Final Report — **SP Light**

Group 7 — TDT4290 Customer Driven Project

Magne Bergfjord
Christian Bøhn
Erik Drolshammer
Leif Christian Larsen
Per Ottar Ribe Pahr

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Contents

List of Figures

List of Tables

Listings

Preface

This project started on September 1., 2005. Our task was to develop a prototype of a tool for architecture and performance modeling based on the Structure and Performance modeling language. Our customers were Department of Computer and Information Science at the Norwegian University of Science and Technology and Modicum Ltd. of United Kingdom.

About 12 weeks later, we delivered the software tool SP Light and this report to the customer. We are confident that we have produced a product of great value and with many possible future uses.

Trondheim, 24th of November, 2005

Group 7 of TDT4290 Customer Driven Project

_____

Erik Drolshammer
Project Manager

_____	_____
Magne Bergfjord	Christian Bøhn
_____	_____
Leif Christian Larsen	Per Ottar Ribe Pahr

Chapter 1

Introduction

In this chapter, we will give a brief overview of the contents of this report.

1.1 Overview

This report contains all formal documents produced during the project, and is divided into the following chapters:

- **Project Definition Plan**. This chapter provides a formal description of the project work. It defines the project, its goals and interested parties, how the group will work with the project and how risk, quality and time are managed throughout the project.
- **Pilot Study**. This chapter was written to get a thorough understanding of the problem the customer wants us to solve. It identifies, explores and evaluates different strategies for solving the problem, and chooses one of them.
- **Requirements Specification**. The requirements specification provides a formal description of the requirements to our solution. This chapter complies with the IEEE Standard 830–1998: IEEE recommended practice for software requirements specifications (see [3B]).
- **Design**. This chapter provides a description of how we will create the system specified in the requirements specification. It provides both textual and graphical descriptions of how our system is designed.
- **Implementation**. This chapter shows how we have implemented the design specified in the design chapter. We shall look at how our implementation works, and also how it implements the requirements of the requirements specification.
- **Testing**. This chapter describes how testing is done throughout the project. It describes test methodology, specific tests and what the results of the tests were.
- **Evaluation**. This chapter evaluates the project and the course. We will evaluate the group process and the group dynamics, internal and exter-

nal communication throughout the project, what we have achieved, and possible future extensions to our work.

1.2 Terms

We have in the interest of readability decided to use some abbreviations and terms throughout the project. The abbreviations we use are listed in the Abbreviations and Terms section on page 173.

Chapter 2

Project Definition Plan

2.1 Introduction

This is the project definition plan (PDP) for group 7 in the course TDT4290 Customer Driven Project at IDI, NTNU.

The aim of this document is to provide a formal description of the project, *how* we are going to work with the project and how we will monitor risks, quality and progress.

The following is included and described by this document:

- **Project mandate.** This provides formal information about the project: name, interests and goals.
- **Project plan.** This provides a progress plan, information on human resources, a description of the different project phases and milestones, and also a time table showing the times at which the team will work each week.
- **Project organization.** This provides information about the group structure and roles of the individual group members.
- **Templates and standards.** This shows different document templates that are to be used in the project, in addition to setting project standards for coding style, document formats and language.
- **Version control.** This describes how the group will use version control to maintain control over document and source code revisions.
- **Project follow-up.** This section describes how project follow-up with respect to time, risk, quality, cost and extent will be done.
- **Quality assurance.** Since this project is highly oriented towards creating a stable and usable product, quality is of great concern. This section describes how we will work to ensure that a quality product is delivered to the customer.
- **Test plans.** This section describes how we intend to test our product to discover errors.

2.2 Project Mandate

This section gives a short introduction to the project and the limitations we have.

2.2.1 Project Name

The project name is "Tool for architecture and performance modeling". The working title for the software being developed is SP Light.

2.2.2 Customer

The customers for this project are The Department of Computer and Information Science at NTNU and Modicum Ltd, UK.

2.2.3 Stakeholders

With this project, we hope to meet the demands of the following groups:

- People designing and analyzing computer architecture based on performance requests.
- Developers trying to make SP better.
- Students learning the SP language.

For partners during the project development, see Appendix A.1.1.

2.2.4 Background

The Structural and Performance (SP) modeling language was invented more than 20 years ago, but it does not have a tool to make modeling easy. Today's method is to draw the models in MS Visio [22W] and to do the calculations i MS Excel.

2.2.5 Effect

The customer wishes that this project will contribute to the following:

- Easier to make SP models, even for people not being very familiar with the SP modeling language.
- Easier to present SP at conferences dealing with architectural modeling.
- Easier to make extensions to SP and see what changes SP needs.

2.2.6 Result

To have the above effect, this project should end up with these results:

- A graphical modeling tool for SP.
- Performance calculation automatically based on values connected to the graphical items.
- Possibility to add extensions and modifications to the tool.

2.2.7 Extent of Project

Our aim is to deliver a lightweight modeling framework that can be installed on any Java compatible PC or laptop. The framework will keep the most important modeling aspects and can be extended to include new rules and objects.

2.2.8 External Conditions

Material resources

- A PC made available by IDI. In addition comes the private PCs of the team members.
- A printer quota of 500 pages per person.
- Freely available copier at IDI.
- Group directory at NTNU, 1GB.
- Mailing lists at IDI.
- TDT4290 Home page with earlier reports.

2.2.9 Economy

The project team consists of 5 persons, each contributing with 310 hours, which sums to 1550 hours for the whole project.

2.2.10 Time

SP Light will be delivered to the customer at November 24, 2005.

2.3 Project Plan

The project plan is a dynamic document describing *what* to do and *when* to do it. It explains the segmentation of the project into phases and states when to start each phase. We believe these phases can be performed in an approximately chronological order, so we have chosen to follow the traditional waterfall model, as shown in Figure 2.1 below. As the example below shows, the main idea is to finish one phase, before moving on to the next. There will always be some iteration, be we hope to keep this at a minimum.

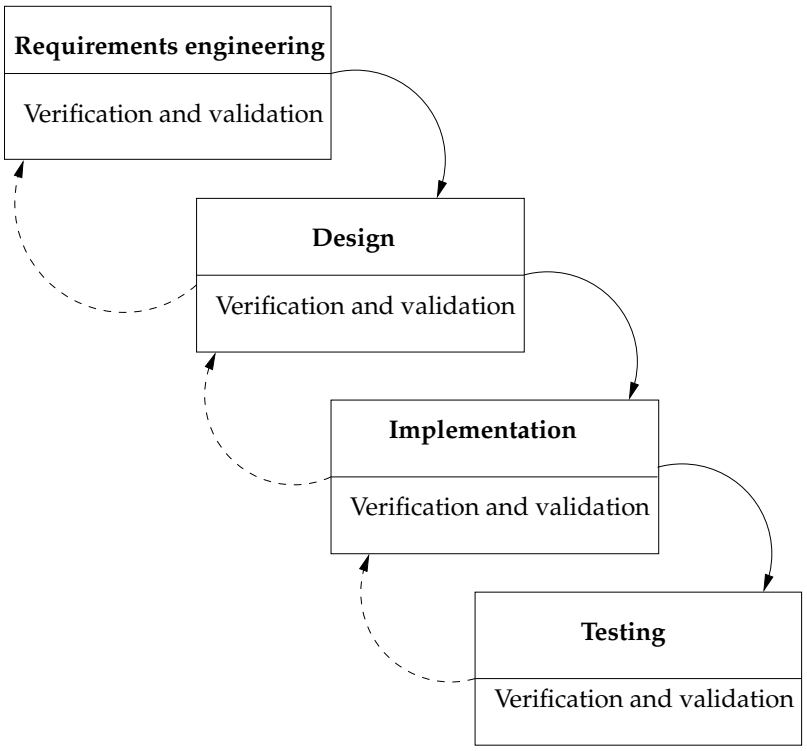


Figure 2.1: The waterfall development model

2.3.1 Human Resources

Table 2.1 on the next page shows how much human resources we plan to spend on each phase. Where we have chosen to diverge from the norm [1], it is to reflect that we already have a requirements specification.

