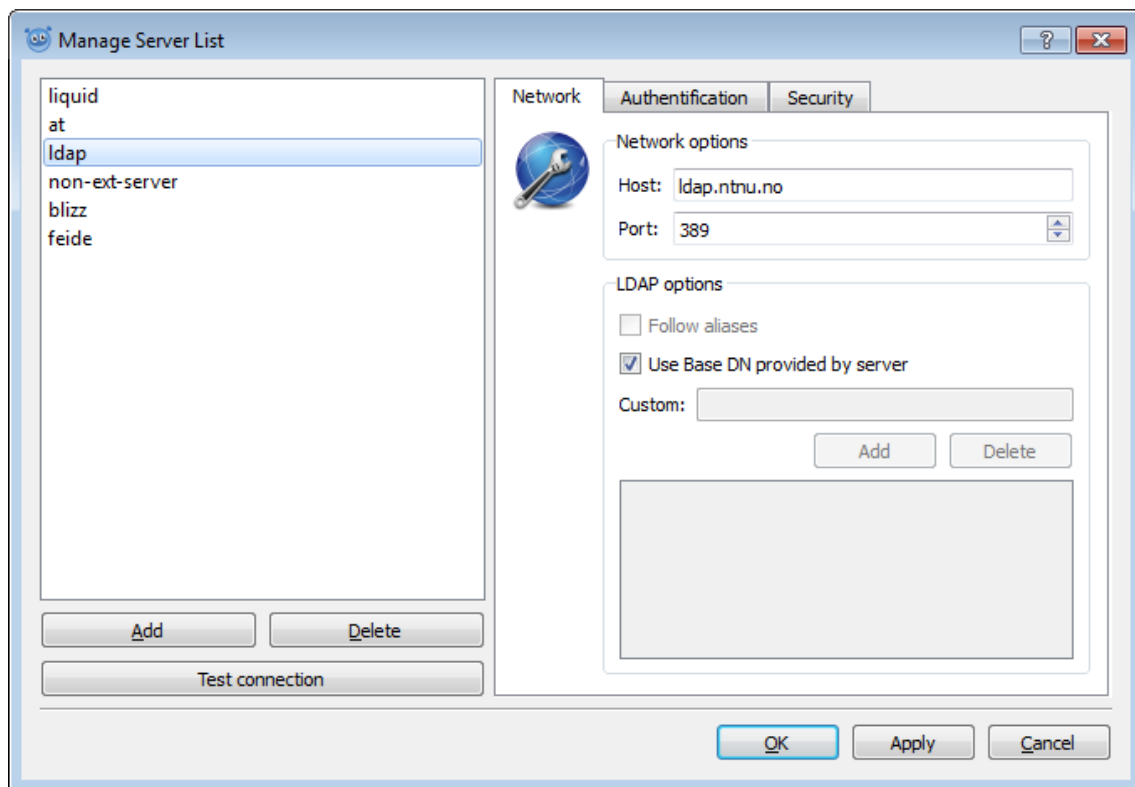


Figure 7.15: The serverdialog after sprint 6

### 7.4 Internationalization

---

As mentioned in chapter 3, the Luma 2.4 translation implementation included minor shortcomings. Mostly the lack of runtime translation without changing the application state was missing. *Translation at runtime without change of application state*, was something we had as a goal for the new implementation. Luma 2.4 was translated into 9 different languages <sup>1</sup>, which was created by community contributors across the world.

Providing the actual translations for Luma 3.0 is not something we are supposed to do, but we will facilitate the translation process for future contributors. This means we need to implement a robust back-end system, that conform to our own and our customer's goals.

#### 7.4.1 Sprint 1

No big problems occurred in the start of sprint 1 and we decided to start looking at the application translation system at the end of the sprint. We managed to implemented support for locating and loading translation files dynamically. We also created a language entry in the menu bar, which was built using available translation files when starting Luma.

One of the issues we encountered during implementation of this functionality was how to locate the path to the translation files. Our solution was to facilitate the *Singleton* pattern. We were not to happy with this solution because, by using the *Singleton* pattern, we also introduced shared state in the application, which is something that usually should be avoided <sup>2</sup>. We, none the less, decided to go for this solution at this point because it enabled us to start testing how dynamic translation of the application was working. The following Python code is a simplification of our *Singleton* implementation, illustrating how we were able to set the translation path on startup, and retrieve it with one instance of the class through out the application.

```
class Paths(object):
    """The Paths class implements the Singleton pattern."""

    def __new__(cls, *args, **kwargs):
        """Checks if there exists an instance of this class."""
        if not cls.__instance:
            cls.__instance = super(Paths, cls).__new__(cls, *args, **kwargs)
        return cls.__instance

    def i18nPath(self):
        """Returns the path to the translation files."""

    def i18nPath(self, path):
```

---

<sup>1</sup>Including: English, Czech, French, German, Norwegian, Portuguese, Russian, Spanish, Swedish

<sup>2</sup>Applications with shared state can often be very hard to debug, if something suddenly does not work like expected.

```
"""Sets the path to the translation files."""
```

### 7.4.2 Sprint 2

#### Implementation

In sprint 2 we improved the method for locating the application translation files. We implemented a `LanguageHandler`, by which we enabled parts of the application to both retrieve the path to the translation files, as well as providing the correct file based on language iso codes <sup>3</sup>.

```
class LanguageHandler(object):

    def availableLanguages(self):
        """Returns the available application language translations in a
        dictionary with language code as key and language names as value."""

    def getQmFile(self, isoCode=''):
        """Returns the translation file for the provided iso code."""
```

The `getQmFile` method refers to the Qt file extensions for compiled translation files (`.qm`).

### 7.4.3 Sprint 3

In sprint 3 we designed and implemented a system that enabled external widgets (like plugins) to get support for dynamic translation. Plugins that wish to get this support should implement the `changeEvent` method, and listen for the `LanguageChange` event. By doing this the widget should then be able to explicitly call its own `retranslateUi` method to achieve dynamic translation at run time.

```
def changeEvent(self, event):
    """This event is generated when a new translator is loaded.
    """
    if QEvent.LanguageChange == event.type():
        self.retranslateUi(self)

def retranslateUi(self):
    """Explicitly translate the gui strings."""
    self.someWidget.setText(QApplication.translate('some text'))
```

### 7.4.4 Sprint 6

In sprint 6 we further improved the internationalization support in Luma 3.0 . We extended the support for country codes and language codes in the filename for the translation files.

---

<sup>3</sup>ISO 639-2: <http://www.loc.gov/standards/iso639-2/>.

## CHAPTER 7. DEVELOPMENT

---

This convention was heavily inspired by the *GNU* internationalization system, *gettext*, and is also the convention for locales supported by the Qt 4 framework. The new convention for translation file names, states that the filename should start with the `luma_` prefix followed by a two letter lowercase *ISO 639* language code, and a optional two-letter uppercase ISO 3166 country code. Both the following translation files contains valid names:

```
luma_en.ts
luma_en_US.ts
```

This meant that we now could support translations for countries with more than one language, or languages with multiple countries (each with its own variants to the language). For example:

```
luma_nb_NO.ts # For support for Norwegian Bokmål
luma_nn_NO.ts # For support for Norwegian Nynorsk

luma.pt_BR.ts # For support for Portugese as spoken in Brazil
luma.pt_PT.ts # For support for Portugese as spoken in Portugal
```

---

<sup>3</sup>*gettext* is a internationalization and localization library for writing multilingual applications: <http://www.gnu.org/software/gettext/>.

<sup>3</sup>ISO 3166: [http://www.iso.org/iso/country\\_codes/iso\\_3166\\_code\\_lists/](http://www.iso.org/iso/country_codes/iso_3166_code_lists/).

## 7.5 Plugin system

---

The plugin system is described in chapter 3.4. The development started with reimplementing of the existing plugin system. The new system had to include all the features and it had to comply with the MVC pattern. Into our third sprint, we started planning new and improved features as well. In total, we used four sprints on the plugin system.

### 7.5.1 Sprint 1

The work on plugins in our first sprint was mostly to study the implementation of plugins in Luma 2.4. Most of our work and what we discovered can be found under section 3.4. We started working on the back-end, improving the `PluginLoader` module. With it, we created a class `PluginObject`, to contain the information about a plugin that was loaded with the `PluginLoader`.

After improving the `PluginLoader` module, we started on the part where the user choose the plugins he wants to have available, and configure them. We made a dialog called the `PluginSettingsDialog`, and a model for the list-view that it uses; `PluginSettingsModel`. The actual GUI that the user sees, is the same as in the old Luma 2.4, but now we use a proper MVC pattern.

#### The reimplementing of the `PluginLoader`

The `PluginLoader` is responsible for finding all folders under the plugin-folder, and check if they are proper Luma plugins. If they are, it will load those plugins with use of Python's `imp` module and the functions `find_module` and `load_module`. The new `PluginLoader` module does the same as the old one, but some changes have been done:

**Environment** We thought it was bad practice to have several modules depending on the `environment` module, just for the purpose of using a few of its attributes. The new implementation does not make use of the `environment` at all. Instead we either find the attribute, or require it in the constructor. The attributes that previously were previously used from the `environment` module included:

- `environment.lumaInstallationPrefix`  
This was the path where Luma was installed on the users computer. We now use `QSettings`, as discussed in section 7.2, for storing various needed path locations.
- `environment.logMessage`  
This was used for logging in Luma 2.4. We now use the `logging` module, included in the Python standard library.

**Private variables and properties** The Python 2.6 release included its own feature for private variables and encapsulated access (Python properties). We have taken advantage of this feature in our reimplementing, and is now able to have better control over object variable access. In the old code you had to call the method `importPluginMetas`



(`pluginsToLoad = []`) if you wanted the `PluginLoader` to load plugins a second time (first time being when you initialize it). Then call the attribute for plugins like this: `pluginloader.PLUGINS`. With the use of properties, we encapsulate the class and make all methods private, and you only need to call the attribute for plugins like this:

```
pluginloader.plugins
```

The caller does not need to trigger a method to reload all plugins, it is triggered in the property of the `plugins` attribute if needed.

**PluginObject** This is just a minor change. We made a class containing all attributes that before were put into a rather complicated dictionary of arrays, to keep the information about a plugin. The `plugins` attribute now returns a list of these `PluginObject` objects, instead of a dictionary.

### The reimplementation of configuring plugins

In Luma 2.4 the `PluginLoaderGUI` was the dialog for choosing available plugins (found in the plugin-folder by the `PluginLoader` in the back-end) and configure them. We changed this dialog's name to `PluginSettings` and it inherits the `QDialog` and GUI module `PluginSettingsDialog`. More had to be done than just changing the GUI libraries from Qt 3 to Qt 4. We also had to implement the MVC pattern. The GUI will no longer call the `PluginLoader` in the back-end, but have a model `PluginSettingsModel` for its `QListView` that does this for it. The model will create its own items for the list-view based on what `PluginLoader` returns. Instead of using the configuration module given from `environment` that was used in Luma 2.4, we use the `QSettings` for saving the choices made by the user. These choices will later on be used when providing the list of available plugins that the user can load from a widget in the main-window.

#### 7.5.2 Sprint 2

We thought the further work on plugins would be straightforward. However this was not the case. We spent a lot of time discussing whether to reimplement more than just the new Qt libraries and MVC patterns.

In section 3.4 one can read about the old implementation of plugins. We considered an alternative solution where all plugins inherit from one Plugin parent-class. This class would have methods and fields, that the plugins should override. In this way we drop looking for attributes in `__init__.py`, and have an entry point in a `plugin-name.py` class that inherits the Plugin parent-class. The host application being Luma should have an interface to all its services that plugins could use, through this Plugin parent-class. The host application from a plugin manager will be able to use the plugins through a plugin interface, also given from the Plugin parent-class. So the Plugin parent-class will be a link, providing the plugins with a service interface, and the host applications with a plugin interface. In this way, a developer writing a plugin does not need to look around in Luma and collect all its services from different modules spread around the application, but find them all gathered from one interface. We also tried to make the application more

encapsulated, but this would require a lot of work, since Python does not support in such strict encapsulation as can be seen in languages as Java and C#. Thus, we decided not to implement application encapsulation.

### The implementation

In our first sprint we implemented the `PluginSettingsDialog`. We continued working on it, and made it work properly. The plugins that the user chooses in the dialog, were to be shown in the main-window. Luma 2.4 does this by creating a list-view directly into the main-window. The new implementation places a widget with a model for the list-view it uses: `PluginListWidget` and `PluginListWidgetModel` found under the *util* folder. In this way we implement the MVC pattern. If a plugin was set to load (in `QSettings` as mentioned before) in the `PluginSettingsDialog`, the model will create an item for the list-view of that plugin, with its corresponding icon. If a user clicks on this item, the main-window will load the plugin-widget on top of its widget-stack. If a user wants to go back to show all available plugins again, the `PluginListWidget` will be set back on top of the widget-stack of the main-window again.

The not so good part of the implementation, that comes from Luma 2.4 that we have not changed is the plugin toolbar. This should have been required from the `__init__.py` file from the plugin-folder. Instead, it is to be found inside the plugin-widget, so one has to assume the author has implemented this, instead of explicitly require this when loading the plugin from the back-end module `PluginLoader`.

### 7.5.3 Sprint 3

Most of the work on plugins in sprint 3 was implementing the plugin system. This means loading and unloading plugins, and the plugin toolbar.

#### Implementing plugin system as in Luma 2.4

The main window hosts a `QStackedWidget`, that contains a stack of widgets, but only one these widgets are visible at a time. The stack will typically contain a widget with the list of plugins to load, and the loaded plugins. The widget that contains this list is called `PluginListWidget`. When a plugin in that list is clicked, the plugin will be loaded into the main-window's `QStackedWidget`. The `PluginListWidget` will then be put behind in the stack and the plugin's widget on top. The plugin may take use of the toolbar. With the use of the plugin toolbar, one can go back again to show the list of plugins. If one chooses to go back, the plugin will only go idle. The plugin toolbar is implemented to do two main tasks:

1. Show where you are, and let you go back from a plugin (to the list of all plugins)
2. Show icons and buttons for each individual plugin.

This is what we implemented, and is nothing new with this implementation from Luma 2.4. After implementing this, we already decided to take a new approach.

### **A new and better implementation**

We decided to go for tabs. This means that each plugin will have its own tab in the main-window. One can simply close a tab, or switch to another, without using a toolbar. This is better than having only one of each plugin, where only one can be used at a time. This means that the `QStackedWidget` will be replaced with a `QTabWidget`. A second change, is that we decided to drop the toolbar completely. Each plugin that needs a toolbar, should create its own.

#### **7.5.4 Sprint 4**

We continued to work on the tabs from sprint 3 among others to integrate tabs with our main window and loading plugins into new tabs. When we finally did get up tabs for each plugin, we discovered some garbage collection problems. It was a circular reference problem. Whenever we closed a tab the plugin was still alive. No memory deallocation occurred. The solution, was to remove all references to the plugin we closed. We also had to call Python's garbage collector manually, after deleting all references.

We also decided to use tabs in the configuration of plugins, so when a plugin is clicked in the settings-dialog, a tab for "about" and "settings" shows.

## 7.6 Browser plugin

The browser plugin provides some of the core functionality of Luma: it allows and enables users to view the content of LDAP servers, view and edit the entries, add/delete entries, and export existing entries.

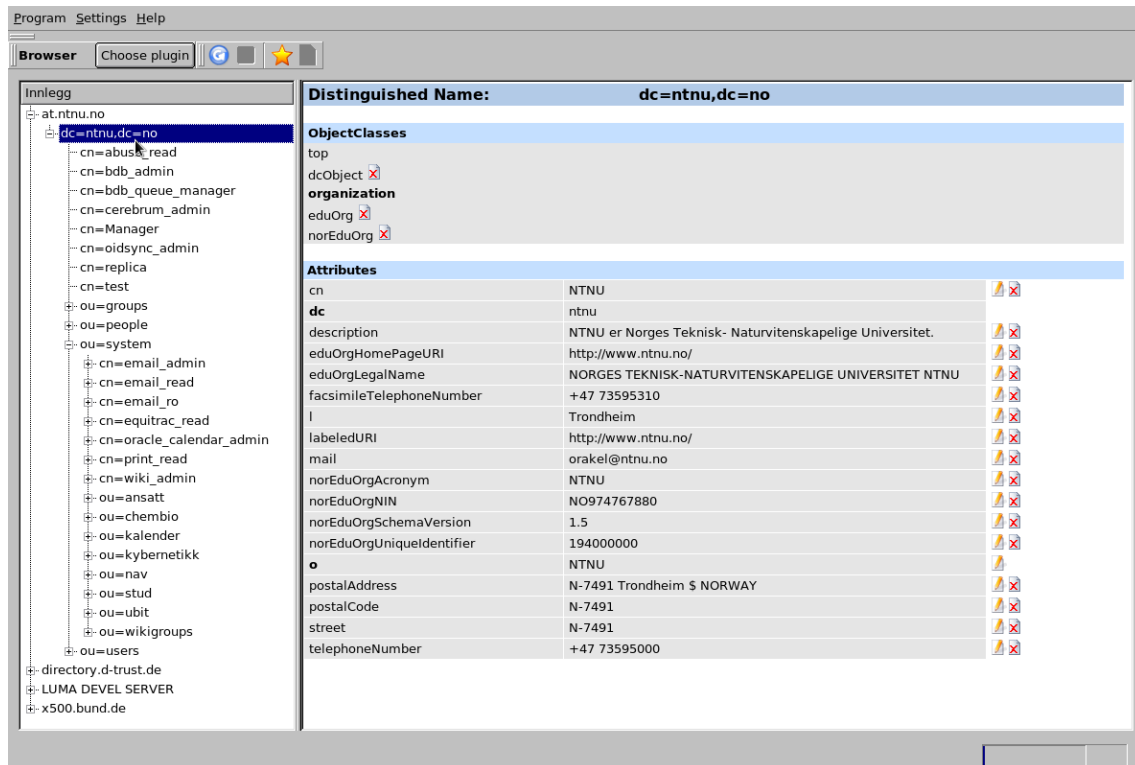


Figure 7.16: The browser plugin in Luma 2.4

The browser plugin existed in Luma 2.4, but some drastic changes had to be done in our implementation. Firstly, the classes used to construct the tree view in Luma 2.4 are now deprecated so little of the code could be used directly. Secondly, the old version did not comply to any form of the Model-View-Controller-pattern. This is something we would had to change anyway, even if the requirements did not specify it. This means that, in Luma 2.4, the code for displaying the data and user interaction is commingled with the code for gathering the data displayed. This is bad practice for a number of reasons, including a loss of separation of concerns (modular separation) which in turn makes the code harder to maintain.