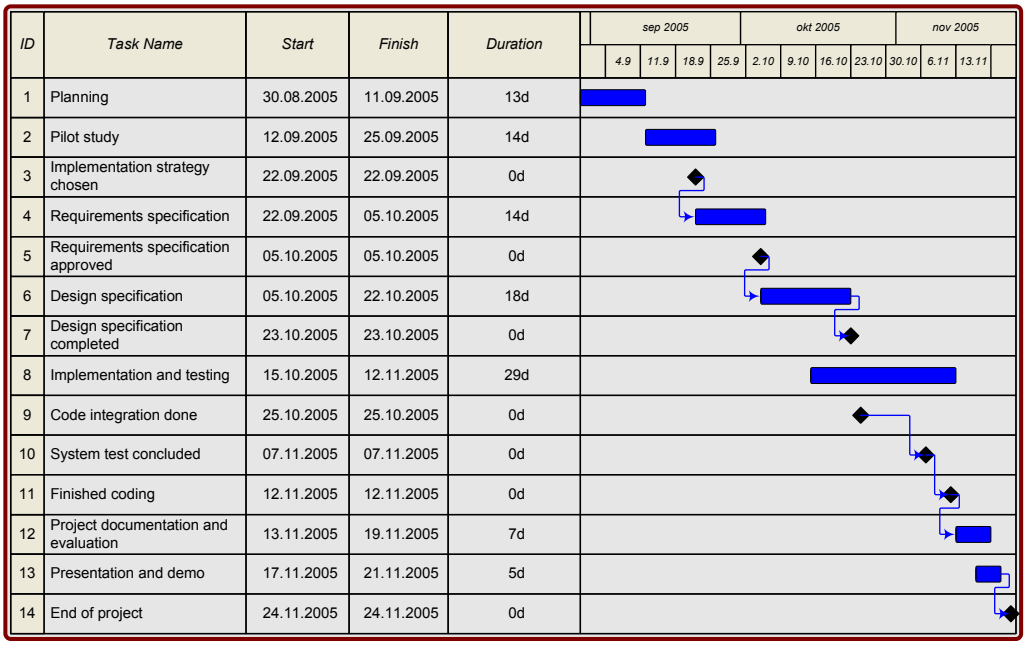
[1] Norm is the statistical mean over earlier groups in Customer Driven Project.

Phase	Norm (%)	Norm (h)	Gr.7 (%)	Gr.7 (h)
Project Management	10	155	10	**155**
Lectures and self tuition	10	155	13	**200**
Planning	7	108,5	9	**140**
Pilot study	15	232,5	10	**155**
Requirements specification	20	310	8	**124**
Design specification	15	232,5	20	**310**
Implementation and testing	13	201,5	20	**310**
Documentation	0	0	3	**47**
Project evaluation	5	77,5	5	**78**
Presentation and demo	5	77,5	5	**78**

Table 2.1: Scheduled time usage for different project phases

2.3.2 Progress Plan

The progress plan is realized as the Gantt diagram shown in Figure 2.2. We did not include the tasks "Project management" and "Lectures and self tuition" in the Gantt diagram as these tasks span the entire project.



ID	Task Name	Start	Finish	Duration
1	Planning	30.08.2005	11.09.2005	13d
2	Pilot study	12.09.2005	25.09.2005	14d
3	Implementation strategy chosen	22.09.2005	22.09.2005	0d
4	Requirements specification	22.09.2005	05.10.2005	14d
5	Requirements specification approved	05.10.2005	05.10.2005	0d
6	Design specification	05.10.2005	22.10.2005	18d
7	Design specification completed	23.10.2005	23.10.2005	0d
8	Implementation and testing	15.10.2005	12.11.2005	29d
9	Code integration done	25.10.2005	25.10.2005	0d
10	System test concluded	07.11.2005	07.11.2005	0d
11	Finished coding	12.11.2005	12.11.2005	0d
12	Project documentation and evaluation	13.11.2005	19.11.2005	7d
13	Presentation and demo	17.11.2005	21.11.2005	5d
14	End of project	24.11.2005	24.11.2005	0d

Figure 2.2: Gantt diagram of the project

2.3.3 Project Phases

The project is divided in phases as shown in Figure 2 and 3.
A phase may have certain activities to be performed or milestones to be reached and these will be described here. For the important phases more detailed descriptions can be found in the phase documents.

Project Management

This phase spans the whole project. Important activities are delegation of tasks in the group, contact with the customer and planning of meetings. Completion of the project is a milestone for this phase.

Lectures and Self Tuition

This is not a phase directly related to the project, but it is included because it is used for time tracking. It includes activities like lectures, the trip to Røros and reading specifications.

Phase descriptions

The following phase-plans will be described in the appendix because they are subject to change as we develop a better understanding of the problems involved. For each of these phases detailed phase documents will be written. The phase documents will be included in the final project report.

- Appendix A.2.1 — Planning
- Appendix A.2.2 — Pilot Study
- Appendix A.2.3 — Requirements Specification
- Appendix A.2.4 — Design Specification
- Appendix A.2.5 — Implementation and Testing
- Appendix A.2.6 — Project Evaluation and Documentation
- Appendix A.2.7 — Presentation and Demo

2.3.4 Milestones

To motivate the group and to keep track of progress we want to set certain milestones.

- **Implementation strategy chosen.** We are done exploring possible frameworks and we have decided on how we are going to implement the solution.
- **Requirements specification approved.** The requirements specification has been approved by the customer.
- **Design specification completed.** The design specification is final.
- **Code integration done.** We are planning to do the implementation phase in multiple iterations. After the first iteration, we will integrate the modules and work with it as one program from there on, with one code integration early in the development phase.
- **System test concluded.** The system test is done and accepted.
- **Finished coding.** We're done coding.
- **End of project.** The project has been delivered to the customer.

2.4 Organization

The group consists of five members, whom all are students in the fourth year of their Master degree study of Computer Science at the Norwegian University of Science and Technology. Telephone numbers and email addresses can be found in Appendix A.1.2. Every member of the group will be assigned one or more roles. In addition one person will be responsible for each phase. This includes the job as QA responsible of the phase.

2.4.1 Organization Chart

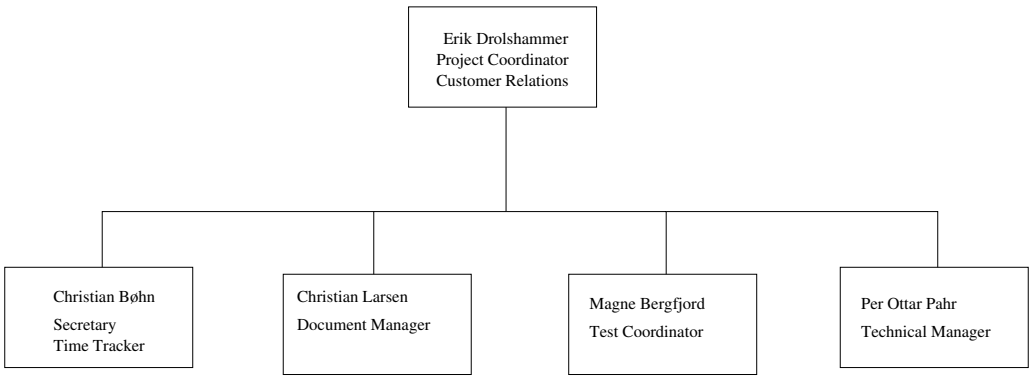


Figure 2.3: Organization Chart

The organization chart is shown in Figure 2.3. We will try to make decisions in group meetings. If this is not possible, the Project Manager and the person responsible for each phase may make the necessary decisions.

2.4.2 Roles

The delegation of roles is shown in Table 2.2.

Roles	**Name**
Project Manager	Erik
Secretary	Christian B.
Customer Relations	Erik
Time Tracker	Christian B.
Document Manager	Christian L.
Test Coordinator	Magne
Technical Manager	Per Ottar
QA	Each member is responsible for a phase.

Table 2.2: Delegation of roles

Project Coordinator

The Project Coordinator is responsible for dividing the workload among the project group. He is also responsible for reaching the designated deadlines on time.

Secretary

The Secretary shall write a report from each meeting and distribute it by email to the interested parties. All reports shall be stored in the group directory.

Customer Relation

The person in this role shall make sure the customer is able to communicate with the group as much/well as possible. He is responsible for meetings with the customer and to make sure the group learns all essential information disclosed at these meetings.

Time Tracker

The Time Tracker keeps track of hours spent and compares it with the project plan. He will work closely with the Project Manager to update the project plan when necessary. In addition he will present a table, weekly, stating how many hours each member of the group worked last week.

Document Manager

The Document Manager is responsible for giving the final report a functional layout. He should also make sure the documents are consistent and written in the same style.

Test Coordinator

The Test Coordinator must ensure that all necessary tests are written in time and that they are performed and approved.

Quality Assurance (QA)

Every person is responsible for the quality of his own work. However, the total quality of a phase is not necessarily equal the sum of the quality of the individual parts of the phase. Therefore each phase of the project will have a QA responsible who is in charge of the overall QA of the phase. The QA responsible for each phase is:

- Planning: Erik

- Pilot study: Christian L.
- Requirements specification: Per Ottar
- Design: Christian B.
- Implementation and Testing: Christian L.
- Project Documentation and Evaluation: Per Ottar
- Presentation and Demonstration: Erik

2.5 Standards and Templates

We will now describe the standards and templates which are going to be used in the project.

2.5.1 Standards

We aim to standardize language, file naming, documents and source code. We will now describe the standards we will follow.

Language

All documents that are to be delivered to the customer will be written in English. This includes comments in the resulting source code. Certain simple documents (primarily meeting summons and meeting reports; see Sections 2.5.2 and 2.5.2) and internal documents will be written in Norwegian. Weekly status reports will be written in Norwegian.

File Naming and Directory Structuring

The standard for naming and structuring the directory structure is described in Section 2.6.

Document Formats

Documents are to be primarily written in LaTeX and made available in PDF format. Simple documents (particularly documents that are to be transferred via e-mail; e.g. meeting summons and meeting reports) will be written in plain text using ISO-8859-1 encoding.

Usage of proprietary formats should be limited as much as possible. Although it is not possible to eliminate proprietary formats from the project work completely and some proprietary tools (e.g. Microsoft Excel) will help the group work more efficiently, we aim to minimize the usage of these tools and formats in order to minimize the group's binding to a particular platform.

Coding Standard

All source code written in the project will conform to Sun Microsystems' Java coding conventions. This ensures consistent source code formatting and indentation. The conventions can be found at [1W].

2.5.2 Templates

We plan to use templates extensively throughout the project. This has two primary advantages:

1. It ensures a consistent layout on all documents, making them easier to read and recognize.
2. It saves time.

We will now give a brief description of each type of template.

Phase Documents

This project uses the waterfall development model of software engineering. To avoid some of the problems with this model, it is important to have good documents describing each phase. To ensure a consistent layout on all phase documents, each phase document will have its own front page and will be written using the same LaTeX template and style. We have also created internal rules for capitalization and typesetting. The contents of each phase document are described in Section 2.3.3.

Meeting Summons and Reports

The templates for meeting summons and reports are respectively shown in Appendix A.3.1 and A.3.2. These documents are to be written in plain text files in order to be easily transmitted and read by e-mail.

It is important that these documents do not include much more than what is specified in the templates. If they get too complex and large, there is a possibility that they will be ignored or not read thoroughly.

Status Reports

Templates for status reports are described in Section 2.7.3. The actual template can be found in Appendix A.3.3.

2.6 Version Control

We will now describe how we will keep track of document and source code revisions in the project.

2.6.1 CVS

We will use the Concurrent Versions System (CVS) (see [2W]) for version control. A CVS repository has been set up in the group directory on the server `login.stud.ntnu.no` at `/home/groups/kpro7/cvs`. We will use Eclipse (see [3W]) for implementation. It has a built in CVS client, which will be used to keep track of both documents and source code revisions.

CVS Structure

The CVS tree has subdirectories (modules) `src` for source code and `doc` for other documents. Further levels of subdirectories are added as needed to organize and express structure.

2.6.2 Standards and Files

Version control will be used for all code files and for the important text files like phase documents. Small files and documents that will only be edited by one person and not included in the final report can be kept outside the CVS tree.

Directories

The directory name should reflect the contents and be on the same level of abstraction. A reasonable relation between nesting level and numbers of directories should be kept.

Files

All filenames not in CVS shall include the date it was last updated and the initials of the person who last edited it. It is also important that the title is meaningful and that the files sort properly in Windows and Linux. We will therefore name the files in a similar fashion as the following example:
`2005.08.30_innkallelse_intern_ED.txt`

2.7 Project Follow-Up

This section will define routines which will help us manage the project.

2.7.1 Project Meetings

Every Monday at 13:15 we will have an internal project meeting (mandatory). The session starts with everybody explaining what he has done since the last meeting. Then we will coordinate the completed work. Lastly we will delegate tasks and start working. We have also scheduled a weekly work session every Thursday at 12:00 (non-mandatory).

2.7.2 Internal Reporting

All group members must update their own MS Excel-file in the group directory. A summary of these files will part of the weekly status report (see Section 2.7.3) and used an incentive at the group meetings.

2.7.3 Status Reporting

No later than Wednesday 12:00 we will send the status report on email. A template showing the layout and basic content can be found in Appendix A.3.3.

2.7.4 Risk Management

We have chosen to list the risk factors in a table. We predict that this will grow, so we have put it in Appendix A.4.

2.8 Quality Assurance

We will now describe the quality assurance guidelines that will be used in the project.

2.8.1 Customer Response Times

We have agreed upon the following response times after discussion with the customer:

- Approval of meeting report from customer meetings: within the next workday.
- Feedback on phase documents: depending on size, but up to 3-4 days for a 50 pages document.
- Answering questions: within the current and the following workday.
- Provide requested documents: 24 hours.

2.8.2 Routines for Producing Quality

For each phase a group member will be given the responsibility of overseeing the phase and performing quality control.

2.8.3 Routines for Approval of Phase Documents

The document manager passes the final documents onto the supervisors for feedback. After correcting any issues the documents are approved by the group, except for the requirements specification which is approved by the customer.

2.8.4 Notice of Customer Meeting

Notices of customer meetings must be sent before 12:00 on Mondays for meetings on Wednesdays. The notice will include time, place, purpose, agenda, and requested preparations for the meeting.

2.8.5 Meeting Reports from Customer Meetings

The meeting reports are to be sent within 23:59 at the same day of the meeting. The customer must raise any objections within one workday.

2.8.6 Notice of Supervisor Meeting

The notice of supervisor meetings must be sent within 12:00 on the day before the meeting and will include time, place, purpose, agenda, and requested preparations for the meeting.

2.8.7 Meeting Reports from Supervisor Meetings

The meeting reports are to be sent within 12:00 at the following day of the meeting.

2.9 Test Plan

The test plan is in the test document in Chapter 7.