The solution was pretty clear from the beginning. Qt 4 provides, and encourages, the use of their model/view-framework when designing GUI-applications. This framework, as the name implies, combines the *view* and *controller*-part of MVC to just *view*, as is common to do in GUI-applications anyway. In addition Qt 4 allows the use of custom delegates in order to customize the handling of user input and display of data in the view.

We will not go much further into the particulars of this system here.<sup>4</sup>

The use of this framework influenced the development: we would have to design and implement a model, again based on the model-classes provided by Qt, providing an interface to the LDAP servers for the views. Or, in other words, a model wrapping the content of the LDAP servers in a way usable to the view-classes; and a view, based on the view-classes, which displayed this content appropriately and provided the relevant actions, the controller part of MVC. Note again how the VC-part of MVC is combined in the Qt-framework.

As it turned out, none of this is as trivial as it might sound to developers unfamiliar with the Qt-framework. While the framework is powerful for complex cases, it can be unnecessarily complex when used to develop simpler functionality like a simple list of items. Some of the reasons for this is the framework's insistence of using a table based model for all data: the model-view-interface is always defined in terms of row- and column-numbers, and in the case of trees also parent-rows. This pretty unique approach to MVC meant it took us multiple weeks before we were completely familiar with the framework itself and could leverage most of its potential.

The following sections will describe the construction of the browser plugin through the project sprints.

### 7.6.1 Sprint 2

The first sprint with work on the browser plugin was sprint 2. At this point we had already had some experience working with the model-view-framework through the server dialog.

We were looking at how to be able to view entries from a LDAP server in a hierarchical tree list, as well as providing a simple display of the entry data. This meant constructing a model based on (subclassing) Qt's model-class `QAbstractItemModel` and using a standard view provided by Qt (`QTreeView` to display this and provide interaction with its model). Interaction at this stage meant expanding and collapsing items in the tree to show child items, and this was provided automatically by the standard view. Figure 7.17 displays the plugin as of the end of sprint 2.

Note that the plugin at this stage is shown and developed completely stand alone from the Luma 3.0 main window. This was because the plugin support, and main window itself, was being worked on simultaneously. This was made possible by the approach used for plugins where they would provide a widget the main window could display where it wanted.

In addition to creating the browser plugin widget itself, we also had to port the back-end classes which provide access to LDAP servers from Luma 2.4. This was largely unproblematic as the only major changes were in relation to the logging described in section 7.9.

The last functionality we added in this sprint was the possibility of specifying a *search-limit* and *filter* for items in the LDAP-entry-tree. This was implemented by the items having variables describing the filter, an entry criteria, and limit, on the number of items,

---

<sup>4</sup>Qt's documentation on MV-programming: <http://doc.qt.nokia.com/latest/model-view-programming.html>

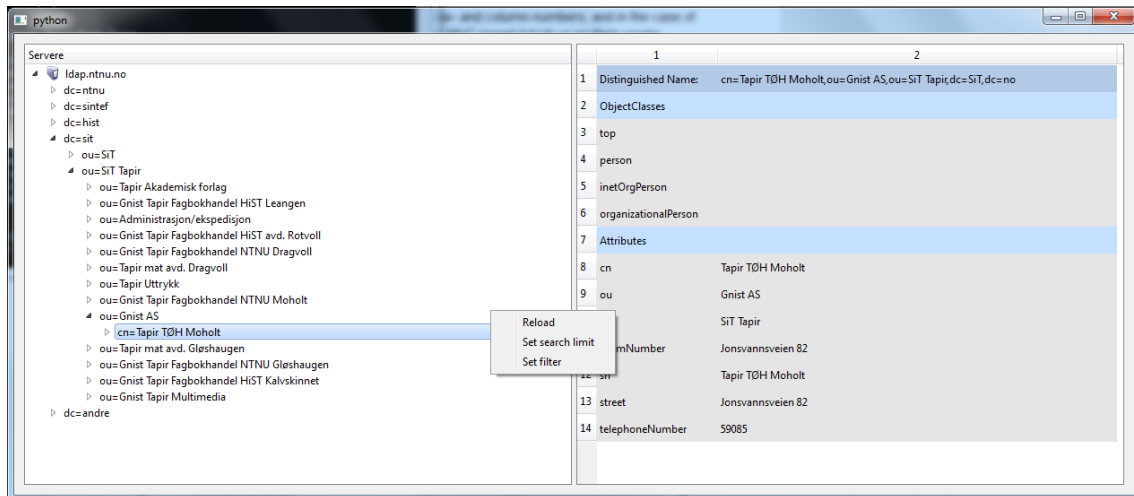


Figure 7.17: The browser plugin as of sprint 2

and using them when fetching the children of this item. Since this was already the responsibility of the item itself, as part of the model, this was easily added. The design for displaying, and what to display in, the context-menu (right-click menu) was at this point somewhat sub-optimal, as the menu-object was passed to the items themselves which then added the supported menu-options. If you could choose *Filter* it meant the item supported this since it added it to the menu itself. This system was improved upon in later sprints by giving the items an interface through which they could indicate which items they support. In later sprints this interaction was further complicated by the introduction of multiple-selection, further necessitating a *this-item-support-x-interface*.

### 7.6.2 Sprint 3

By this sprint the ability to browse server-content was complete. Next was porting the existing solution for displaying LDAP-entries. This was reasonably straightforward, even though some classes used were now deprecated, replacements were easy to find. The handling of loading icons was completely changed to use Qt's resource system.

In terms of new features, this sprint featured the introduction of tabs for displayed entries. This meant multiple entries could now be viewed and edited at the same time, something which was not possible in Luma 2.4. In Luma 2.4, when you opened an entry and started editing you were locked to it until you chose to save or discard changes.

Another notable addition was the ability to view entries without having received the LDAP-schema from the server. This can happen if the server is mis-configured, as is the case for e.g. NTNU's publicly available LDAP server. In these cases users of Luma 2.4 will only receive an error message, while with our version one has the option to view the content anyway.

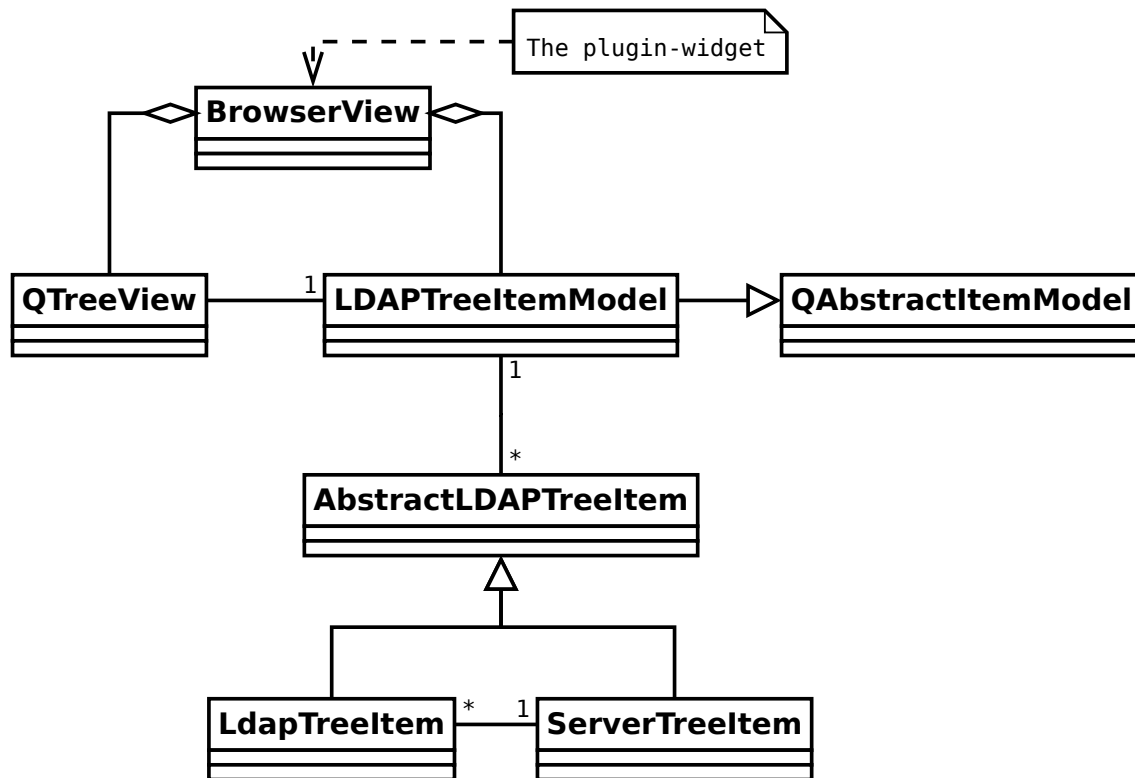


Figure 7.18: Classdiagram for the browser plugin as of sprint 2

### 7.6.3 Sprint 4

This sprint featured two new features, export and delete, as well as new error-handling, some internal changes to the interface for what items support as well as support for multiple-selection of items.

Previously, both in Luma 2.4 and our version in the previous sprints, errors encountered when browsing the LDAP-tree were displayed as error-dialogs as shown in figure 7.20 and 7.21. This we felt was both distracting, even though error-messages are supposed to be, and was harder to connect to where exactly the error occurred. In addition, it was not possible to later see where the error happened without checking the log, which is not shown in the figures.

With these issues in mind, we changed it so that errors were displayed as child-items in the tree as you can see in figure 7.22. This made it easy to spot where the error occurred, as well as remembering that there was an error as it is readily shown in the tree.

The changes to the interface concerning what the items support were to move away from using some sort of "isinstance"-check to see the type of the item (e.g. a server or ldap-item, or a call to a method which would return a non-None value if it was a ldap-item such as `item.getSmartObject()`). Instead, all items were given a `getSupportedOperations()`-method, through adding it to the `AbstractLDAPTreeItem`-superclass and requiring sub-

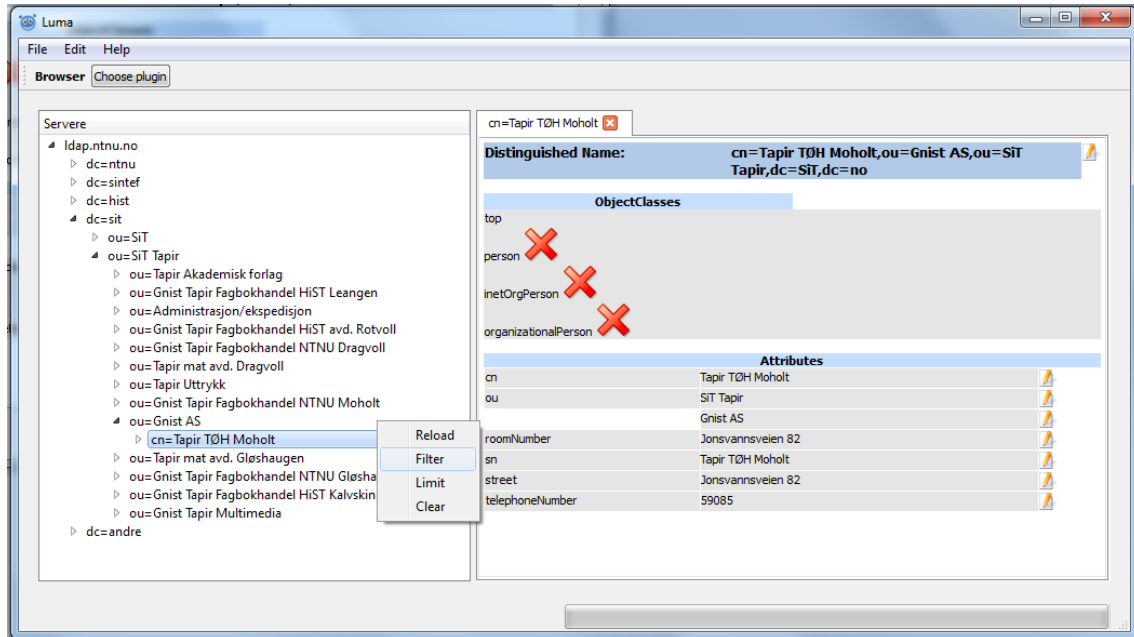


Figure 7.19: The browser plugin as of sprint 3

classes to override it if they wanted to indicate they support something. This method return a bitmask indicating what is supported. This bitmask is generated by binary OR-ing constants indicating the possible supported actions. The following code sample illustrates this for an item supporting the addition of a filter as well as exporting.

```
def getSupportedOperations(self):
    return AbstractLDAPTreeItem.SUPPORT_FILTER | \
        AbstractLDAPTreeItem.SUPPORT_EXPORT
```

The constants are defined as follows (taken from the `AbstractLDAPTreeItem` module):

```
# Used from getSupportedOperations()
# which returns the result of or-ing (|) the supported operations
# e.g. "return SUPPORT_FILTER | SUPPORT_LIMIT"
SUPPORT_NONE = 0 # Should only be used alone
SUPPORT_RELOAD = 1 # Probably works on all items
SUPPORT_FILTER = 2 # Indicates the item has implement setFilter
SUPPORT_LIMIT = 4 # Indicates the item has implement setLimit
SUPPORT_CLEAR = 8 # Probably works on all items
SUPPORT_ADD = 16 # Can add child-items
SUPPORT_DELETE = 32 # Can remove this item
SUPPORT_EXPORT = 64 # Can be exported
SUPPORT_OPEN = 128 # Can be opened (has smartdataobject)
```



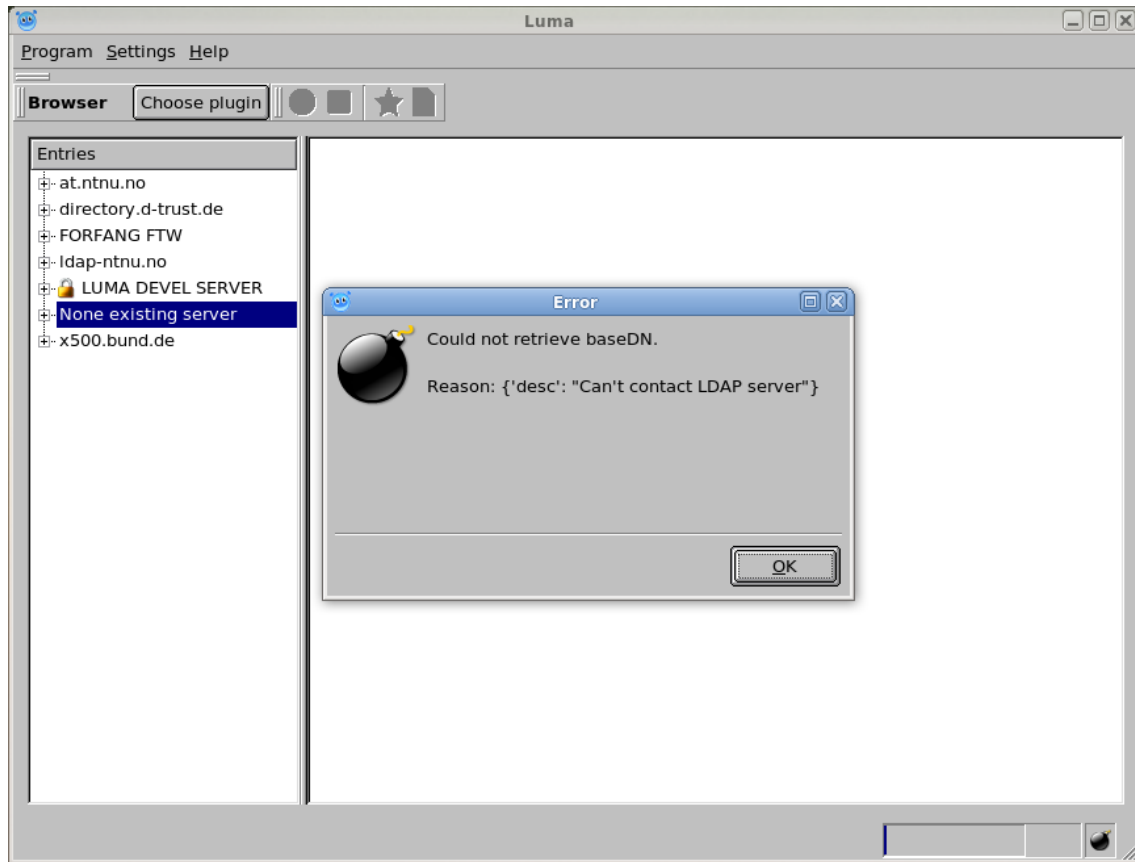


Figure 7.20: Failure to connect to a server in Luma 2.4

Note how the constants are defined as multiples of two in order to support this usage. In order to check for support, we use code similar to this:

```
supports = item.getSupportedOperations()
if support & AbstractLDAPTreeItem.SUPPORT_EXPORT:
    self.exportItem(item)
```

The `&`-operator ANDs the two values, resulting in 0 if the item does not support the operation checked for (meaning the if fails), or the operation-value if it does (meaning the if goes through).