Chapter 3

Pilot Study

3.1 Introduction

This pilot study's primary goals are to gain a thorough understanding of the problem we are going to solve and to identify and evaluate strategies for solving the problem. We will do this by structuring the pilot study in the following way:

- The customer wants an SP modeling tool. We therefore give a short introduction to SP modeling to explain what the tool should do.
- Description of how SP modeling is done today and the software being used.
- Desired functionality of the solution.
- A market analysis to check whether or not there exists any tool we can buy, use, modify or write a plugin for.
- Possible strategies and evaluation of some tools of current interest.
- Conclusion.

Since the customer has already written a requirements specification draft, we will base some of our pilot study on these requirements. Although some of the requirements will be rewritten later, we will use these while performing a market analysis, to better see what problems may come and prepare ourselves for writing the final requirements specification.

3.2 Structure and Performance

The Structure and Performance (SP) modeling language is used to show the architecture of an information system and the links between different parts of the system. The information system can have any type of components, which makes it possible to use for both software and hardware.

A typical use case can be somebody designing a web shop, needing to know how many customers they can serve, or what hardware and software they need to serve a certain amount of customers. With SP, it is possible to see what data is computed at each level, and when combined with some analysis of queuing networks, one can compute average response time or the number of users being served per hour.

The SP modeling language consists of components connected with links. The components are typically memory, CPU, LAN and disk, or more high-level components like databases, servers, users and applications. These components keep information about what operations they do. See Figure 3.1 for a graphical example. The links between components are either memory links, processing links or communication links. To calculate performance, complexity specification matrices are made on these links, based on operation values from each of the two components they are connected to.

Another important aspect of SP is compactness. Compactness deals with how data grows or changes structure when being shuffled from one component to another, using the component specific operations.

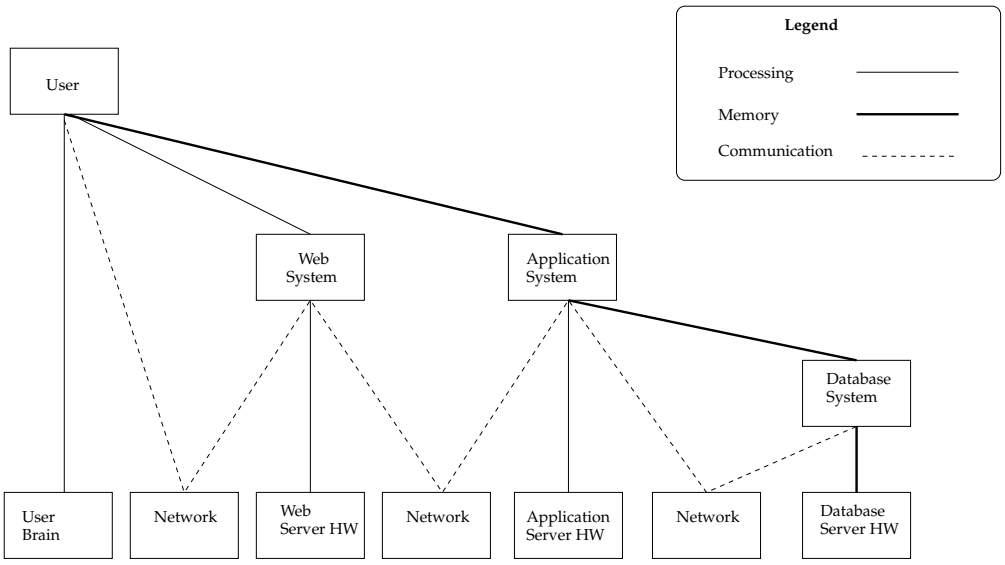


Figure 3.1: Example of an SP model

3.3 Current Solution

Today's solution is based on using MS Visio for the modeling and MS Excel for the calculations. These programs have been chosen because they are fairly flexible and easy to work with. They make it possible to get the job done in lack of a dedicated, more specialized tool. But the lack of a dedicated tool makes it difficult to get acceptance for the modeling language in the industry.

3.3.1 Problems with the Current Solution

While the tools are flexible, they do not impose any constraints or restrictions on how the modeling is done. Everything is possible within the limits of the programs themselves. This makes the learning curve rather steep as you are required to have a good understanding of SP and its rules in order to keep your model compliant and easy to understand.

Another problem is the lack of cohesion between the model and the data. The data is not connected to the components in the model, and you cannot click on them to display data from a specific component. The modeler must therefore manually keep the model and data consistent.

3.3.2 Alternative Solutions

There have been multiple attempts at implementing a stand alone SP modeling tool before. Unfortunately they were too difficult to use. The reason for this was strict constraints and demands of data input from the user at a very early stage. The prototypes were also made before the widespread of GUIs and would probably not have been useful today for generating a general interest in SP.

3.4 Desired Solution

The requirements of the customer are described in [1B]. In short, the customer wants us to implement a lightweight SP modeling tool, which can be used to easily create, load, save and export SP diagrams, and calculate performance and scalability parameters. We will now take a closer look at what the customer expects from the tool. The following is not and is not intended to be a formal requirements specification, but a description of key functionality the customer wants us to implement.

3.4.1 Integration of Modeling and Calculation

The customer wants to be able to create SP models *and* do performance and compactness calculations on the model within the same tool. This is important for at least three reasons, all of which are tied to ease of use. First, it will save time when the user does calculations on architectural models, since the user does not manually have to transfer information between two independent tools. Second, integrated tools will be able to provide SP-specific help and be able to check that the SP models are consistent. Third, customizing calculations and rules will be easier to do in a tool which understands the SP language and the context in which it is used. Therefore, an integrated tool is a requirement.

3.4.2 Few Limitations

Although integration is important, it is also important that the tool is not too rigorous in forcing the user to input a lot of information. An architectural modeler will often know very little about the problem domain and system when they are in the early phases of modeling. Therefore, the tool must allow for simple sketching and exploring without forcing the user to always create complete and correct models. Instead of forcing the user to create certain models, the tool should act as a *guide* to help create correct models, as described in [1B], Section 2.2.1.

3.4.3 Platform Independence

Java has been chosen to be the implementation language. Consequently, the tool will be cross-platform, which the customer wants it to be.

3.4.4 Extensibility

After the lightweight tool is completed, the customer wants to build a more extensive tool with more functionality, as described in [1B], Section 7. In building this more extensive tool, it would save the customer a lot of resources if they could use the lightweight tool, that we are going to implement, as a basis. In addition, since the development of the SP language is not finished yet, it is

possible that certain graphical symbols and rules will change during the next years. The customer therefore wants an extensible design and also a plug-in system which allows users to easily add new graphical symbols to the tool.

3.4.5 The User Interface

[1B] specifies how the graphical user interface of an SP tool should work. It divides the graphical user interface of the tool into four *views* in addition to a set of common tools available in each view.

The Project View

The project view is an overview or main view of the project the user is working on. The project view allows the user to tie several diagrams together into a single project and get an overview of the diagrams in the project. This makes it easier to manage large SP modeling projects.

The Architectural View

The architectural view is the part of the tool where the actual drawing takes place. It consists of an SP drawing tool, allowing the user to create SP diagrams with movable components and connectors, and a rule control system which can verify and check that the model is consistent. If the model is not consistent, the rule control system will let the user know what rules are being broken and offer help.

The Performance View

The performance view will typically be used after the user has created a model in the architectural view. The same model created in the architectural view will be visible in the performance view, but architectural operations are replaced by operations for performance modeling calculations. This includes specifying and viewing operations and complexity specifications of each component in the diagram and specifying queuing network matrices for the leaf nodes of the SP hierarchy. After feeding the necessary information to the tool, it can calculate performance parameters such as response times and service demand for the system.

The Compactness View

The compactness view is somewhat similar to the performance view in that the model created in the architectural view is also visible in the compactness view, but architectural operations are replaced with operations for registering data types, associating data types with the components and calculating compactness properties of each component.

The Tool Set Common for All Views

There will also be a tool set common for all the views. The tool set will provide functionality for opening and closing projects and diagrams, editing, printing and so on. From the common tool set the user can choose what view he or she wants to use. There will also be a comprehensive help system reachable from all the views.

3.5 Market Analysis

In this section we will discuss the current market situation and look at existing solutions that might help us solve the problem or parts of it. A brief introduction to *open source* software licensing models will also be given.

3.5.1 Current Market

As explained in Section 3.3 there are no current tools for SP modeling, and thus no competition. There is a demand for such a tool because it will make SP easier to work with and help promote the language.

3.5.2 Users

A modeling tool for SP will primarily be used by researchers and students of performance engineering. Because there are no other specialized tools available it is reasonable to expect that this user base wants the program. These users have lots of experience with computers and software, and this should be taken into account when the interface is designed and documentation written.

3.5.3 Open Source Licenses

Open source is a wide term that includes different licensing models and specific licenses. Different licenses may grant different degrees of freedom with respect to modification and redistribution of the code and derived works. In the most restrictive form, open source can allow you only to look at the source code, but not redistribute or modify it in any way. A license can also be less restrictive, allowing you to modify the code, and maybe even use it in commercial products without releasing the modified code. Source code could even be released into the public domain, with no copyright restrictions at all. A brief description of the most common licenses can be found in Appendix B.1.

3.5.4 Existing Partial Solutions and Frameworks

Here we take a look at some modeling programs and frameworks and decide whether or not they are of any use to this project. We are looking for modeling tools that can be modified directly or with plug-ins to fit SP. There are many existing tools and frameworks for modeling in UML, ER and other languages, and we will take a closer look at some of them.

Umbrello UML Modeller

Umbrello [7W] is an open source program for UML modeling. Umbrello is programmed in C++ and released under the GNU GPL. Umbrello uses XML, in

compliance with the XMI Standard, to save data. The program is quite specific for the UML language, with no support for plug-ins that might extend it to another language. It is difficult to tell if the code base would be useful for developing an SP modeling tool.

Mogwai

Mogwai [8W] is a program for ER modeling, but the project has also resulted in a framework for development of Java Swing applications. The GUI framework, and quite possibly some of the modeling code, might be useful for our project. Mogwai is released under the GPL.

DbModeller

DbModeller [9W] is a Java application for ER modeling with XML support. The graphical interface and XML support are features that might be reused. DbModeller is released under the GPL.

Cohesion

Cohesion [10W] is a modeling tool and framework in Java released under the BSD license. There is plug-in support to define new modeling languages, but there seems to be no XML support in the last release. XML support was promised for "the next major release", but this did not happen. The last release was in October 2000. The lack of activity in the project might be a problem, support of any kind should not be expected.

Dia

Dia [11W] is a diagram modeling program released under the GPL. It supports ER, UML, flowcharts, network and circuit diagrams. Definition of new shapes is supported using XML. XML is also used for saving diagrams. Dia also supports plug-in scripts written in Python.

3.5.5 What About Commercial Products?

There are many commercial modeling and simulation products, but without source code or a plug-in API it is not possible to extend such applications. Obtaining a license to the source code of a commercial product might be possible, but probably expensive.

3.6 Evaluation Criteria

To help us choose from the different possible strategies, we have elicited some absolute requirements from the customer:

- **Language**
  The program must be written in Java. It is not necessary to be backwards compatible, using only the newest Java-version is alright.
- **Operating system**
  The program will mainly be run on Microsoft Windows XP, but it should also run on Linux (2.6 kernel).
- **Hardware**
  The program must run smoothly on a normal[1] x86 compatible personal computer (PC).
- **Storing files**
  Projects and diagrams must be stored i XML-format. (See F25 p.67 in [1B].)
- **Integrated software**
  All functionality must be integrated into *one* program. I.e. to use one tool to model and one tool to calculate, is not an acceptable solution.
- **Use existing software**
  The customer wants 100% control over the SP core functionality, thus everything defining SP must be protected. The customer also requests that the chosen solution does not in any way restrict further development.

We have chosen to use this short list of absolute requirements instead of weighing the requirements. This simplification is justifiable if the list narrows the alternatives down to 1-3 alternatives. This seems very likely, since the customer knows the market and its available tool sets very well.

[1] By normal, we mean a computer with specifications similar to the desktop computers available for purchase from 2000 and onwards.

3.7 Alternative Strategies

The customer wants to solve a problem, and our job is to find the best solution to the problem, given the time and resources we have available. A short explanation of the alternative approaches will be given. For in-depth coverage we recommend [2B].

3.7.1 Buy an Existing Solution

If there already exists software which solves the problem, this is often the best and cheapest solution. Issues to consider when looking at existing products is not only how much it costs to buy, but also how much it will cost to configure it and how much it costs to maintain. Second we have to consider if it can be expanded and scaled in accordance with the customer's needs.

3.7.2 Write a Plug-in to Existing Software

Many general purpose programs support plug-ins to expand the functionality. This can prove beneficial if the customer already uses the software in question or can in other ways take advantage of the main program. If neither of these are true, the risks may outweigh the advantages. It is a cumbersome process to buy a program, develop a plug-in for it and lastly configure them both to the domain in question. The process introduces a lot of unknown variables, so we recommend this strategy only in very special circumstances.

3.7.3 Start with an Existing Open Source Project

Open Source code is a concept which might be a feasible strategy, if the customer is willing to distribute the modifications and in other ways live up to the requirements of the license.

3.7.4 Develop from Scratch

The last alternative is also the most flexible: Develop the software from scratch ourselves. This is a costly solution, but it may still be cheaper than the others, especially if they require a lot of tailoring as well. Software contracted by the customer, is also the property of the customer. This business advantage is often much valued.

3.8 Evaluation

Here we will discuss which of the options that are viable.

3.8.1 Buy an Existing Solution

We did not find an existing solution or a program that could be easily customized to solve the problem according to the customer's wishes. This is not unexpected since the customer should in this case be fully aware of any existing solution.

3.8.2 Write a Plug-in to an Existing Program

We did not find a program suitable for this. The current solution is based on MS Visio and MS Excel, and neither of those tools are suitable to extend so that it would fit the requirements of the customer.

3.8.3 Start with an Existing Open Source Project

We have found that the main obstacle with this strategy is the license. The customer wants to have full control of the SP core as they want to be cautious due to bad experiences in the past. If the source for the entire product has to be distributed as in accordance with a GPL license, then others might make slight changes to the SP language and we will end up with multiple derivations. This is something that the customer wants to avoid at all costs.

Another requirement that conflicts with some of the evaluated programs, is the development language. The customer wants the solution to be implemented in Java as this is a language well known inside the academic community. This will make it easier to extend the program later.

Still, we found one possible non-GPL candidate. Cohesion is a BSD licensed UML tool with some support for defining new modeling languages. Unfortunately the project is rather complex and is lacking in some areas, like XML support. And with no active development since 2000, we are reluctant to have to deal with something where we will have to first get a good understanding of the code before we will be able to modify it.

3.8.4 Develop from Scratch

This appears to be the most viable solution in our case. It fits the requirements of the customer and should be possible within our time frame. We can still use some of the tools found under the other options as inspiration for how a good modeling tool should look and feel.

3.8.5 Conclusion

After excluding solutions according to the evaluation criteria, we are left with two options: develop from scratch or continue developing on the existing open source project Cohesion. The Cohesion option faces two prominent problems: One, the lack of XML support. And two, that no development has taken place the last five years. The work required to gain a thorough understanding, and *then* start to extend it, is therefore deemed too great. This does not mean, however, that we do not want to take advantage of Cohesion. We hope to make use of concepts regarding the GUI, and if we are very lucky we can reuse some code as well. *The chosen strategy is to develop from scratch.*

The chosen solution has been approved by the customer. This concludes the pilot study.