When selecting multiple items, we iterate through all of them and check that they support a certain action before making this available through the context-menu. This ensures one can't end up attempting an unsupported action on an item.

Exporting items were reasonably straightforward using provided library methods, so the work amounted to providing a GUI for selecting where to export and in what format. In addition, the user can select to export the entire subtree of an item or just the items actually selected.

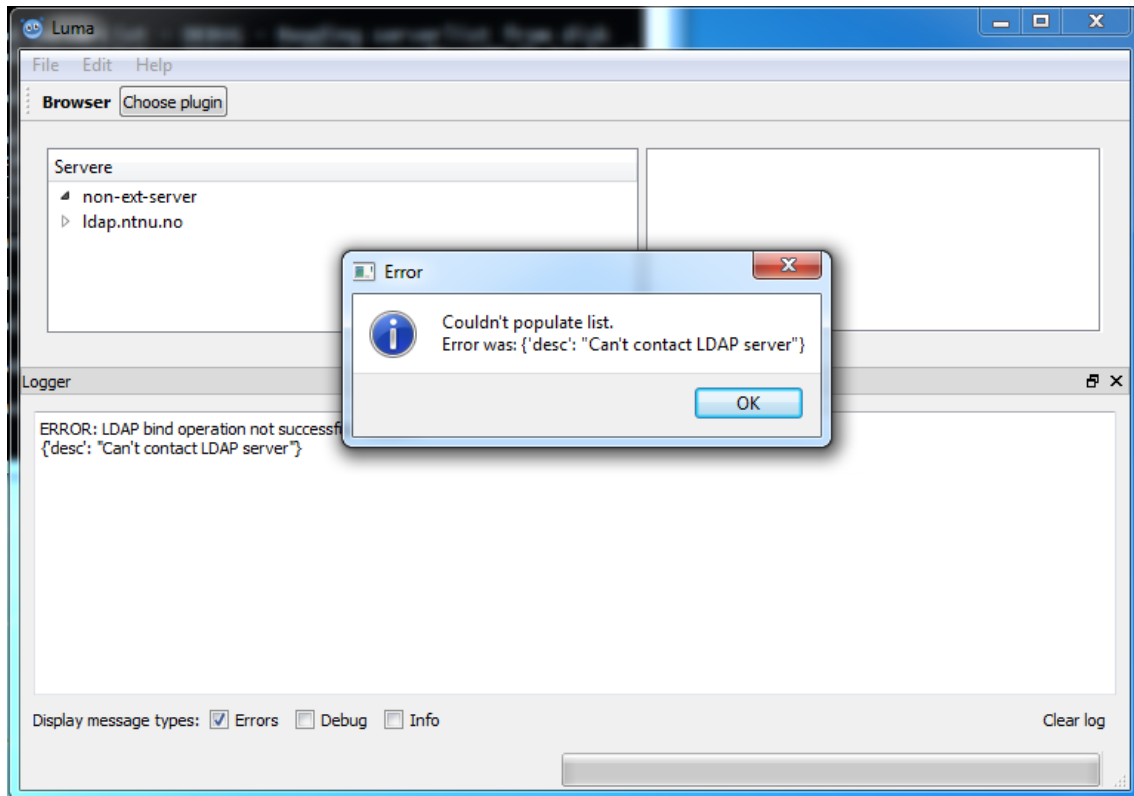


Figure 7.21: Error when trying to connect (open) the server named no-ext-server after sprint 3

The ability to delete was in two forms: single and multiple selection. With a single item selected, one can simply right-click it and choose *Delete*. The user is then asked to confirm the action. The model then tries to delete the item on the server and, if successful, it is also removed from the model and, by extension, the view.

When multiple objects are selected for deletion, a dialog pops up showing all the selected items. The user can then de-select items if he has changed his mind. After confirming the selection the items are attempted deleted and the success or failure is displayed by the individual items.

### Entry view

In Luma 2.4 there were only one default view in the browser plugin. It was a display of generated HTML code, with the values of the ldap-object added while generating. The customer wanted a feature, similar to what *JXplorer* has as described in section 2.4.1, with the possibility to create custom html views that depends on which object-classes the current ldap-object has.

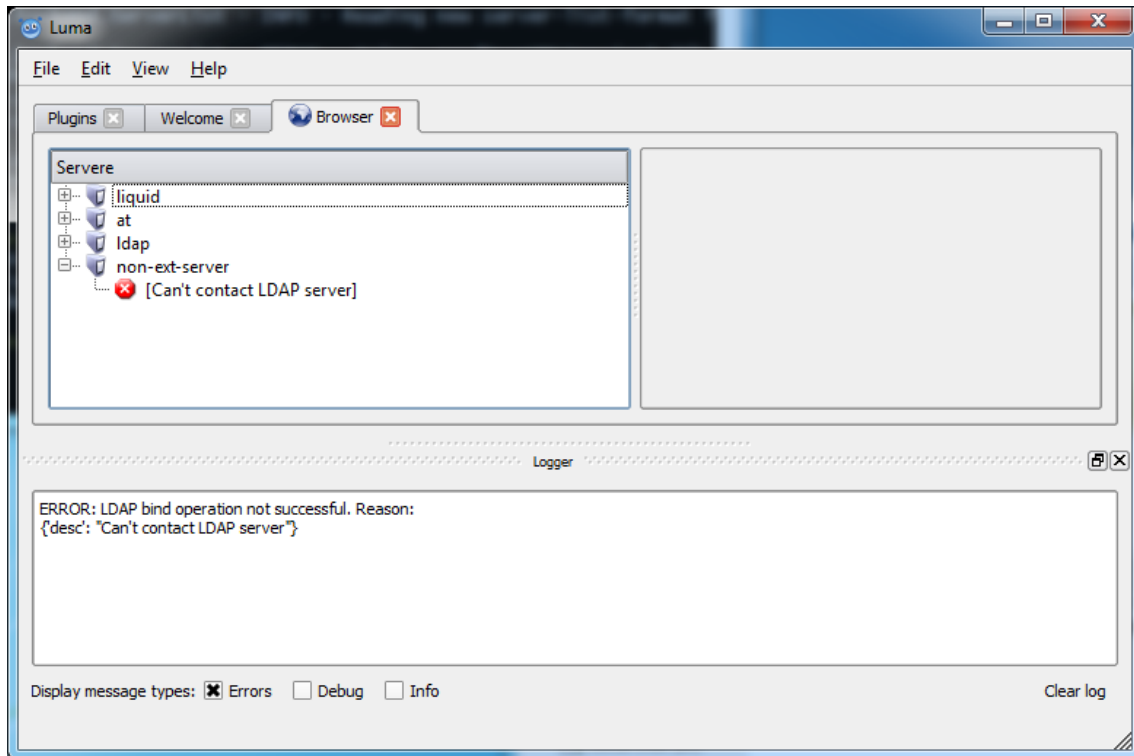


Figure 7.22: The same error as in figure 7.21 with sprint 4

### HTML templates

To implement the views we made a small extension to the html-language, that allowed the user to mark in a html document where, and what should be displayed.

```
<htmltag> ::= ldapobject function = <functioncall>[ id = <attributeid>]
           [ style = <stringstyle>]
<functioncall> ::= getPrettyDN | createClassString | createAttributeString
                 | attribute
<attributeid> must be one of the objectclasses attributes
<stringstyle> ::= rows
```

getPrettyDN is the full DN string of the ldap object

createClassString is a list of all of the ldap objects classes

createAttributeString is a list of all of the ldap object attributes,  
with values

attribute is a list of all of the attribute name and the values

stringstyle is the style of the string, rows gives a list of html table  
rows

We chose to make a folder where the user could put his custom views. The files at the first level will be displayed for all ldap-objects, and the files at the second level will be displayed if the name of the folder matches one of the ldap-object's classes.

```
templates/
templates/inetOrgPerson/main.html
templates/organization/main.html
templates/classic.html
templates/error.html
```

Classic is the default view and error displays structural errors with the ldap-object.

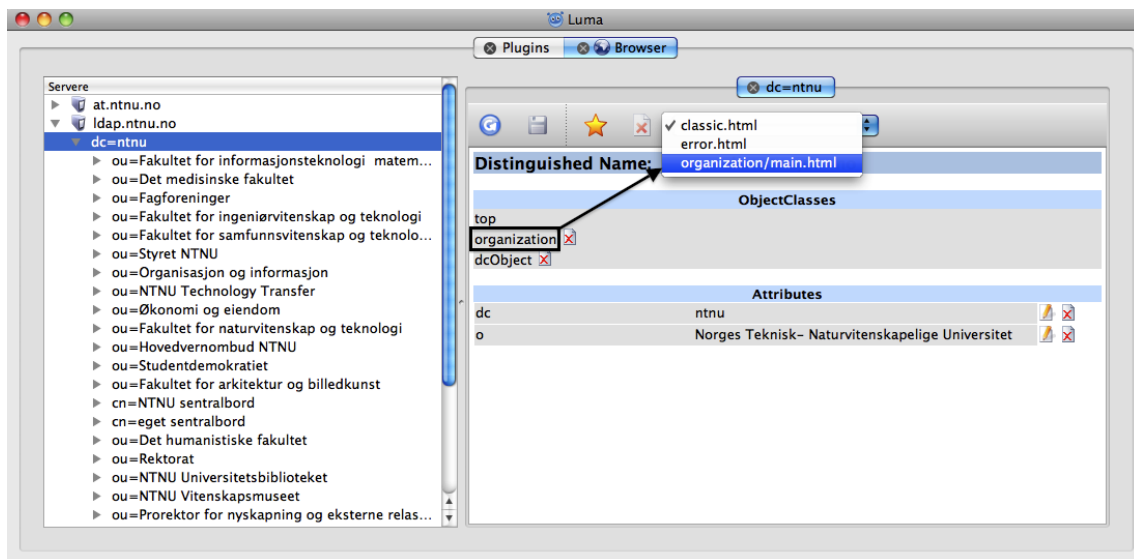


Figure 7.23: Entry view

#### 7.6.4 Sprint 5

At this point the functionality of the browser plugin was mostly complete except for the ability to add items. The work here amounted to converting old dialogs, connecting them to our code for displaying context-menus and then providing functionality reload so that the addition is shown. This was completed without running into any major issues.

This sprint also featured a major change to how the model loaded items. The back-end code for retrieving data from LDAP servers was at this time working by starting a thread which did the retrieval and doing calls to `QApplication.processEvents()` until the retrieval was complete. This call tells Qt to process all events currently queued in its event-queue, for example requests to resize windows, push buttons or expand items. Using this solution, the application was responsive to user input while the retrieval was in progress.

Unfortunately this had the side effect that requests for expanding items or opening windows while a request was in progress would be processed before the first request was finished. Control would not be returned to the waiting-for-retrieval-finished-loop before the event had been processed, and since operations on items in the ldap-entry-tree also are processed as events, one could perform operations on items already being worked on. Opened dialogs would also block the retrieval of items until the dialog was closed.

We solved this problem by running the backend code in its own thread (a `QThread` created by the model. The call to `processEvents()` by design only ends up doing something if it is called from the designated GUI-thread, so it would in this case end up doing nothing. This meant the solution could be implemented without changes to the backend code (which could negatively effect other parts of the program). When the data was retrieved if was signaled using Qt's slots/signals-system to the model (signals works across threads) where it would be immediately added regardless what state the rest of the program was in.

Some benefits of this change are that multiple items can be loaded simultaneously instead of in a last-operation-first-fashion: if you expand an item with a long list of children after an item with a shorter list of items, the shorter list will be retrieved and shown first. Previously the first operation performed would not return before the latest operation was complete. The change to this signal/slot-system also mitigated any possible problems of allowing interaction with the model while in a "fetching-data"-state (we never encountered any but it seems like it conceptually could). To be sure to avoid this possibility we chose to display a message-dialog informing the user that retrieval was in progress, indicating that the user should not cancel the message and continue working (though he could). With the new solution this was not an issue, and we chose to take advantage of this by allowing the users to interact with other items while other were loading and instead show which items were currently with a right-aligned "Loading..." text on the item's row.

### 7.6.5 Sprint 6

At this point the template-plugin was complete, so the ability to use the created templates in the browser was added. This was reasonably straight-forward as it had much in common with the already implemented add-entry-method. The only real difference is the preloaded smart-data-object used.

### 7.6.6 Performance comparison with Luma 2.4

The redesign of the browser plugin internals together with some optimization of the backend, resulted in our version of Luma being significantly faster than Luma 2.4. Table 7.1 shows the results for comparing loading-times for extended LDAP server entries in Luma 3.0 and Luma 2.4. We used NTNU's LDAP server, located at `ldap://at.ntnu.no`, and the times identifies how long it takes from the subtree of `ou=users` is expanded until the items are shown on screen. The `ou=users` entry on NTNU's LDAP server, contains 50434 child entries, making the expansion of the entry a demanding operation. The test was performed 3 times for each version and on identical hardware for both versions of the

## 7.6. BROWSER PLUGIN

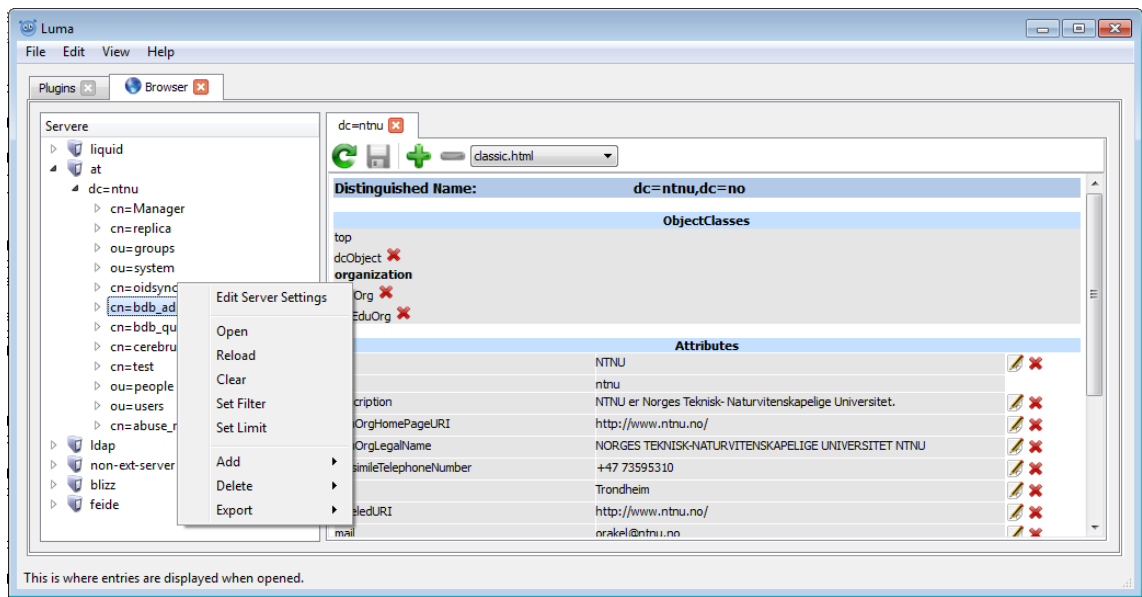


Figure 7.24: The browser plugin after sprint 6

application:

**CPU:** Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz  
**Cache:0** L1 cache 64KiB  
**Cache:1** L2 cache 4MiB  
**Memory:** 2x2GiB à 667 MHz

Table 7.1: Browser performance-comparison results.

Luma version	Time in seconds
2.4	14.4
3	11.3

### 7.7 Template plugin

---

The template plugin adds the functionality of standardized entries. Users can for example make a template that declares the information an entry for a student at NTNU should have, shown in figure 7.25. This template can be used later in the browser plugin to make new student entries.