Chapter 4

Requirements Specification

4.1 Introduction

This document is the requirements specification for the product *SP Light* by group 7 in the course TDT4290 Customer Driven Project. This document aims to follow and comply with IEEE Standard 830–1998: IEEE recommended practice for software requirements specifications (see [3B]) closely. However, not all parts of this standard apply to this project, so we have omitted or adjusted certain parts of the standard where appropriate.

We will begin by describing the purpose and scope of this document, definitions and terms used, relevant references and an overview of the rest of the document.

4.1.1 Purpose

There are two primary purposes and goals for this document. First, this document establishes an agreement between us and the customer about what the software product will do. Second, this document is to define all significant requirements for the system, which in turn will be used to create a design for and implementation of the system in addition to forming a basis for quality assurance and verification of the final system. This requirements specification is, due to these purposes, primarily written for the customer and the supplier of the software product.

4.1.2 Scope

This document will only concern itself with the product *SP Light*. The product is a Structure and Performance (SP) modeling tool. It will be used for the following purposes:

1. Create and perform calculations on SP models.
2. Build a more extensive SP modeling tool in the future.

3. Demonstrate of the possibilities of the SP modeling language.
4. Be a basis for evaluation of the SP modeling language. (The SP modeling language is still under development).

This document will not describe any design or implementation details. The goal is to describe *what* the product will do, not *how* the product will be made.

4.1.3 Definitions and Abbreviations

We will now define some terms used in this document.

Customer. This is used to denote the Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway and Modicum, Ltd., Cheshire, United Kingdom.

Supplier. This is used to denote group 7 in the course TDT4290 Customer Driven Project in the fall, 2005 semester. Note that the supplier has written this document. Therefore, we will use "we" throughout the document to denote the supplier.

Shall. This will be used to denote that a requirement is absolutely required in the product.

Furthermore, we will use the abbreviations in Appendix 8.5.2 throughout the document.

4.1.4 References

The references used in this document can be found in the References section.

4.1.5 Overview

The rest of this document is organized into two sections.

First, we will in Section 4.2 give an overall description of the system. We will describe the product perspective (how the product interacts with other products), product functions (a summary of the major functions of the product), user characteristics and intended users of the product, constraints that will limit the implementors' options, assumptions that may affect the requirements specified in this document, and also identify requirements that will possibly be delayed until future versions of product.

Second, we will in Section 4.3 describe the specific requirements of the system. We will describe non-functional and functional requirements. The functional requirements are in turn divided into functionality requirements and interface requirements.

4.2 Overall Description

This section is intended as a background for the specific requirements in Section 4.3. We will now describe general factors affecting the product and its requirements.

4.2.1 Product Perspective

SP Light will be a stand-alone, single user application, with one exception: The results obtained through the calculations shall be available in a format compatible with MS Excel. This is an absolute requirement for results to be used in queuing network analysis, since this functionality is postponed to a later version of SP Light. See 4.2.6 for further details. The help functions of SP Light depend on an Internet connection and a working web browser. In all other aspects, the product is completely self-contained.

4.2.2 Product Functions

We will now describe the major functions of the system. The product that will be created will be a Structure and Performance (SP) [1] modeling tool. The major functions of the tool, which will be further specified in Section 4.3 of this document, are listed below:

1. *Creating and checking SP diagrams*. The user shall be able to load, save, draw, view and edit SP diagrams, using a graphical user interface. Standard user interface features, such as cut, copy, paste, undo and redo shall be included in the user interface. The tool shall also be able to check if a model created by the user follows the rules of the SP modeling language, and if not, report to the user the problems with his or her model.
2. *Performance calculation*. The user shall be able to add operations to the components of SP diagrams created with the tool, and specify performance parameters for each component. This is specified in a matrix called a *complexity specification matrix* (CSM). After specifying performance parameters for each component, the tool will be able to do performance calculations (such as service demand and response time) on the components in the diagram.
3. *Compactness calculation*. The user shall be able to specify compactness functions[2] for each component by describing *implemented data type matrices* (IDT matrices), which are to compactness calculations what CSMs are for performance calculations. The tool will subsequently be able to do compactness calculations on the components in the diagram.

[1] SP is a modeling language used for modeling and calculating performance aspects of computer systems. For further information on SP, we refer to [1B].

[2] Compactness functions are used to describe how much a data type grows when propagated through the components of a system. This is important in order to know, for instance, how much bandwidth and memory is required in the system.

4. *Modeling projects*. The user shall be able to easily organize several diagrams into one large modeling project.
5. *Help*. There shall be a help system where the user can get information about the SP language and how to perform tasks in the tool.

4.2.3 User Characteristics

The users of SP Light will mostly be academia. We will consider two main types of users, normal users and SP language researchers. A typical user might be a student of performance engineering and is assumed to have the knowledge of a typical fourth year student of computer science and software engineering. It shall be possible for the researchers that develop SP to update SP Light to reflect future changes in the rules of the SP language. Both user types are assumed to have a high technical knowledge level.

4.2.4 Constraints

Interfaces to Other Applications

It shall be possible to export data to a format supported by MS Excel.

High-order Language Requirements

The program shall be implemented in Java.

Reliability of the Application

There are no reliability requirements beyond what can be expected from a standard single-user application.

4.2.5 Assumptions and Dependencies

As previously mentioned, the program will be implemented in Java. It shall run under the Java 2 Platform Standard Edition (J2SE) Runtime Environment 5.0 [15W].

4.2.6 Apportioning of Requirements

Given the project time frame, some functionality wanted by the customer must be delayed to a later version. This concerns the queuing network analysis and the help wizard. Some other help functions may also be given a low priority, see 4.3.2 for details.

4.3 Specific Requirements

We will now describe the specific requirements of the system. For further information on the background of the requirements, we refer to [1B].

4.3.1 Non-Functional Requirements

We will now list the non-functional requirements of the system. These are requirements that are not directly related to what the system will *do*, but *how* and in which environment it operates. We have divided the non-functional requirements into several categories, and will now describe the requirements in each category. Note that all of the following requirements are highly important and the tool shall satisfy all of them. Consequently, no priorities have been assigned to these requirements.

User Interface and Human Factors

NF–1 Users with computing experience similar to fourth-year computer science students at NTNU shall be able to use and learn the tool. No previous experience with SP shall be required. After trying the tool for 15 minutes, a user with this competence shall be able to draw diagrams and perform simple calculations.

NF–2 The tool shall as far as possible follow the design guidelines described in [17W].

Documentation

NF–3 The purpose of a class must be documented in the beginning of the file.

NF–4 Every non-trivial method shall be documented. A method is non-trivial if it contains more than five statements or more than one conditional test.

NF–5 Javadoc, as described in [14W], shall be used.

NF–6 The installation howto and quick-start guide shall be included in every release.

NF–7 The customer shall be able to update and add to the user documentation without knowledge about the software implementation.

Hardware and Software Environment

NF–8 The tool shall run on x86 personal computers sold from the year 2002 and onwards. It shall require no more than 650MiB of disk space.

NF–9 The tool shall run on MS Windows XP and Linux with major kernel version 2.6.

NF–10 The tool shall be written in Java version 1.5 and shall be able to run using the J2SE Runtime Environment 5.0, which is obtainable from [15W].

NF–11 If an Internet connection and a working web browser are ready for use, the tool's help functionality shall be available.

Performance

NF–12 The tool shall have reasonable response time. By reasonable response time, we mean the following: At least 50% of a group of at least 6 students, fourth year or above, at NTNU will characterize the system response time as "very good" or "excellent" after using the program for at least 15 minutes.