Like most of the plugins we have made, this also existed in Luma 2.4. There is only a few changes in the use and design of the plugin. In our version the name and description of the templates can be changed. Something else that is changed is that the attribute and objectclass views have extended selection and therefor support deletion of multiple attributes and objectclasses. The dialog for the adding of attributes is also changed from having check boxes for those you want to add, to selections of the attributes to be added. There are also some minor GUI improvements like adding splitters so the user can customize the GUI more. The implementation also uses the MVC pattern mentioned in section 7.6.

#### 7.7.1 Sprint4

In sprint 4 we started working on the template plugin. At the end of the sprint we had added most of the basic functionality and had all of the GUI done. We could add and remove templates and change their name and description. The views and models for attributes and objectclasses worked for showing objectclasses and attributes. We chose to keep the way the objectclasses and attributes were presented in their views. Objectclasses are still represented only with a list of strings and attributes a table with icons on the *must*, single and binary columns and the rest strings. The models used are very similar to the models used earlier in the project and therefor were easier and took shorter time to make. The possibility to read and save the list over templates to file was also added at this stage in the project.

#### 7.7.2 Sprint5

The last functionalities were added in sprint 5. Among these are adding objectclasses and attributes retrieved from the server the template is going to be used to make entries on. In addition when adding objectclasses, the program checks the objectclass attributes for *must* (i.e. mandatory attribute) and adds the attributes if they are not already in the template's list of attributes. The same attributes are removed when objectclasses are deleted, but not if the objectclasses that are left still support the attribute. We also ensured that attributes which are *must* could not be delete.

#### 7.7.3 Sprint6

In the last sprint there were only some adjustments and the adding of the low priority functionality *custom-must*. In a meeting in sprint 6 the customer suggested that *custom-must* was added so users could force attributes to have value when added in the browser

## 7.7. TEMPLATE PLUGIN

plugin. We added this possibility by changing the attributes *must*-value to a blue check mark in the template plugin.

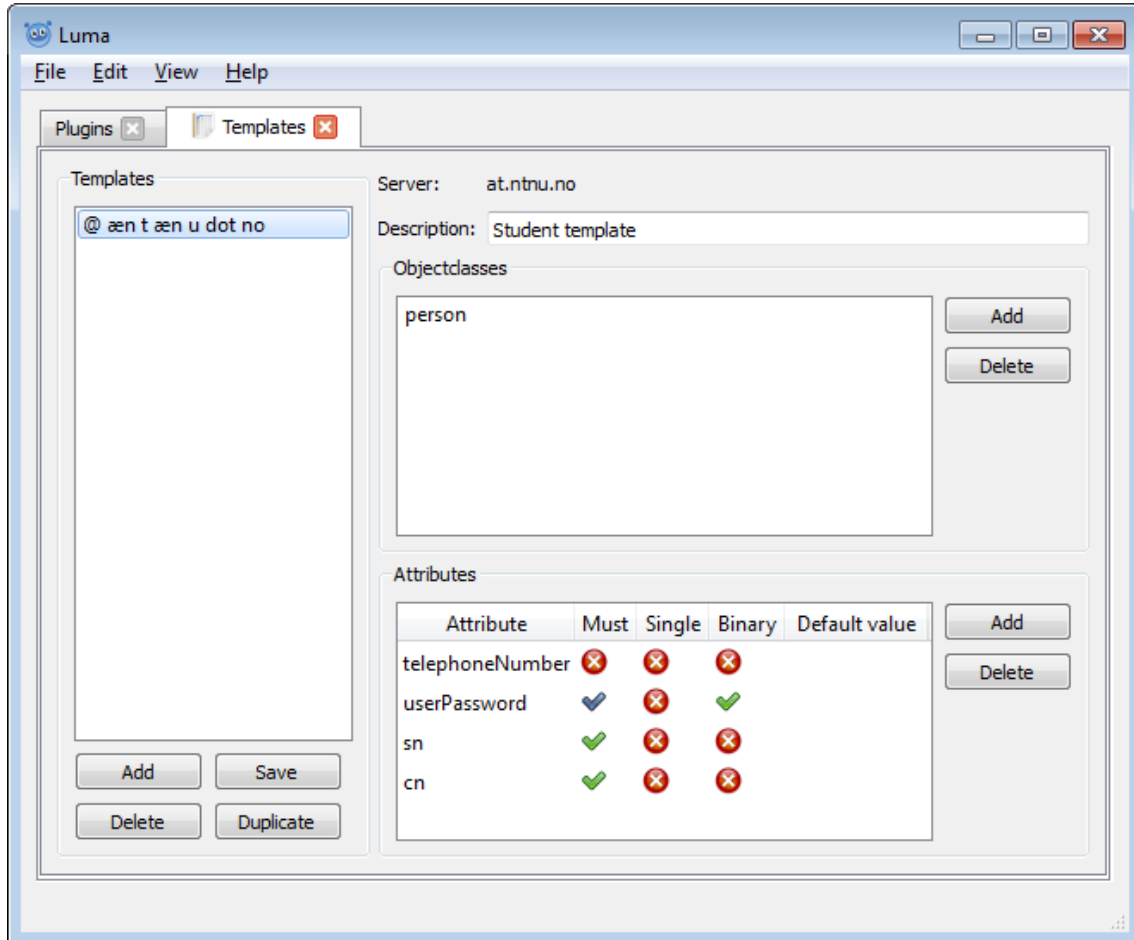


Figure 7.25: The new template plugin



### 7.8 Search Plugin

---

Being able to search for arbitrary entries on an LDAP server is an expected functionality in any LDAP administration tool. Luma 2.4 made this functionality available through the *Search plugin*. As described in chapter 4 (functional requirement number 6), we also needed to implement a search functionality in the new Luma implementation. During sprint 4 and 5 we redesigned and implemented the search plugin.

#### 7.8.1 Sprint 4

##### Design

We started to look at the search plugin in the end of sprint 4. The search plugin consists, mainly, of two parts: The *search form*, where the user chooses server, baseDN, and provide the search filter to apply on the LDAP search operation; and the *result view* where the actual search results are displayed. The old design of the search form was spanning horizontally across the screen, using all available screen space. This means that less vertical space was left for the result view.

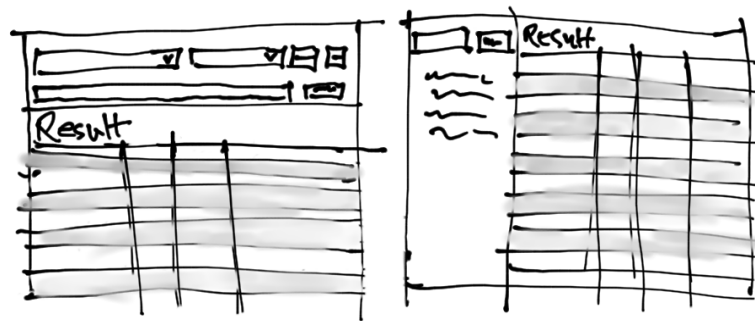


Figure 7.26: Search plugin redesign mockup (old design to the left)

With the assumption that a regular LDAP search filter, rarely includes too many object attributes, we figured that the screen space required to display the search results was far more limited by the available vertical size of the screen, than the available horizontal screen size. We therefore redesigned the search form, and made it more like a sidebar located to the left side of the screen. This way we could display more results per scroll than the case was with the old design. A mockup sketch, illustrating the difference between the two designs, is shown in figure 7.26.

##### Implementation

**Search form** The actual implementation of the search form was finished just before sprint 4 ended. We added some additional search options to the new search form. This

included the option to define the search level<sup>4</sup> and the option to set a size limit<sup>5</sup> on the search operation. This was an improvement compared to the old implementation. But even with the new improvements, the overall functionality of the search plugin, remained very limited at this moment. We were able to do some basic search operations on selected servers, but we had no implementation of the result view, to actually display the search results. We therefore needed to include the search plugin feature in sprint 5 as well. The search form after sprint 4 is illustrated in figure 7.27, and the search form as seen in Luma 2.4 is illustrated in figure 7.28.

Figure 7.27: The search form after sprint 4.

Figure 7.28: The search form as seen in Luma 2.4.

## Testing

Because we did not have time to implement the actual result view during sprint 4, not much testing was done on this feature at this point. We did however register that the search operation in the new implementation received the same amount of results as the old implementation, which the same search criteria.

<sup>4</sup>The search level of an LDAP search can either be `SCOPE_BASE`, `SCOPE_ONELEVEL` or `SCOPE_SUBTREE`.

<sup>5</sup>The size limit defines the amount of entries to retrieve from the search. This is given as an integer (where, for example, 10 means to retrieve only 10 entries. Note that 0 means no limit).

### 7.8.2 Sprint 5

#### Design

In sprint 5 we continued to work on, and improve, the search plugin. In addition to further improvements to the search form, we finished the design of the result view. The result view did, however not need to be redesigned, and we ended up using the same design concept as in Luma 2.4: a table view where each column represents an attribute value from the search result. The result view is illustrated together with the search form in figure 7.35.

In sprint 5 we also created a completely new *filter builder*, inspired by the somewhat badly designed *filter wizard* in Luma 2.4. With the filter builder we wanted to create a simple but efficient tool for building complex LDAP search filters. We ended up with designing a simple LDAP search filter editor, as illustrated in figure 7.31, and described more in detail in the implementation section.

#### Implementation

**Search form** Among the improvements to the search form in sprint 5, was the attribute auto complete feature. This feature gives the user a list of (possible) matches from all available attributes, specific to the selected server, while he types in the search filter criteria. The attribute auto complete feature is illustrated in figure 7.30. This feature can be turned off or on through the settings widget for the search plugin, giving the user the freedom to choose.

We also added a tool button (located to the right of the search button), which enables the user to continue editing the current filter in the filter builder. This can be very effective if the current filter fails to pass the LDAP search filter validation.

Figure 7.29: The Luma 3.0 5 Search plugin, search-form design.

Figure 7.30: The Luma 3.0 5 Search plugin, search-form with autocomplete enabled.

**Filter builder** LDAP search filters follow a strict polish notation syntax, where each filter component must be inside closing braces. Examples of LDAP search filters can be:

```
(&(objectClass=*)(cn=*Uvsløkk))
(&(objectClass=person)(sn=*sen)(!(cn=Per*)))
```

When we started to implement the filter builder, we wanted to create a simple and efficient tool for building complex LDAP search filter. In the filter builder the user can insert filter components composed by objectClasses or attributes with corresponding values. The combobox is filled with options based on the selected type, and is always filled with options specific to the current selected server in the search form. We also implemented syntax highlighting in the filter builder, which can be useful when creating complex search filters. The filter builder with and without syntax highlighting is illustrated in figure 7.32 and 7.31.

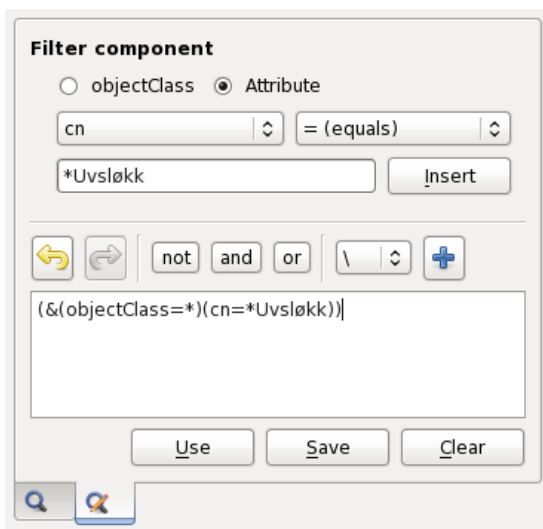


Figure 7.31: The Luma 3.0 5 filterbuilder design.

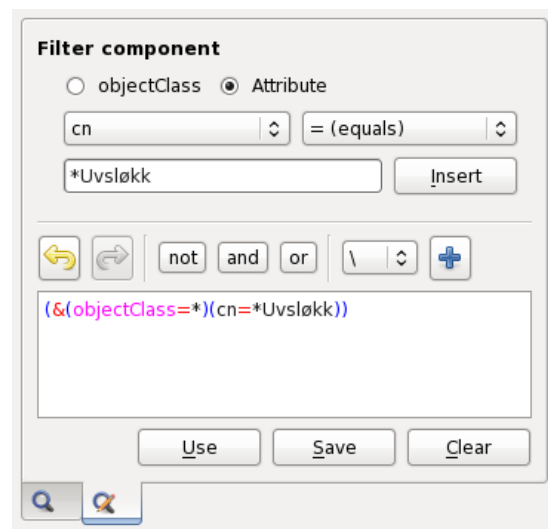


Figure 7.32: The filterbuilder with syntax highlighting enabled

**Result view** As mentioned in the design section, there were not so much we needed to improve from the old result view. We did however need to do some refactoring, as part of following the MVC design pattern. But even though the main design is somewhat unchanged, we did add some additional features and functionality to the result view. Among these is the filter box, which we have made accessible though the common key sequence CTRL-F. This feature enables the user to do some post filtering on the returned search results. The filter box supports three types of filter syntaxes . The user must

<sup>4</sup>The user can choose to filter the result view with *Fixed String*, *Regular Expression* or *Wildcard*.

select one of these syntaxes and the column he wishes to apply the filter on. We also implemented support for having multiple result views open at once. This was achieved by using tabs to display each search result. Both the filter box feature and the support for multiple result views is illustrated in figure 7.33.

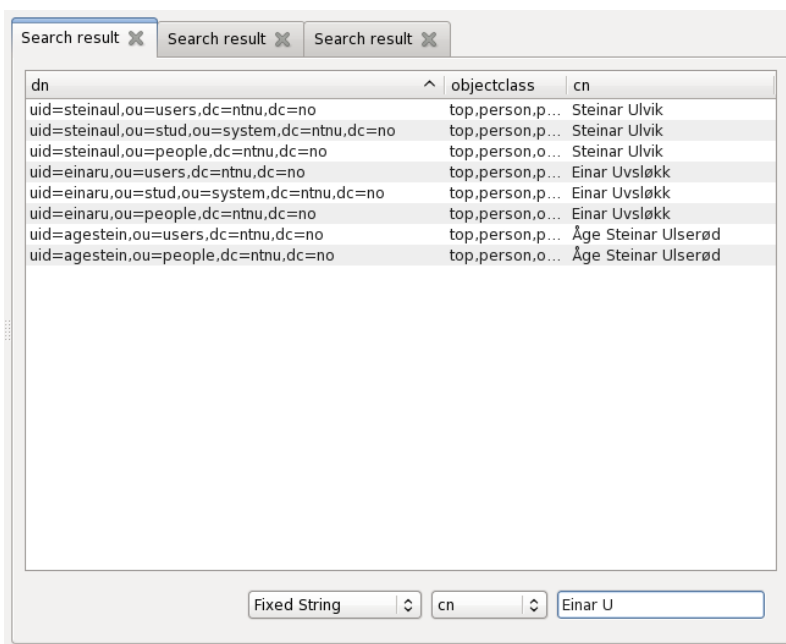


Figure 7.33: The new Luma result view with filter support.

### 7.8.3 Sprint 6

As a consequence of the improvements made to the back-end, regarding binding to and performing various operations on LDAP-servers as described in section 7.10, some changes had to be made to the implementation and design of the search plugin internals. The user interface, however, remained untouched.

#### Implementation

The introduction of the `LumaConnectionWrapper` in the back-end, enabled us to perform asynchronous search operations. What this meant was that we got a much cleaner code, as the need for tweaking GUI responsiveness was eliminated at the plugin level. However, the implementation with the use of the new `Connection` interface proved to be a little more tricky than first assumed. The search plugin was the last of the plugins to facilitate the new `Connection` API. This resulted in previously unseen bugs and errors being discovered, primarily regarding threadpool cleanup, which in turn proved to further improve the new back-end code.

7.8.4 Summary

The old and new version of the search plugin is illustrated in figure 7.34 and 7.35 respectively.

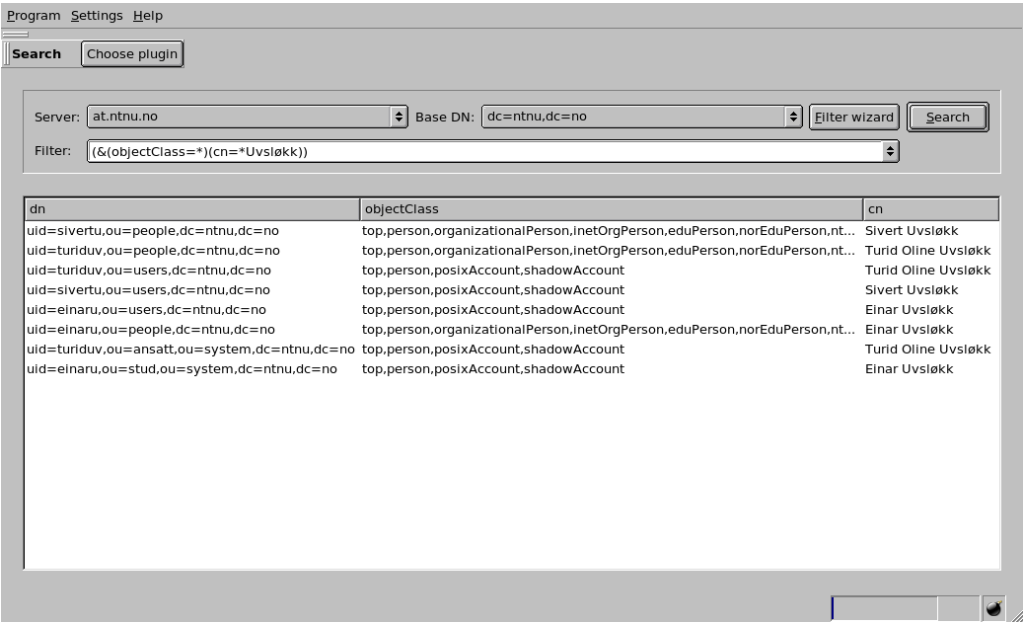


Figure 7.34: The Luma 2.4 search plugin.