Reliability

NF–13 The tool shall have an auto-save feature which stores a backup of the files currently open in the tool every 10 minutes.

4.3.2 Functional Requirements

We will now describe the functionality requirements of the system. These are requirements for what should be possible to do with the tool.

We will use the following priority levels:

H This denotes a requirement of crucial and very high importance. It is absolutely necessary that the product satisfies this requirement. If the product does not satisfy these requirements, it will severely degrade the functionality of the product.

M This denotes a requirement of medium importance. These requirements are not absolutely required in order for the product to be usable, but they represent functionality that is generally helpful or in other ways desirable.

L This denotes a requirement of low importance. These requirements are not required in order for the product to be usable, but they represent functionality that in certain situations can be helpful. However, the functionality is not considered to be generally as important as the functionality described by the requirements with a medium priority.

External Interfaces

H F–1 Output from calculations shall be presented in a tab-separated list readable by MS Excel.

H F–2 Output from calculations shall include the results from each level of the model.

H F–3 The tool shall be able to start the local computer's default web browser.

Storage

H F–4 Every project shall have its own file containing diagram and library data.

H F–5 All data shall be stored in XML format.

H F–6 It shall be possible to store different versions of a diagram.

Standard User Interface Features

H F–7 It shall be possible to create, open and save projects.

H F–8 It shall be possible to import a diagram into an existing project.

H F–9 The system shall provide cut, copy and paste using the system's clipboard functionality.

H F–10 It shall be possible to print diagrams.

M F–11 It shall be possible to undo (and redo) at least 20 user operations.

M F–12 It shall be possible to zoom in to and out of the diagrams in even steps.

L F–13 Keyboard shortcuts shall be provided for all menu items.

L F–14 Every diagram shall have a unique identifier.

L F–15 It shall be possible to set the zoom at a given percentage.

L F–16 When printing, diagrams shall fit on A4 or A3 sheets.

Help Functions

H F–17 User documentation shall be accessible from the help menu.

H F–18 User documentation shall be available on a web page.

H F–19 The help menu shall provide an about box.

H F–20 The about box shall include the name of the program, version of the program, names of developers, link to home page, license agreement and copyright information.

H F–21 Right-clicking on an element shall provide a link to further information online.

H F–22 An installation manual shall come with every release of the program.

M F–23 If accessing the user documentation fails, the error message shall explain why and give the user an URL to the target.

M F–24 Tool tips shall be provided whenever icons are used without textual representations.

User Documentation

H F–25 Expanding the documentation shall be limited to developer users.

M F–26 Expanding the documentation shall at most require one modification of the source code.

M F–27 User documentation shall be divided into cross-referenced topics.

M F–28 User documentation shall be searchable.

L F–29 User documentation shall include information on the SP language.

L F–30 User documentation shall include information on how to use SP Light.

L F–31 User documentation shall include tips about how to create good SP diagrams.

Rule Checking

H F–32 It shall be possible for developers to create new rules.

H F–33 It shall be possible for developers to change rules.

H F–34 It shall be possible to select which rules to use when checking.

H F–35 After the rule checking is complete, a list of all violations of the chosen rules shall be displayed.

H F–36 It shall be possible to minimize the error log window, while making corrections to the diagram.

M F–37 The error log shall contain date and time, name of diagram and references to further information on the potential problems.

M F–38 The checking shall be initiated with a button.

M F–39 If a rule check fails for a connector, it shall be colored red.

M F–40 It shall be possible to enable or disable color coding.

Function Parser

H F–41 Functions shall use infix ordering.

H F–42 The parser shall validate syntax.

H F–43 The parser shall support addition, subtraction, multiplication and division.

H F–44 The parser shall support exponentiation and parentheses.

L F–45 The parser shall support the guard operator.

L F–46 Developer users shall be able to expand the parser with new operators.

Components and Connectors

H F–47 A connector shall store CSMs and IDT matrices and thereby represent the relation between two components.

H F–48 It shall be possible to give a component a name.

H F–49 It shall be possible to add services to a component.

H F–50 It shall be possible to give a service a name.

H F–51 It shall be possible to add logical storages to a component.

H F–52 It shall be possible to give a logical storage a name.

H F–53 It shall be possible to assign data capacity/limit to a component.

H F–54 It shall be possible to use the sub-diagram component to denote a sub-system.

H F–55 It shall be possible to use a modified sub-diagram component as a means of abstraction.

H F–56 When connecting a sub-diagram with another component, it shall be possible to choose which component(s) in the sub-diagram the connector shall connect to. It shall also be possible to choose which component(s) in a diagram an outgoing connector in a sub-diagram shall connect to, while viewing the sub-diagram.

M F–57 There shall not be any limit on how many components and connectors a sub-diagram can contain.

L F–58 It shall be possible to change from single component type to multi component type and vice versa.

Matrices

H F–59 Rows in the IDT matrix shall contain the IDTs of the hierarchically uppermost component of the link.

H F–60 Columns shall contain the IDTs of the hierarchically lowermost component of the link.

H F–61 It shall be possible to specify a compactness function for each memory connector.

H F–62 The compactness function represents a mapping between the logical elements in the upper-level and the lower-level of the connector, and shall be specified using a matrix.

H F–63 When created, a CSM shall be filled with empty cells with no name, no description and content evaluating to zero.

H F–64 Cells in CSMs shall have a value specified by an expression recognized by the function parser described in Section 4.3.2.

H F–65 It shall be possible to add functions and constants to SP Light's library.

H F–66 Each project shall have its own library.

H F–67 A project's specific library shall extend SP Light's built-in functions and constants.

H F–68 It shall be possible to specify work load and data load on the top level.

M F–69 It shall be possible to import functions and constants from the project's library.

M F–70 It shall be possible to export functions and constants to the project's library.

M F–71 Cells in CSMs or IDT matrices shall be addressable.

M F–72 If a cell has been given a name, it shall be possible to refer to it in other matrices in the project.

M F–73 Cells in CSMs or IDT matrices shall be able to have a textual description of the cell content. The user shall be able to choose whether or not to provide a description of the cell.

General GUI Requirements

H F–74 Icons shall look and feel as in MS Windows and Office.

H F–75 Every process expected to take more than ten seconds shall have a progress bar.

M F–76 Every process expected to take more than one second should cause the mouse cursor to turn into an hourglass.

M F–77 All modal dialog windows shall have a cancel button, unless no input is expected. (I.e. it only displays a message.)

Main Window Requirements

H F–78 The main window shall have a drawing area, a drawing tool set, and a tree structure showing all diagrams in a project, all sub diagrams, components and operations.

H F–79 The top level menu (level 1) shall include the items File, Edit and Help.

H F–80 At menu level 2 the functionality in 4.3.2 shall be placed.

H F–81 Double-clicking a diagram in the tree structure shall open the diagram in a new tab, or focus on an existing tab if the diagram is already opened.

M F–82 The drawing area shall be tabbed and it shall be possible to show different diagrams in different tabs.

M F–83 There shall be a button in the main window that opens the window for rule checking.

Drawing Requirements

H F–84 The tool shall be able to draw three different components: a normal rectangle, a rectangle surrounded by a dashed line and a rectangle with a shadow.

H F–85 The tool shall provide three different connectors: a dashed line, a thin line and a thick line.

H F–86 The component box shall not be resizable.

H F–87 The drawing area shall have scrollbars.

H F–88 All the components and connectors shall be movable.

H F–89 When moving a component, connectors shall remain attached.

H F–90 When detached from a component, a connector shall retain all data.

H F–91 A connector going in or out of a sub-diagram shall be named.

M F–92 It shall be possible to perform drag-and-drop of the symbols from the sidebar menu.