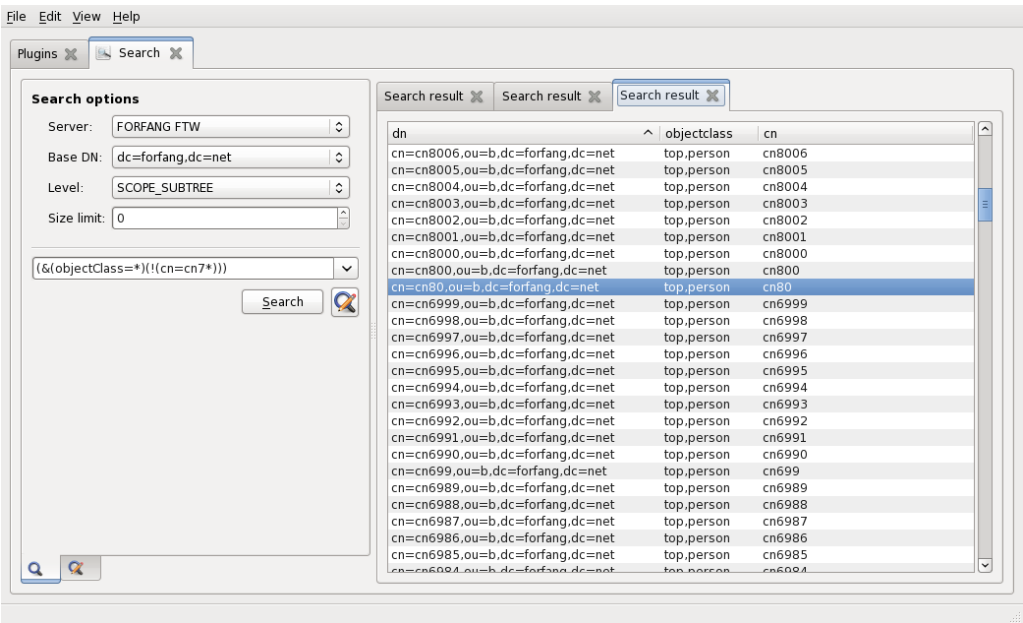


Figure 7.35: The new Luma search plugin with support for multiple result views.

The main improvements in the new version of the search plugin is:

- the new implementation of the User interface, resulting in a more intuitive and screen-size efficient user interface.
- the implementation of the filter builder. A greatly improvement compared to the filter wizard found in Luma 2.4, with its many more features, utilizing the composing of complex LDAP filters.
- the support for having multiple result views opened at once, as a direct benefit from using a tabbed layout.
- the ability to do additional filtering on a returned result set, introduced with the result view filter box.

## 7.9 Logging

The internal logging in Luma 2.4 uses its own implementation of logging through a global `environment`-module, which all classes would then be depended on. Through discussion with the product owner it became apparent this system is a remnant from when Luma 2.4 was first developed. Python has since version 2.3 (released in 2003) provided its own logging-facility, which compared to the logging in Luma 2.4, is vastly superior in every way, so we saw no reason not to replace the old logging system. In terms of work this would mean changing existing code similar to:

```
import environment
environment.logMessage(LogObject("Info", message))
environment.logMessage(LogObject("Debug", message))
```

..in to:

```
import logging
_logger = logging.getLogger(__name__)
_logger.info(message)
_logger.debug(message)
```

Features gained by using Python's logging include: guaranteed thread-safety, multiple possible (and preexisting) handlers (log to console, a widget, file), filtering based on log-level, info about calling class and thread.

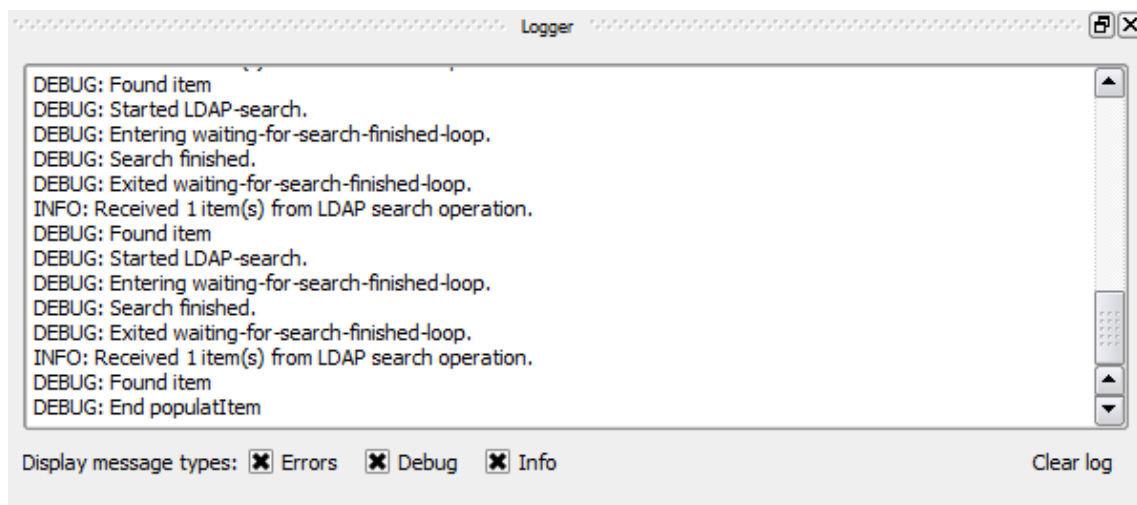
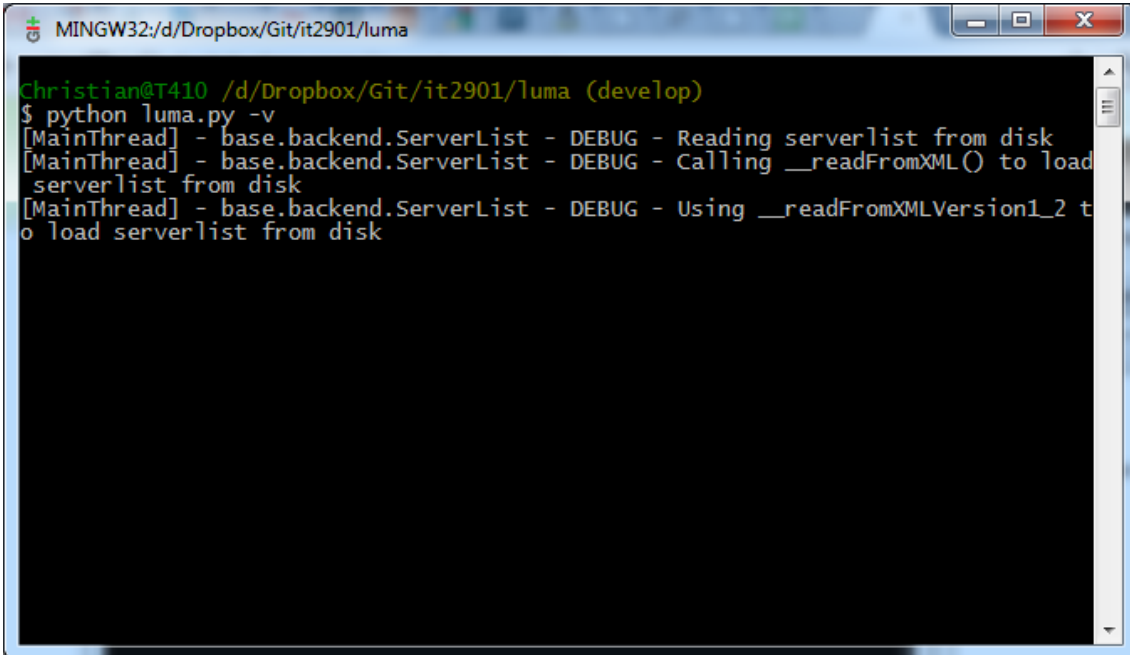


Figure 7.36: The loggerwidget

In addition to changing the logging itself, we also had to rewrite the logger-widget used to display the log in the GUI. This was no problem as it consisted of implementing our own handler which would then process the log according to what log-levels the user have chosen to display and then display the log in the widget-textfield, shown in figure 7.36.

The conversion to the new logger was done continuously as old code was ported while the loggerwidget was added in sprint 2.





```
MINGW32:/d/Dropbox/Git/it2901/luma

Christian@T410 /d/Dropbox/Git/it2901/luma (develop)
$ python luma.py -v
[MainThread] - base.backend.ServerList - DEBUG - Reading serverlist from disk
[MainThread] - base.backend.ServerList - DEBUG - Calling __readFromXML() to load
serverlist from disk
[MainThread] - base.backend.ServerList - DEBUG - Using __readFromXMLVersion1_2 t
o load serverlist from disk
```

Figure 7.37: Logging to the console when the `-verbose` flag is used

## 7.10 Back-end

The LDAP-code in the back-end of the program was, until sprint 6, left almost completely unchanged from what was used in Luma 2.4. The code worked by starting a Python-thread doing the client-server-interaction while calling `QApplication.processEvents()` until the thread was finished, as described in section 7.6.4. This was largely unproblematic, but it had a few weaknesses we would have liked to improve on, but this was not a priority. In Sprint 6 we discovered a major problem with this solution which prompted us to do a redesign.

### 7.10.1 Sprint 6

With the change to using tabs for the plugins in the main-window, we discovered a bug/design-fault which was also present in Luma 2.4. Because of the backend code running `QApplication.processEvents()` while waiting for the data to be retrieved, it is possible to close the plugin while it is working. This would prompt Python's garbage-collector to delete the plugin and by extension also the back-end object and its started and currently running thread. The result was a segmentation-fault / crash (threads seemingly does not like being destroyed while running).

As mentioned this weakness was also present in Luma 2.4, but without the ability to run and easily close plugins this was not nearly as much of an issue during normal use. However, one could easily reproduce this by selecting to "reload plugins" while one or more of the plugins are performing an operation.

To resolve this issue, we choose to redesign the back-end LDAP-code to both resolve the problem and also implement some of our envisioned improvements. We first completely separated out the LDAP-code to its own class, and then went ahead with making our own wrapper-class around it. This wrapper would then provide the opportunity to run the LDAP-code in its own thread (similar to how the old code worked), and also provide a completely new API making it possible to receive the data through Qt-signals. The former works similar to the old solution with the exception of the threads being implemented as `QThreads` (not Python-threads) and with the change that a reference to the thread is retained as long as the thread is running. This stops the thread from being destroyed even if the originating plugin is destroyed. The latter mentioned API means plugins can now choose to be notified through signals when the data is retrieved instead of having to be waiting for it to be ready. Using this solution also completely bypasses the calls to `processEvents()`, eliminating its associated issues.

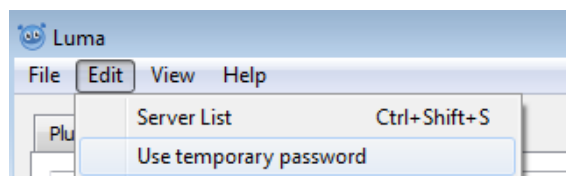


Figure 7.38: Users has the ability to use a temporary password which is not saved to disk

This redesign was not implemented without issues though. The old code took advantage of that it was usually (though it was not guaranteed to) run in the main GUI-thread by asking the user for another password should his entered password be wrong or missing. This was usefully for a few reasons, including mistyping and the ability to not having the password be saved to disk as part of the server-configuration. However, commingling back-end and GUI-code like this is obviously not particularly good design and that the code assumed it was running in the GUI-thread without checking was not good either. Our solution remedied these problems, though the temp-password-functionality had to be changed since it did not belong in the back-end like it did previously. Discussion with the customer revealed that the main purpose was not to help mistypes of password (since in this case the user should instead fix it in the server dialog – not retype in on each program launch), but rather to allow for a using servers with passwords without requiring it to be saved to disk (unencrypted). With this in mind we implemented the possibility for the user to select this temporary password using the *Edit*-menu, see figure 7.38. One could argue this is a "downgrade" in functionality in regards Luma 2.4 (where the dialog-box was displayed automatically on any use of the back-end code), but the overall improvements to the design and implementation of the back end is in our opinion worth it.

## 7.11 Installation

---

At the end of sprint 2, the customer articulated the desire for having some sort of installation procedure for the application. The installation procedure should be usable on every platform we was targeting. We took this new wish into consideration and scheduled it for inclusion during the planning of sprint 3.

### 7.11.1 Sprint 3

During sprint 3 we started to look at ways to deploy the application on different platforms. We spent a good amount of time studying and comparing various solutions for this problem. There exists many solutions for deploying Python applications, both cross-platform and platform specific solutions. We have gathered the essence of our study on this issue in appendix D.

We ended up writing a setup script using the *distutils* module found in the Python standard library. This module enabled us to create both built and source distributions of the application. While this is sufficient for some users, we also looked into the ability to create so called frozen distributions. This is a way of deploying software that involves bundling all required runtime dependencies into one install-able package. On most UNIX and UNIX-like platforms, this is not necessary as the Python interpreter most likely is already installed (or easily install-able through a package manager). However, on platforms like Windows, no Python interpreter is installed by default, and the user is therefore in the need of installing a bunch of software before being able to install our application.

On the Linux platform there is a lot of different flavors (distributions), which all follow some sort of guidelines for installing software. Most of the big distribution provide so called package repositories with installation and software management tools to easily and convenient retrieve different software. By providing good documentation of the code, the licenses used, and installation instructions with required dependencies, we will do our best to accommodate the various Linux distribution packagers.

### Settings, storage and file-paths

One issue we ran into during the deployment and installation feature, was the way Python imports other modules, as well as how to ship resources, like icons and translation files. First of we fixed the path issues in the source code with a manipulation of the `__file__` attribute, to located the relative execution root of the application. We also re-factored the code base, following the Python Enhancement Proposal 0328<sup>5</sup> regarding the use of relative `import` statements to access other Python modules. We also took advantage of the Qt framework resource system, resulting in all application resources (icons and translation files) being compiled into Python code for access at runtime. In order to automate the process of this resource compiling we also created the *Luma Tool Chain* concept, discussed in section 7.12, which consist of custom Python scripts intended to aid developers in the development and deployment work.

---

<sup>5</sup>PEP0328: <http://www.python.org/dev/peps/pep-0328/>

### Implementation

At the end of sprint 3 we had a functional, but minimal, setup script that was able to install the application on Linux systems. The deployment process was done as illustrated in the following example, with commands issued from the command line:

```
python setup.py sdist                # Create a source distribution
```

By which a `tar.gz` compressed archive was created. By then downloading and extracting this archive the installation procedure was as simple as running the following command from the command line:

```
tar xvzf luma-3.0.3-sprint3.tar.gz # Extract the source distribution
cd luma-3.0.3-sprint3
sudo python setup.py install        # Install the application system wide
```

### 7.11.2 Sprint 6

In sprint 6 we took up the work from sprint 3, on the installation feature in order to make it truly cross platform. The main issue with the state of our setup script, was that it was only targeting the Linux platform.

We made the script runnable on the rest of the platforms, by adding various platform checks before the actual setup in the script. This way we was able to not include resource files intended for the Linux platform, with the other platforms, and vice versa. The following mockup Python code illustrates the essence of the solution:

```
...
if sys.platform.lower().startswith('win'):
    _include_platform_spesifics = dict(windows_spesific_files)
elif sys.platform.lower().startswith('darwin'):
    _include_platform_spesifics = dict(macosx_spesific_files)
elif sys.platform.lower().startswith('linux'):
    _include_platform_spesifics = dict(linux_spesific_files)
...
setup(
    name='luma',
    version=3.0.6,
    packages=findPackages(),
    _include_cross_pltaform_files,
    **_include_platform_spesifics
)
```

With the new version of the setup script we where able to run a number of various deployment commands, which resulted in successfully building source distribution for both UNIX and Windows, graphical installers for Windows and generic rpm packages for Linux systems:

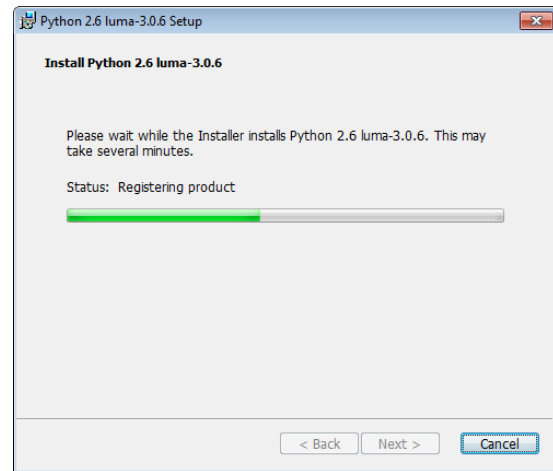
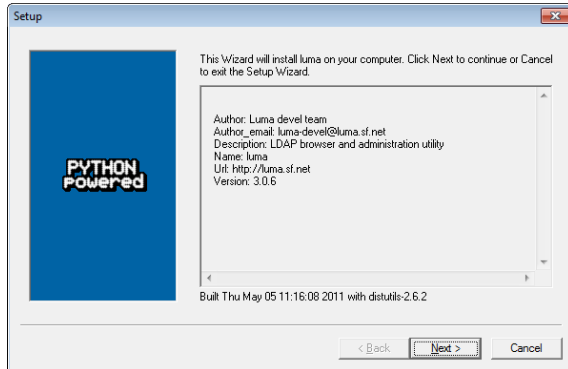


Figure 7.39: Windows graphical exe install.      Figure 7.40: Windows graphical msi install.

```
python setup.py sdist          # Create a source distribution
python setup.py bdist          # Create a built distribution
python setup.py bdist_exe      # Create a windows exe installer
python setup.py bdist_wininst  # Create a windows msi installer
python setup.py bdist_rpm      # create a generic RPM intsaaller
```

Because of the Linux distribution *Fedora* being one of the development platforms in the project, we also created a modified rpm spec file targeting this system in particular. With it we were able to build a Fedora 14 install-able RPM package with the command:

```
rpmbuild -ba SPECS/luma.spec
```

The resulting RPM package was then install-able, either through the command line, or through the graphical *PackageKit* program as illustrated in figure 7.41:

```
yum localinstall luma-3.0.6b-4.fc14.noarch.rpm --nogpgcheck
```

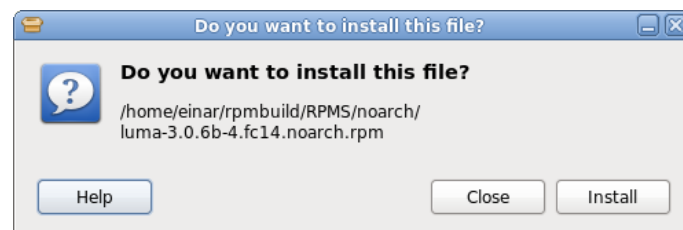


Figure 7.41: Linux graphical RPM install.

### 7.12 Luma Tool Chain

---

As the implementation of Luma 3.0 progressed, the need for preparing and processing various resources, needed by the application, became more and more time consuming. We quickly started to think about ways to automated some of these processes. We started to write some simple wrapper scripts for the vast selection of tools that is shipped with both Qt4 and PyQt4. These scripts were later joined, in what we have named the *Luma Tool Chain*.

#### 7.12.1 Sprint 3

##### Compiling UI files to Python code

In sprint 3 we wrote a simple Python script that wrapped the PyQt4 tool `pyuic4`. This PyQt4 tool is used to generate Python code from the XML based UI files created with the Qt 4 Designer application. Our script was capable of locating and generate Python code for all or just a selection of the available UI files. In addition the script placed the generated Python code into the correct locations, based on the conventions we had defined for the repository structure.

We named the script `lumarcc.py`. This script greatly improved the project development work flow, especially regarding the User Interface development.

#### 7.12.2 Sprint 4

We continuously improved the `lumarcc.py` script while the application development progressed. Whenever a new task became too time consuming with the regular tools, we added new features to our script, resulting in the script expanding its capability.

##### Updating the Luma Project file

All the Luma 3.0 Python source files, Python files generated from UI forms and the resource files, is defined in the `luma.pro` project file. The project file is a utility and essential part of the Qt 4 *qmake* build system. Even though this build system is not available for the PyQt4 bindings, the definitions in the project file is needed to utilize the Qt 4 resource system. The syntax of the file is very straight forward. A section from the Luma 3.0 project file is shown below:

```
...
FORMS += resources/forms/MainWindowDesign.py
SOURCES += luma/luma.py
SOURCES += luma/base/gui/MainWindow.py
TRANSLATIONS += resources/i18n/luma_nb_NO.ts
RESOURCES += luma/resources.py
...
```

Maintaining this file manually is manageable for small projects, but becomes a bit of a challenge when the number of source files and external resources grows in pace with the project progress. This was also the case with Luma 3.0 . By extending the `lumarcc.py` script even further, we incorporated the ability to fully generate and / or update this file dynamically. This is done by executing the following command:

```
python lumarcc.py --update-pro
```

### Updating the Luma Resource file

The Qt 4 resource system provides a powerful cross-platform way of accessing external resources, like icons. All resources is defined in a XML-based text file. An excerpt from the Luma 3.0 resource file, `luma.qrc`, looks like this:

```
<!DOCTYPE RCC><RCC version="1.0">
  <qresource prefix="i18n">
    <file alias="nb_N0">resources/i18n/luma_nb_N0.qm</file>
  </qresource>
  <qresource prefix="icons/32">
    <file alias="list-add">resources/icons/32x32/actions/list-add.png</file>
  </qresource>
</RCC>
```

The `luma.qrc` must be processed with the PyQt4 tool `pyrcc4`, in order to compile the resources, defined in the `luma.qrc` file, into Python code that can be used by the running application. By adding the methods: `updateResourceFile` and `compileResources` to the `lumarcc.py` script, it became able to locate the required resources, fix the correct relative path to each resource, and generate appropriate prefixes and aliases for the files. In the application code, we could access these resources like described in the following code sample:

```
from PyQt4 import QtGui

someWidget.setIcon(QtGui.QIcon('/:icons/32/list-add'))
```

The following command is used to generate or update the `luma.qrc` file:

```
python lumarcc.py --update-qrc
```

#### 7.12.3 Sprint 5

We soon found the `lumarcc.py` script becoming a bit fragmented and hard to extend without breaking present functionality. We therefore decided to refactor the script. The result was a script capable of handling most, if not all, of the preprocessing of resources for the Luma 3.0 application, including compiling UI files to Python code, updating and compiling the icons and translation files and updating the project file.



### 7.12.4 Sprint 6

#### Kickstarting Luma translation

As a result from the improvements in the internationalization support for the application, we found time to extend the *Luma Tool Chain* with a script intended to help future application translators getting started translating Luma 3.0 . The script `lumai18n.py` is capable of creating skeleton Luma 3.0 translation files, with filenames following the specified conventions for Luma 3.0 .`ts` files, as described in section 7.4.4.

To create a skeleton translation file for Brazilian portuguese, one can run the following command from the command line:

```
python lumai18n.py -L pt -C BR
```

The script is also capable of lookup possible matches for languages and countries. The following example demonstrates how the previous translation file also can be created:

```
python lumai18n.py --language po --country br
```

```
More than one match was found for language: po
```

- (0) Polish
- (1) Portuguese

```
Please choose a language: 1
```

```
More than one match was found for country: br
```

- (0) Britain (United Kingdom)
- (1) Gibraltar
- (2) Brazil
- (3) Brunei
- (4) British Indian Ocean Territory

```
Please choose a country: 2
```

```
Luma translation file will be created:
```

```
Destination: resources/i18n/luma_pt_BR.ts
```

```
Translation file succesfully created...
```

### 7.12.5 Summary

What became the *Luma Tool Chain*, started out with a few independent utility scripts, meant to help ourselves in the development work on this project. These scripts then were improved and re-factored to enforce future Luma 3.0 developers in the development and the deployment of the application.

# 8

## Testing

### 8.1 White-box

---

White-box tests are the low-level tests that requires knowledge of the implementation, thus becoming quite complex for larger amounts of code. For our project these were the unit tests that we performed after writing/editing a class or a function. It was important to cover as much as possible with units test early, as debugging a GUI-program is very hard.

#### 8.1.1 Unit-test

We wrote unit-tests for the most important classes in the back-end, that are included in the luma/test folder. The following units were tested:

Table 8.1: Unit test

Class	Unit
LumaConnection	search, bind, missing serverobject, logger, empty serverlist
LumaLogHandler	logger
ServerList	empty list, add, delete, write, getters and setters

### 8.2 Black-box

---

Black-box tests only require knowledge of the relation between input and output. Performing the tests usually consists of performing a few simple steps of input, but as the test area is large, selecting the tests is difficult.

#### 8.2.1 Integration testing

Integration testing is done when merging different functions and classes together. Unit testing has to be done before the integration testing. As we developed different modules in parallel, we had to do integration testing in groups when merging our code together.

Table 8.2: Module integration

Modules needing integration	Sprints affected
Settings Server dialog Luma Main window i18n Installation	1, 2, 3, 4, 5, 6
Ldap back-end	1, 6
Plugin	2, 3
Search	4, 5, 6
Template	4, 5, 6
Browser	2, 3, 4, 5, 6

### 8.2.2 System testing

System tests are tests that concerns the program as a whole, and are therefore after the integration tests. Our system tests were related to the functional, and non-functional requirements specified in the Requirements chapter. We performed the system tests at the sprint review meeting, and together with the customer. The purpose of these tests were whether we had understood the specifications or not.

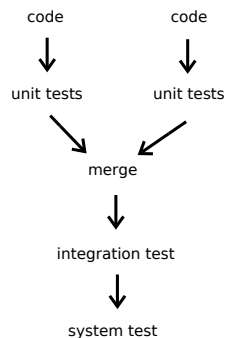


Figure 8.1: Testing structure

## 8.3 Tests features

---

A summary of the most important test features are shown in table 8.3.

## 8.4. FUNCTIONAL REQUIREMENTS TESTING

---

Table 8.3: The most important test features

Functional	Non-functional
LDAP(connect, browse, modify)	Cross platform (Windows/Linux/Unix/Mac OS)
Import to local system	Multilingual support
Plugins	Usability
Logfile	Scalability

### 8.4 Functional requirements testing

---

We tested each functional requirement from Chapter 4 individually. The different tests are shown below.

Table 8.4: Functional requirement test 1.1

Test ID	FT1.1
Test Name	Add LDAP server to the list
Requirement	FR1
Environmental req.	None
Test description	1. Start Luma 2. Open server dialog 3. Add server to the list 4. Restart and check that the server is still there
Test result	OK

Table 8.5: Functional requirement test 1.2

Test ID	FT1.2
Test Name	Edit existing LDAP server configuration
Requirement	FR1
Environmental req.	Server list contains elements
Test description	1. Start Luma 2. Open server dialog 3. Edit the attributes of a server 4. Restart and check that the server has the new values
Test result	OK

## CHAPTER 8. TESTING

---

Table 8.6: Functional requirement test 1.3

Test ID	FT1.3
Test Name	Delete LDAP server from the list
Requirement	FR1
Environmental req.	Server list contains elements
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Open server dialog</li><li>3. Remove a server from the list</li><li>4. Restart and check that the server is not there</li></ol>
Test result	OK

Table 8.7: Functional requirement test 2.1

Test ID	FT2.1
Test Name	Connect and browse a LDAP server
Requirement	FR2
Environmental req.	Server list contains valid server, server is populated
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start the browser plugin</li><li>3. Expand a server-node to connect</li><li>4. Expand its children to browse it</li></ol>
Test result	OK

Table 8.8: Functional requirement test 3.1

Test ID	FT3.1
Test Name	Delete entries on a LDAP server
Requirement	FR3
Environmental req.	Server and entry exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start the browser plugin</li><li>3. Expand a server-node to connect</li><li>4. Delete entries in the tree</li><li>5. Restart and check that the entries are gone</li></ol>
Test result	OK

## 8.4. FUNCTIONAL REQUIREMENTS TESTING

---

Table 8.9: Functional requirement test 4.1

Test ID	FT4.1
Test Name	Add entries on a LDAP server
Requirement	FR4
Environmental req.	Server exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start the browser plugin</li><li>3. Expand a server-node to connect</li><li>4. Add entries to the tree</li><li>5. Restart and check that the entries are there</li></ol>
Test result	OK

Table 8.10: Functional requirement test 5.1

Test ID	FT5.1
Test Name	Edit entries on a LDAP server
Requirement	FR5
Environmental req.	Server and entries exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start the browser plugin</li><li>3. Expand a server-node to connect</li><li>4. Edit entries in the tree</li><li>5. Restart and check that the entries has the correct values</li></ol>
Test result	OK

Table 8.11: Functional requirement test 6.1

Test ID	FT6.1
Test Name	Search the LDAP server
Requirement	FR6
Environmental req.	Server and entries exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start search plugin</li><li>3. Search and check the result</li></ol>
Test result	OK

## CHAPTER 8. TESTING

---

Table 8.12: Functional requirement test 7.1

Test ID	FT7.1
Test Name	Export from LDAP to file
Requirement	FR7
Environmental req.	Server and entries exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Start the browser plugin</li><li>3. Expand a server-node to connect</li><li>4. Edit entries in the tree</li><li>5. Restart and check that the entries has the correct values</li></ol>
Test result	OK

Table 8.13: Functional requirement test 8.1

Test ID	FT8.1
Test Name	Add custom plugins
Requirement	FR8
Environmental req.	Custom-made plugin exists
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Open choose plugin dialog</li><li>3. Select plugin to use</li><li>4. Start the plugin</li></ol>
Test result	OK

Table 8.14: Functional requirement test 9.1

Test ID	FT9.1
Test Name	Error-logging
Requirement	FR9
Environmental req.	
Test description	<ol style="list-style-type: none"><li>1. Start Luma</li><li>2. Do something erroneous</li><li>3. Open the log and check that it was logged</li></ol>
Test result	OK

## 8.4. FUNCTIONAL REQUIREMENTS TESTING

---

Table 8.15: Functional requirement test 10.1

Test ID	FT10.1
Test Name	Specify template-objects
Requirement	FR10
Environmental req.	Server and plugin exists
Test description	1. Start Luma 2. Open the template-plugin 3. Fill in templates 4. Use the template in a plugin
Test result	OK



# 9

## Evaluation

This chapter contains our evaluation of various aspects regarding the project.

### 9.1 Challenges summary

---

In this section we make a short summary of how we responded to the predicted challenges from chapter 1.2.

#### Implement MVC in Qt

While we were already familiar with the concepts of MVC, Qt's implementation took some time to get familiar with. Mainly because of the use of tables in regards to the interface between the model and the view, as discussed in section 7.6. However, after a few weeks working with it, things started to really make sense, and we had no real problems using it.

#### Plugins

It would have been next to impossible to implement the MVC pattern without rewriting the plugin system and plugins since much of the old code was not written in a way were this was feasible. Doing the conversion required substantial knowledge about both Qt and LDAP to perform, and was as such somewhat more time consuming than expected because of the learning time required. Despite this we managed to perform the required conversion while also adding some new features along the way.

#### Cross-platform

This was probably the most interesting challenge to work with. Implementing a cross-platform application turned out to be very educational. We learned about many platform differences, and got to face the challenge in resolving them, in a real application.

The trickiest part was to create a cross platform installation procedure. Even though installation was not part of the requirements when we started the project, the customer, about halfway into the project, articulated that some sort of installation procedure would be desirable. With a good amount of time invested in the problem, we came up with a single script that is capable of producing source distributions for all the major platforms

(Windows/Linux/Unix/Mac OS). We also created automatic graphical installers for the Windows platform (.exe and .msi), and a RPM spec file that successfully builds RPM packages, used by many Linux distributions, including *Red Hat*, *Fedora*, *SUSE*, *OpenSUSE*, and *Mandrake*.

## 9.2 Group structure

---

The roles we set up from the pre studies chapter did not really take effect because we did not run into any major problems along the way requiring the role holder to take a major decision. We don't look at this as a bad thing. Most agile development methods would argue that roles are not necessary; the members are supposed to pitch in and do whatever it takes to make the project successful - regardless of title or role. This is what we did. We could have tried to keep strict roles, but we quickly discovered activities in an agile project does not exist in isolation. Analysis, coding, design and testing are continuous activities, they never end. Take for example the work of Quality Assurance; everyone in the project are responsible for the quality in their own work. Instead of having one person working on quality assurance, we pitched in and helped each other when needed.

## 9.3 Customer relations

---

We had regular contact with our customer. Mostly we communicated over chat and email, but we also had meetings to discuss larger problems and show what we had done. We are very happy with our customer and did not encounter any problems in communication or agreement.

## 9.4 Development method

---

As mentioned in chapter 2 we chose to use an adaptation of Scrum as our development method. We chose to use sprints, product backlog, and sprint backlog. What we did not use was a strict daily stand-up meeting, Scrum-roles, and estimation in hours on the sprint backlogs.

We mostly managed to meet each day, but not always with full attendance. This was because we all had other subjects and classes to attend to, in addition to this project. As a result of this we did not want to give anyone the *Scrum master* role, since no one could meet every day. Instead we arrange for two days each sprint where everyone had to attend. These days was used to coordinate and make group decisions.

Instead of creating hour estimates for the backlog entries, we put together everything we thought we could manage to do in each sprint in the sprint backlogs. This was based on how complex the features and the old code appeared. We did this because in our experience estimating hours rarely turns out accurate, so you end up making unhelpful estimates that is changed during development. This worked out very well for us, we met our own and the customer's expectations on what we should be able to finish within the allotted project time.

## CHAPTER 9. EVALUATION

---

The work for each sprint was planned at the beginning of each sprint. We usually managed to do all the work planned for each sprint, and in cases where we did not finish the work, we passed it on to the next sprint. We always managed to have runnable code in the repository at the end of each sprint.

### 9.5 Tools and utilities

---

An important part of our development process was coordinating work over Redmine. We are happy with the use of Redmine, and it was easy to log hours and see how much time we'd spent on which issues, bugs and features. We highly recommend a project management (web) application like Redmine to anyone who works with an agile project.

As mentioned in the sprint review 6.1 we changed our repository and revision control system from Subversion (SVN) to Git. We felt that Git worked much better when it came to the branching, when starting a new feature, and the later merging once it was complete. Overall we are very pleased with Git, and we would definitely encourage future groups to try it as well.

### 9.6 Social gatherings

---

We have learned that it is important for the group dynamics and general well-being of the group that we reward our self with something not related to the project. After major milestones, we had social gatherings with food and maybe a movie. These events have been very important to us, since they allowed us to get to know each other better outside of the project environment. This is also something we would encourage later project groups to do.

### 9.7 Further work

---

The project purpose was to make a working prototype and we feel we have definitely met this goal and even exceeded it. We would argue that Luma 3 is already quite far on the way to become a release candidate as it has already got the basic functionality of Luma 2.4 implemented. There are however still some features we feel should be added before it reaches the status of a release candidate, including some which are still missing from Luma 2.4 in addition to others.

#### 9.7.1 Search plugin

Further work that could be done on the search plugin includes:

- Open search result in browser
- Exporting/deleting search result
- Filter validation, controlling filter for syntax errors before sending search

### 9.7.2 More plugins

Luma 3 does not yet include these plugins which exists in Luma 2.4:

- Addressbook
- Massive User Creation
- Admin Utilities
- Schema browser
- Usermanagement

### 9.7.3 Installation packages

The libraries for Qt 4 and python-ldap needs to be installed on the machine to run Luma 3.0 . Up until now Luma 3.0 has never had and still does not have an installer that also contains these dependencies and installs them for you if needed. This could be added in the future for ease the installation. Fortunately we have explained how to set up this in the user manual.

# 10

## Conclusion

We have successfully implemented a solution that satisfies the requirements of the customer. The application works and exceeds our customer's initial expectations of just receiving a prototype at the projects end. To confirm this we received the following feedback from our customer:

*The students have been very efficient and have shown me they can work independently. The task at hand was specific, but also with room for suggestions for improved user experience as well as better code. The project members have excelled my expectations on this matter. We have established communication channels through email, irc and meetings on campus.*

*The students have shown initiative, ran a versioncontrolsystem of their own choice, and let me have full project insight using webbased project management system including Gant-chart and issue-tracking. The previous codebase has made its way into major Linux distributions such as Ubuntu, Debian, Mandrake, Suse and others. I'm convinced that the product from this project in time also will be found in major Linux distributions as well as direct download for other platforms such as Windows and OS X - and last but not least: benefit many happy LDAP administrators world wide.*

Mvh

Bjørn Ove Grøtan

### 10.1 Comparison of Luma 2.4 and Luma 3.0

---

In this section we will make a short summary that compares our solution, Luma 3.0 , to the previous, Luma 2.4. We will list those features that are new or missing.

#### 10.1.1 Comparing functional features

**Tabs** In contrast to Luma 2.4, *Luma 3.0* uses tabs to organize the various widgets and plugins for the application. This improvement makes Luma 3.0 more modern and intuitive than Luma 2.4. Upon starting Luma 3.0 for the first time the user is also greeted with a *welcome tab*, with instruction on how to get started. This is also an improvements to the usability for the application.

## 10.1. COMPARISON OF LUMA 2.4 AND LUMA 3.0

---

**Key binding** In contrast to Luma 2.4, *Luma 3.0* has convenient key bindings implemented, with the *superuser* in mind. The various key bindings follow both platform and standardized conventions for commonly used key bindings.

**Logger and error handling** In contrast to Luma 2.4, *Luma 3.0* has better logging facilities. Like Luma 2.4 it is possible to filter the logging based on message categories (error messages, debug messages and informative messages), but in this version the handling of errors is more meaningful to the user.

**Command line arguments** In contrast to Luma 2.4, *Luma 3.0* has support for a number of command line arguments, including clearing configuration files and other files created by the application.

**Search plugin** Like Luma 2.4, *Luma 3.0* also has a search plugin implemented. However, this version has a much improved user interface, filter utilities and support for multiple result views. In this version it is also possible to do additional filtering on a returned result set. In contrast to Luma 2.4 it is not possible to export and delete entries from the search plugin in *Luma 3.0*.

**Browser plugin** In contrast to Luma 2.4, *Luma 3.0* has support for customizing the entry view through HTML templates. Also it is now possible to ignore missing server schema so that user can still view the content on the server.

### 10.1.2 Comparing non-functional features

**Cross-platform** In contrast to Luma 2.4 being a UNIX only application, *Luma 3.0* is now able to be installed and used on multiple platforms, including Windows/Linux/Unix/-Mac OS.

**MVC** In contrast to Luma 2.4, *Luma 3.0* now implements a proper MVC pattern.

**Current technology** With the removal of all legacy PyQt3 code in Luma 2.4, *Luma 3.0* no longer suffers from deprecated technology.

**Speed** Luma is now faster as mentioned in section 7.6.5.

**Less bugs** The development of Luma 2.4 have been stalled for a long time. During this time more bugs have been reported on the bug tracker. In *Luma 3.0* these and other bugs are taken care of and removed.

**Supported plugins** As a result of only a selection of the plugins have been ported to PyQt4, Luma 2.4 still has a wider selection of plugins available, than *Luma 3.0*.

# Bibliography

- [1] About luma. <http://luma.sourceforge.net/about.html>.
- [2] Advantages of user stories for requirements. <http://www.mountaingoatsoftware.com/articles/27-advantages-of-user-stories-for-requirements/>.
- [3] Apache subversion. <http://subversion.apache.org/>.
- [4] Apple human interface guidelines. <http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines/XHIGIntro/XHIGIntro.html>.
- [5] Argonne ldap browser/editor. [http://www.anl.gov/techtransfer/Software\\_Shop/LDAP/LDAP.html](http://www.anl.gov/techtransfer/Software_Shop/LDAP/LDAP.html).
- [6] Dia. <http://live.gnome.org/Dia>.
- [7] Eclipse (integrated development environment). <http://www.eclipse.org/>.
- [8] Git (fast version control system). <http://git-scm.com/>.
- [9] Gnome human interface guidelines. <http://library.gnome.org/devel/hig-book/stable/>.
- [10] Gnu general public license. <http://www.gnu.org/licenses/gpl.html>.
- [11] Gpl-compatible free software licenses. <http://www.gnu.org/licenses/license-list.html#GPLCompatibleLicenses>.
- [12] Jxplorer ldap browser. <http://jxplorer.org/>.
- [13] Kde human interface guidelines. <http://techbase.kde.org/Projects/Usability/HIG>.
- [14] Latex. <http://www.latex-project.org/>.
- [15] Ldap client api for python. <http://www.python-ldap.org/>.
- [16] Luma. <http://luma.sourceforge.net/>.
- [17] Luma bugtracker. [http://sourceforge.net/tracker/?func=browse&group\\_id=89105&atid=589029](http://sourceforge.net/tracker/?func=browse&group_id=89105&atid=589029).
- [18] Miniscrum (online scrum tool). <http://miniscrum.com/>.
- [19] Pydev (python development environment). <http://pydev.org/>.
- [20] PyQt, python bindings for qt. <http://www.riverbankcomputing.co.uk/software/pyqt/intro/>.
- [21] Python programming language. <http://python.org/>.

- [22] Redmine (project management web application). <http://www.redmine.org/>.
- [23] restructuredtext. <http://docutils.sourceforge.net/rst.html>.
- [24] Subclipse (eclipse subversion plugin). <http://subclipse.tigris.org/>.
- [25] Symlabs ldap browser. <http://symlabs.com/products/ldap-browser/>.
- [26] Windows user experience interaction guidelines. <http://msdn.microsoft.com/en-us/library/aa511258.aspx>.
- [27] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, first edition, 2001.
- [28] Clayton Donley. *LDAP programming, Management and Integration*. Manning, first edition, 2003.
- [29] Bjørn Ove Grøtan. Grunnlegende teori om ldap kataloger og programmering med ldap. Hftet er utarbeidet på bakgrunn av workshop i regi av Uninett april 2001, 2005.
- [30] Ben Schneider and Catherine Plaisant. *Designing the User Interface*. Pearson, fifth edition, 2005.





## Official meetings

### A.1 Meetings with customer

---

Date	Venue	Agenda
2011-01-21	S22, Sentralbygg 2	Start-up meeting, presentation of task with requirements
2011-01-31	EL4, Elektro bygget	Introductory course in LDAP
2011-02-14	irc.freenode.net : #luma	Demonstration, feedback and discussion of the results from sprint 1
2011-03-01	S22, Sentralbygg 2	Demonstration, feedback, and discussing the priorities of sprint 3
2011-03-24	S22, Sentralbygg 2	Customer could not meet, took meeting over IRC
2011-04-29	S22, Sentralbygg 2	Demonstration and feedback
2011-03-01	S22, Sentralbygg 2	Demonstration, feedback, and discussing the priorities of sprint 3

Table A.1: Overview: meetings with customer

**A.2 Meetings with supervisor**

---

Date	Venue	Agenda
2011-01-19	ITV 242, IT bygget	Getting to know each other, organize group, requirements elicitation
2011-01-28	R80, Realfagsbygget	Feedback on preliminary report
2011-02-11	R81, Realfagsbygget	How to improve on project report before deadline on mid-semester
2011-02-25	ITV 464, It bygget	Feedback on the midterm report, demonstration of the work after sprint 2
2011-03-11	ITV 464, It bygget	Feedback on the report, demonstration of the work after sprint 3
2011-03-25	ITV 464, It bygget	Feedback on report and demonstration
2011-04-15	ITV464, It bygget	Feedback on report and demonstration
2011-02-11	R81, Realfagsbygget	How to improve on project report before deadline on mid-semester
2011-02-25	ITV 464, IT bygget	Feedback on the midterm report, demonstration of the work after sprint 2
2011-03-11	ITV 464, IT bygget	Feedback on the report, demonstration of the work after sprint 3

Table A.2: Overview: meetings with supervisor

# B

## Crash course in LDAP

### B.1 What is LDAP

---

LDAP stands for Lightweight Directory Access Protocol, and is a lightweight protocol for accessing directory services. It was created in the mid 1990s as a response to the *heavier* X.500 Directory Access Protocol (DAP) which required the Open System Interconnection (OSI) protocol stack to operate. Instead of the OSI stack, LDAP uses only TCP/IP to access the directory services and is as such both easier to implement and more suitable for use over the internet.

### B.2 Directories and directory services

---

A *Directory* is simply a collection of information. In the computer world, directories exist everywhere. It is similar to a database, but tends to contain more descriptive, attribute-based information. Due to the fact that information in a directory is generally read more often than it is written, directories are tuned to give quick-response to high-volume lookup or search operations[28]. As LDAP is a protocol it only defines how to access the directory services, not how the services themselves operate or store the internal data. As such there are many implementations of LDAP-enabled services, including Microsoft *Active Directory*, *OpenLDAP*, Apaches's *Directory Server* and Oracle's *Internet Directory*.

### B.3 The LDAP information model

---

The LDAP information model is based on a hierarchical tree-like structure of directory *entries*, as illustrated in figure B.1. Each entry contains a collection of attributes and has a unique *Distinguished Name* (DN). Each attribute in the entry has a *type*<sup>1</sup> and one or more *values*[29]. Which attributes an entry can contain is defined by the objectclasses the entry is part of. An objectclass inherits the attributes from its parent objectclass(es), and they can be defined as either required or optional. For instance, the objectclass **person** requires the presence of the attributes **cn** (common name) and **sn** (surname), while **description** and **telephoneNumbers** are some of the optional attributes for the class.

---

<sup>1</sup>The type is typically a mnemonic strings, i.e. **cn** for common name, or **uid** for user identification.

### B.3. THE LDAP INFORMATION MODEL

The definitions of the attributes as well as the available objectclasses are structured in a *schema*. The basic schemas used by virtually all LDAP-servers are defined in the RFC2252<sup>2</sup> and X.501 standards, but there are naturally more schemas available.

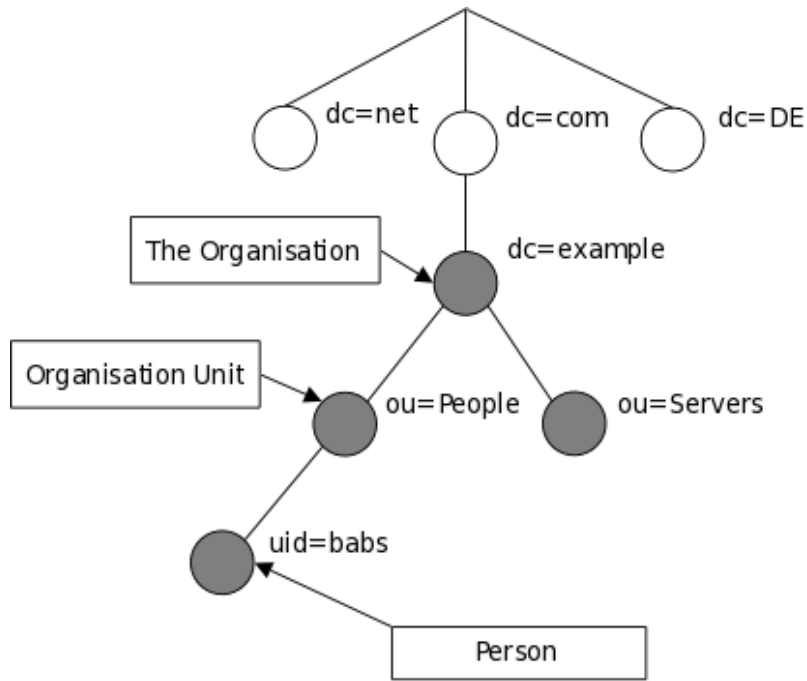
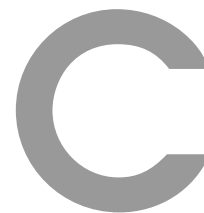


Figure B.1: LDAP directory tree

<sup>2</sup><http://www.ietf.org/rfc/rfc2252.txt>



# Cross-platform GUI development

## C.1 Human Interface Guidelines

---

Developing cross-platform graphical applications is not just a technical issue. Most platforms, with a graphical desktop environment, has its own Human Interface Guidelines [9][13][26][4]. These guidelines are software development documents, which offers application developers recommendations, rules and best practices, to follow on the various platforms. The guidelines aim to improve the user experience, by making application interfaces more intuitive and consistent [30].

Even though the main focus on this project is on the technical part, we taken the time to look into parts of the application where we could improve and take advantage of platform integration.

## C.2 The menu bar

---

The menu bar is a common part of most applications with a graphical user interface. It serves as a place to group various application specific actions, like opening and saving files. The menubar is implemented in various ways in different applications, and platforms. Each platforms Human Interface Guidelines have a section concerning the menu bar. On Windows and the systems running the X Window system, the menu bar is usually anchored at the top of the application window, right beneath the title bar. On Mac OS, however, the menu bar is located at the top of the screen, for all applications.

### C.2.1 Windows and the X Window System

UNIX and UNIX-like systems, such as Linux, typically run the X Window System, with a desktop environment on top of it. Today, GNOME and KDE are the dominant desktop environments for these platforms, and are often installed by default on Linux systems. When developing for platforms running the X Window System it is natural to focus on these environments. Both Windows and platforms using the X Window System usually anchor the menu bar at the top of the application window, and the Human Interface Guidelines for Windows, GNOME and KDE, are also quite similar when describing the recommendations for the menu bar. The application menu bar in Luma will be organized as described in table C.1 for both Windows and the X Window System.

Table C.1: The layout of the menubar on Windows and X Window Systems.

File	Edit	View	Help
Quit	Serverlist	Toolbar	About Luma
	Reload Plugins	Statusbar	
	Configure Plugins	Logger Window	
	Language	Fullscreen	
	Preferences		

### C.2.2 Mac OS

The menu bar on Mac OS systems is located at the top of the screen, as is visible only for the application currently active. The recommendations and conventions for grouping menu bar actions on Mac OS, differs from the recommendations and conventions used on Windows and X Window Systems. The Luma menu bar on Mac OS will be organized as described in table C.2.

Table C.2: The layout of the menubar on Mac OS.

Application	Edit	View
About Luma	Serverlist	Toolbar
Quit Luma	Reload Plugins	Statusbar
	Configure Plugins	Logger Window
	Language	Fullscreen
	Preferences	

## C.3 Icons

---

Systems using the X Window System (UNIX and UNIX-like systems like Linux), make use of so called icon-themes. These icon themes lets the user customize and control what icons is used by application. Standards and specifications for icon-themes is defined by the freedesktop.org project <sup>1</sup> Unlike these systems, Windows and Mac OS, does not use system wide icon-themes. When developing cross platform applications we need to include application default icons for these platforms. In order to still let the users of X Window Systems take advantage of the icon-theming abilities, we can ship out own icon theme, and follow the icon naming specifications defined by the freedesktop.org project.

The Qt application framework, provides a mechanism that enables us to lookup icons

---

<sup>1</sup>freedesktop.org is a open source/open discussion software project working on interoperability and shared technology for X Window System desktops

## APPENDIX C. CROSS-PLATFORM GUI DEVELOPMENT

---

from icon-themes on platforms that support icon-themes <sup>2</sup>, and provide fall back icons that will be used on platforms that don't support it.

```
icon = QIcon.fromTheme('icon-name', QIcon('fallback-icon'))
```

**Example:** *Qt icon-theme lookup with fallback*

---

<sup>2</sup>With the Qt Application framework, the supported platforms is the ones that run the X Window System



## Cross-platform Python deployment

Python is an interpreted language, which means an interpreter program is needed to execute the code. This differs from compiled languages like C, where the code, after being compiled, is executed on the host CPU. Deployment of applications written in interpreted languages becomes a little bit more involved because of the need for an interpreter program. Basically we have to deal with two scenarios when deciding on the deployment process.

In system environments where a Python interpreter is already installed, the deployment can be very straightforward. You can simply distribute the source code, and make sure the executable code is in the `PYTHONPATH`<sup>1</sup>. This can be achieved either by providing an install script that facilitates Python's own standard library in order to locate the path, or leave it up to the user to ensure that the application is found by the Python interpreter. On most UNIX systems this becomes a most likely scenario, because Python is already installed by default.

On Windows, however, Python is not installed by default, which introduces scenario two; system environments with no Python interpreter. One solution, and perhaps the easiest, would be to hand the user a list of all the required software, and tell him to install these before installing our application. If the user got some technical skills, this might not be too much to ask, but it might eliminate other, not-so-technical users. In order to support the last group of users, we ship our application bundled with a customized Python environment. This way the user gets the needed interpreter bundled together with the needed Python library modules, and our own application source code.

The last scenario with the bundled distribution would result in a much bigger installation size, compared to only shipping the application source code. But installation size, must be compared with ease of use.

### D.1 Available tools

---

There exists a selection of tools for deploying and distributing Python applications. Some are platform independent, and some are targeting specific platforms.

---

<sup>1</sup>The `PYTHONPATH` contains a list of directories where Python looks for modules that can be imported. By default these directories contain modules in the standard Python library.



### D.1.1 Distutils

Distutils is part of the Python standard library, and is the standard mechanism for distributing Python packages and extensions. By writing a basic Python setup script (`setup.py`), and possible configuration definitions (`setup.cfg`), the distutils module supports creating both source and built distributions. The architecture of Distutils lacks conventions, and there is no default way for uninstalling or updating an installed application.

```
from distutils.core import setup

setup(
    name='luma',
    packages=['luma',
              'luma.base',
              ... ],
    scripts=['bin/luma'],
)
```

```
>>> python setup.py install
```

**Example:** *Distutils: setup.py*

### D.1.2 Setuptools

Setuptools is a third party collection of enhancements to the distutils module. Like distutils, setuptools is able to both build and distribute Python packages. Unlike distutils, it is also able to check for dependencies between Python packages. By default, setuptools distribute Python packages as *eggs*, which is a form of zipped package.

```
from setuptools import setup, find_packages

setup(
    name='luma',
    packages=find_packages(),
    scripts=['bin/luma'],
)
```

```
>>> python setup.py install
```

**Example:** *Distutils: setup.py*

### D.1.3 Py2exe

Py2exe is a third party distutils extension targeting deployment on the Windows platform. It converts Python programs into executable Windows programs, so that no Python installation is required.

```
from distutils.core import setup
import py2exe

setup(
    name='luma',
    packages=['luma',
              'luma.base',
              ... ],
    windows=[
        {
            'script': 'luma.py',
            'icon_resources': [(1, 'resources/icon.ico')],
        }
    ],
    options={'py2exe': {'skip_archive': True, 'includes': ['sip']}}
)

>>> python setup.py install
```

**Example:** *Py2Exe: setup.py*

#### D.1.4 Py2app

Py2app is a third party program that targets the Darwin platform, which is used on Mac OSX. It is an extension to the setuptools collection, and creates standalone Mac OS X application bundles from Python programs.

```
from setuptools import setup
setup(
    name='luma',
    packages=['luma',
              'luma.base',
              ... ],
    setup_requires=['py2app'],
    app=['luma.py'],
    options={
        'py2app': {
            'iconfile': 'resources/icon.icons',
        }
    }
)

>>> python setup.py install
```

**Example:** *Py2app: setup.py*

### D.1.5 PyInstaller

PyInstaller is another third party program that converts Python packages into stand-alone executables. PyInstaller works on both Windows, Linux, and Mac OS X.

```
>>> python Configure.py
>>> python Makespec.py /path/to/luma/src/luma.py
>>> python Build.py specfile
```

**Example:** *PyInstaller*



## Luma User documentation

The following pages contains the *user documentation* we have written for the application. We have used *reStructuredText* [23] to write the user documentation. *reStructuredText* is a plain text syntax markup and parser system capable of producing a vast number of output formats from documents written in plain text. Table E.1 describes the various output formats supported by *reStructuredText*.

Table E.1: Output formats supported by reStructuredText.

Format	Extension	reST tool
HyperText Markup Language	.html	rst2html
Extensible Markup Language	.xml	rst2xml
OpenDocument Text	.odt	rst2odt
Nroff manual pages	.gz	rst2man
Latex	.tex	rst2latex

### E.1 Available formats

---

The source files for the *Luma User documentation* is available in the repository. A complete version of the *Luma User documentation* in *HyperText Markup Language* is currently hosted on one our own ntnu based websites, and can be viewed at:

<http://folk.ntnu.no/einaru/luma/doc>.

For the report we have included the *Luma Userguide* and the *Luma Installation* documents. These was generated from plain text files to L<sup>A</sup>T<sub>E</sub>X files, ready to be processed with a typical L<sup>A</sup>T<sub>E</sub>X preparation tool. Note that the page numbering for the user documentation files do not follow the remainder of the report.

# Luma Userguide

*Luma is LDAP management made easy.*

## 1. Getting started

On UNIX and UNIX-like systems after a successful installation, Luma can be started from a shell, with the following command (provided the startup script is installed in the system `PATH`):

```
$ luma
```

On Windows systems you can start Luma from `cmd.exe`. If you installed from a source distribution this can be done with:

```
$ luma.py
```

If you installed a frozen bundle of Luma, it can be started with:

```
$ luma.exe
```

For an overview of the available commandline options, you can run:

```
$ luma -h
```

If you are running on a UNIX or UNIX-like system, you can read the `luma(1)` man page, by running:

```
$ man luma(1)
```

## 2. Editing the serverlist

If you are running Luma for the first time you will be greeted with a *Welcome Tab*. Here you will find a quick overview of how to get started using the Luma application. The first thing you need to do is to edit the **serverlist**. This can be done by selecting `Edit → Server List` from the menubar or with the keyboard shortcut `Ctrl + Shift + S`.

When adding servers to the serverlist you are presented with many of options. The Server Dialog is basically divided into three section:

- **Network:** Here you can define the server hostname, port number and base DN to use.
- **Authentication:** Here you can choose how to bind to the server.
- **Security:** Here you can choose your connection type, server certificate and client certificate.

### 3. Activate and use a plugin

When you have added servers to the serverlist its time to manage the available plugins. By default no plugins is activated. To activate a plugin you must select `Edit → Settings` from the menubar to open the settings dialog, and select the plugins tab. A list of all available plugins will be shown, and you can select the plugins you can to activate. If the plugin support it, you can also edit plugin specific settings from the same dialog.

When you have activated one or more plugins and closed the settings dialog, the activated plugins will be listed in the *Plugins tab* in the main window. (If the *Plugins tab* is closed you can open it by selecting `View → Show Plugin List` from the menubar). To open a plugin you simply double click it, and a new tab will be created for the plugin. This way you can have multiple plugins open at the same time.

#### 3.1. Available plugins

The base Luma application includes a number of plugins which brings and extends functionality to the application. configure the plugin settings (if the plugin supports this).

##### 3.1.1. Browser

The browser consists of the list of server-trees on the left, and the entry-view on the right.

###### Server tree

The following operations is available when right clicking a node: Edit server settings

- **Open:** Loads the selected object, if not already loaded, and displays it in the entry-view.
- **Reload:** Removes the children from memory, and reloads them.
- **Clear:** Remove the children from memory.
- **Set filter:** Set the search filter used to collect the children, and reloads them.
- **Set limit:** Set the limit of the nodes children.
- **Add:** Add an entry using the selected nodes DN, and the selected template.
- **Delete:** Delete the node on the server.
- **Export:** Export the entry to file, with subtree or with subtree and parent.

###### Entry-view

The entry-view has the following functionality (on top):

- **Reload:** Reload current entry, asks whether to save if it has been modified.
- **Save:** Save changes done to the current entry.
- **Add attribute:** Opens a dialog where you can select attributes from a list, and add them to the entry.

- **Delete object:** Deletes the entry from server, if it is a leaf node.
- **Switch between views:** A drop down list where you can select views.

And the following inside the document:

- **Delete objectclass:** A red cross behind the object class, deletes the object, and the attributes that are no longer supported.
- **Edit attribute (RDN):** Add a attribute as RDN (only on CREATE).
- **Edit attribute (Password):** Type a password, only ascii is allowed when using encryption.
- **Edit attribute (Binary):** Edit a binary attribute, opens a file dialog.
- **Edit attribute (Normal):** Plain text input (Not allowed for RDN).
- **Delete attribute:** Delete the attribute (Not allowed for must with only one value)
- **Export binary:** Export the value to file.

### 3.1.2 Templates

The template plugin is for making templates to use in making ldap entries. On the left side is the template list and it shows existing templates. The right side is the template view and shows information on the selected template.

#### Template list

The four buttons under the template list is:

- **Add:** Opens a dialog for adding a new template.
- **Save:** Saves the template list to disk.
- **Delete:** Deletes the selected template.
- **Duplicate:** Duplicates the selected template.

#### Template view

In the upper part of the template view is the name of the server, which the template belongs to, and the description of the template. In the *Objectclasses* section the template's objectclasses is shown and can be added and deleted. The *Attributes* section works the same way as the *Objectclasses* section, but you can also change the *must* of an attribute with no *must* to a blue checkmark. This is to show that you want the template to force an attribute to get a value when made in the browser plugin.

### 3.1.3. Search

The search plugin supports arbitrary LDAP search operations on a selected server. The plugin also includes a convenient Filter builder, which can be used to build complex LDAP search filters.

#### Search form

In the *Search form* you select the server you wish to do a search operation on, the Base DN you wish to connect with, the search level, and a possible size limit for the search. The search level options is:

- **SCOPE\_BASE:**
- **SCOPE\_ONELEVEL:**
- **SCOPE\_SUBTREE:** (*Default*)

The *size limit* defines the maximum number of matches to retrieve from the search operation. The default value is 0 (which is the same as None).

To perform a search you simply select a server, enter a search filter and click on the *Search* button. If the search filter contains no syntax errors, a result view is displayed at the left. If the search filter contains syntax errors, an error message is displayed. It is also possible to continue editing the filter in the *Filter builder*, by clicking the tool button next to the Search button.

#### **Filter builder**

The *Filter builder* is intended to help the you construct complex LDAP search filters. Based on the currently selected server you are presented with a complete list of object classes and attributes that is supported on the server.

The *Filter builder* is divided into a search criteria component creator, and a filter editor. When you create a search filter criteria, you insert it into the editor. The component will be inserted at the cursor position in the editor. In the filter editor you are able to perform various operations on selections. This includes to *or*, *and* or *negate* a selection of the search filter. You also is able to insert escaped special characters into the filter.

Filters created in the *Filter builder* follows the *LDAP String Representation of Search Filters* specifications defined in RFC4514<sup>1</sup>.

#### **Result view**

When a search operation successfully returns. The matching LDAP entries are displayed in a new tab. The search result is displayed in a table view. The columns in this table represents the DN plus one column for every attribute used in the search filter.

It is also possible to do additional filtering on columns in the result view. To open the result view filter box you can use the keyboard shortcut `Ctrl + F`. Here you can choose the filtering syntax to use and the column to apply the filter on.

The available filter syntaxes is:

- **Fixed String:**
- **Regular Expression:** Note that this option can be very slow on large result sets.
- **Wildcard:** Note that this option can be very slow on large result sets.

## **4. Luma keyboard shortcuts**

Keyboard shortcut	Action
<b>Main Application</b>	
Ctrl + L	Toggles the <i>Logger Window</i>
Ctrl + P	Show the <i>Plugin List</i>

... continued on next page



Keyboard shortcut	Action
Ctrl + Q	Quit the application
Ctrl + W	Close the currently selected tab
Ctrl + Shift + S	Open the Server dialog
Ctrl + Shift + W	Show the <i>Welcome Tab</i>
F5	Reload the plugins
F11	Toggle fullscreen mode
F12	Open the <i>About Dialog</i>
<b>Search plugin</b>	
Ctrl + F	Opens the filterbox in a result view
Ctrl + W	Close the currently selected result view tab
<b>Browser plugin</b>	
Ctrl + W	Close the current entry tab

On Os X Ctrl is replaced with Meta

## 5. Problems and bugs

Luma tries to provide relevant feedback to the user, when illegal operations, errors and/or other problems occur. If you encounter some issues where the application feedback is missing, you can try to start the application from a shell with the `-v` or `--verbose` option:

```
$ luma --verbose
```

This will print more information, about what is going on, to *standard out*. It is also possible to view *Error*, *Debug* and *Info* messages, produced by the application, in [5.1. The Luma Logger Window](#).

### 5.1. The Luma Logger Window

The *Logger Window* is not displayed by default. To display it you can select `View → Logger Window` from the menu bar, or use the keyboard shortcut `Ctrl + L`. If you want the *Logger Window* to be displayed every time you start Luma you can select this in the *Settings Dialog* (`Edit → Settings` in the menubar).

The *Logger Window* can be customized to display only selected types of messages. The message that Luma produces is categorized in:

- **Error:** Messages for things that have gone wrong.
- **Debug:** Messages mostly intended for the developers to hunt down various issues. Some of these messages can be of great value when a problem occurs.
- **Info:** Messages that only contain verbose information of things that happen successfully.

### 5.2. Reporting bugs

The Luma bugtracker can be found here: [http://sourceforge.net/tracker/?group\\_id=89105](http://sourceforge.net/tracker/?group_id=89105).

## **6. Contact and support**

Application news and contact information can be found on the official Luma website <http://luma.sf.net/>.

## **Footnotes**

---

<sup>1</sup><http://tools.ietf.org/html/rfc4515>

# Luma Installation

## 1. Platforms and dependencies

Luma is a crossplatform application written in the Python programming language. It is developed and continuously tested on a number of platforms and operating systems. The development platforms include:

- Fedora: 14 (GNOME)
- Ubuntu: 10.04, 10.10
- Microsoft Windows: Vista, 7
- Mac OS X

In addition Luma is tested successfully on the following platforms:

- Fedora: 14 (KDE)
- Chakra GNU/Linux: 2011.02 (Cyrus)
- Microsoft Windows: XP

In order to successfully install and run Luma, you will need to install the following libraries on your system:

- Python  $\geq 2.6 < 3$
- python-ldap  $\geq 2.3$
- PyQt4  $\geq 4.8$

You can install and/or run Luma in a number of ways. This includes:

- [2. Installing Luma from a source distribution](#)
- [3. Installing a prepackaged distribution of Luma](#)

## 2. Installing Luma from a source distribution

Source distribution for Luma is available for installation using the `distutils` modules in the standard python library. Source distributions can be downloaded as tarballs `.tar.gz` (UNIX and UNIX-like) or as zipped archives (Windows) `.zip`. If you are running on Linux you could see if your distribution provides prepackaged distributions of Luma ([3. Installing a prepackaged distribution of Luma](#))

## 2.1. Installing the latest tarball

If you are installing on UNIX and UNIX-like systems, you should use the latest tarball. The tarballs is known to install without any problem on both Linux and Mac OS X. Luma is easily installed with the following commands:

```
$ tar xvzf luma-3.0.6.tar.gz
$ cd luma-3.0.6
$ sudo python setup.py install
```

## 2.2. Installing the latest zipped archive

If you are installing on Microsoft Windows you should download the latest zipped archive, and open your `cmd.exe`:

```
$ unzip luma-3.0.6.zip
$ cd luma-3.0.6
$ python.exe setup.py install
```

## 3. Installing a prepackaged distribution of Luma

The following platform specific Luma packages are available:

### 3.1. Linux

#### Fedora 14

Testbuilds is currently available at <http://folk.ntnu.no/einaru/luma/dist>. To install run:

```
$ wget http://folk.ntnu.no/einaru/luma/dist/luma-3.0.6b-4.fc14.noarch.rpm
$ yum localinstall luma-3.0.6b-1.fc14.noarch.rpm --nogpgcheck
```

### 3.2. Microsoft Windows

For Microsoft Windows there exists both `.exe` and `.msi` installers. Note that all the required dependencies must be installed seperately.