M F–93 The size of each component box shall be preset to a width of 15 columns and a eight of 3 rows of characters.

M F–94 The drawing area shall have a grid.

M F–95 Each component shall have a number on the top right, indicating how many operations it has.

M F–96 It shall be possible to detach a connector from a connection point and move it to an arbitrary component.

M F–97 When reattaching a nonempty connector, the tool shall check that the attachment is valid.

M F–98 When a connector is moved to a component with different properties, the user shall be able to choose “abort” or “continue as new connector”.

L F–99 It shall be possible to draw a connector by drawing connected straight line segments (a “polyline”) rather than a single straight line.

L F–100 Each component box shall have five connection points for links on top of it, and five more under it.

L F–101 Components shall have the option to be drawn as a wide, rectangular box, to model i.e. a bus.

L F–102 The font used shall be Arial (or another tidy and commonly used text font) in a readable size for printing.

Component Window Requirements

H F–103 The component window shall have two tabs: one for operations and one for data structures.

H F–104 It shall be possible to open a sub-diagram by double-clicking it.

H F–105 The operations window shall contain a list of all operations the component has.

H F–106 It shall be possible to add and remove operations and data structures from a component.

H F–107 For multi components on the hardware level, it shall be possible to register how many components are actually contained in them.

H F–108 It shall be possible to register a rate on components, when a rate is applicable.

H F–109 It shall be possible to register a size on components, when a size is applicable.

M F–110 When double-clicking a component, either in the diagram or in the tree structure, the component window shall open.

M F–111 The tool shall be able to calculate service demand from the properties entered about the hardware components.

Connector Window Requirements

H F–112 The component window shall have two tabs: one for complexity and one for compactness.

H F–113 The connector window shall have a matrix consisting of rows being operations from the upper component connected, and columns being operations from the lower component being connected.

H F–114 In the matrix cells, it shall be possible to enter constants or functions, with possible external references (i.e. references to functions in the function library or references to elements in some CSM or IDT matrix.)

H F–115 When operations are added or deleted from a component, the size of all affected matrices shall be updated.

4.4 Summary

This requirements specification contains the requirements elicitated from the customer. They are sorted by function and have been prioritized.

We hope to implement all requirements with a priority level of **H**, but we realize that this might be overly optimistic. Therefore, if time runs out, the rule checking and function library functions will be left out. These are functions of major importance in SP Light — so even though we may not have the time to implement the actual functions, we aim to implement a structure for implementing these features so that adding them in a later version of SP Light will be a relatively easy task.

Chapter 5

Design

5.1 Introduction

The purpose of this document is to create a description of how we will create the system described in the requirements specification. We will do this through both textual descriptions and UML models.

First, we will first look at the overall system design in Section 5.2, followed by important design decisions in Sections 5.3. Then, we will look at how the components of the overall design are designed in Section 5.4. Finally, we will summarize our findings in Section 5.5.

For UML modeling, we have chosen to use Poseidon[21W]. There are two primary reasons for choosing Poseidon:

1. Poseidon can generate Java source code from UML diagrams. Thus, after we have created the UML diagrams, creating a code skeleton in Java requires no effort and saves time.
2. Poseidon supports reverse engineering: It can read Java code and output UML diagrams. This is a major help to us, because *when* (not *if*) the design changes during the implementation, we do not have to update the UML diagrams manually. This saves time and guarantees that there is always consistency between the UML diagrams and the source code, which would likely not be the case if we had to do the updating manually.

Thus, using Poseidon saves time. Certain non-UML diagrams are modeled using MS Visio [22W].

5.2 General system architecture

A UML package diagram, showing the overall subdivision of the system, is shown below in Figure 5.1.

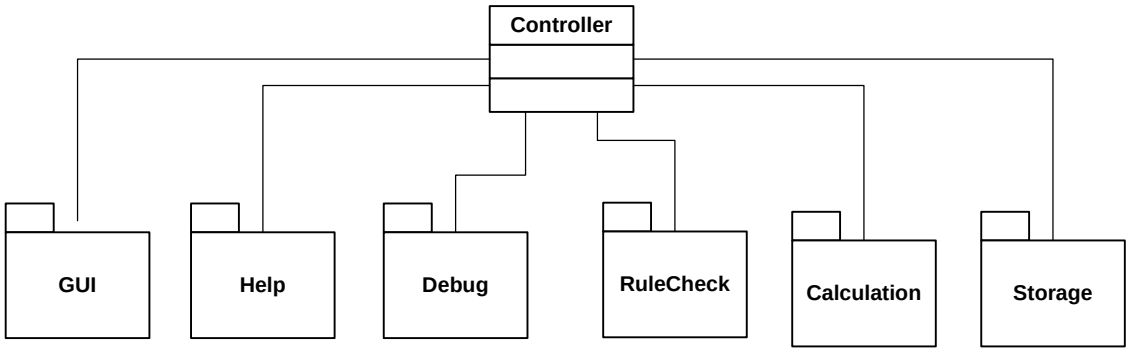


Figure 5.1: Package diagram

We will now take a look at the responsibility of each package. In Section 5.4, we will further describe the packages.

5.2.1 Controller

The Controller class is responsible for linking the GUI with the system logic. On user interaction, the controller will receive calls from the GUI part and call the appropriate methods in the other packages. The GUI will be using listeners in order to be notified when changes in the underlying data models occur (primarily due to user interaction). This is the commonly used MVC pattern; see for instance [20W] for further details.

5.2.2 GUI

The GUI package is responsible for displaying the GUI of the system and receive and respond to input from the user. The GUI will communicate with the Controller to perform the tasks the user wants to do.

5.2.3 Help

The Help package is responsible for providing help to the user. As mentioned in the requirements specification, documentation will be provided via a Wiki

available on the web. The Help package of the program is responsible for providing links to and opening the Wiki (via a Web browser on the user's machine) when the user requests help — for instance by accessing the "Help" menu from the program or right-clicking some item in the program and selecting "Help."

5.2.4 Debug

The Debug package will contain various debugging functions, used when developing the program. For instance, it will contain routines to print out certain data structures of the program to the screen so that they can be inspected and checked for errors.

5.2.5 RuleCheck

The RuleCheck package will check if a SP diagram follows the rules of the SP language. This involves, for instance, traversing the model the user has created and checking that it is correct according to the rules in the SP language.

5.2.6 Calculations

Calculations are of great importance in SP models. This package will take care of all calculations to be done on SP models. It will also parse and evaluate mathematical expressions the user has typed in. Furthermore, this package contains functionality for storing user-specified functions in a project.

5.2.7 Storage

The Storage package contains functionality for reading and writing files from/to a (local or remote) file system. Primarily, this package will read XML files and create the data model in the program which represents the information in the XML file, and write the current data model out to an XML file. The package will also contain functionality for exporting the information in the SP diagrams to a tab-separated format (readable by MS Excel).

5.3 Important Design Decisions

We will now describe design decisions which have a profound impact on the design of the system.

5.3.1 Storage

The use of XML for storage has been a requirement from the beginning. To implement this we have decided to use XMLBeans because it provides the mapping to/from Java objects. A thorough description of how we plan to use XMLBeans can be found in section 5.4.4.

5.3.2 Import/Export Through SP Light

Early on it was desirable to move diagrams between different projects outside of the program, e.g. moving files from one project folder to another using MS Windows Exporer. Each diagram would be stored as its own file, but this has been changed into storing a complete project as a single file. This was to make the use of XMLBeans easier. Due to this we have added a new requirement for importing/exporting diagrams inside of SP Light.

5.3.3 Retain or Not Retain Data?

Connectors are used to represent the relationship between two components using matrices. The matrices depend on the services and logical storage units which might differ from component to component. This is the reason we have decided to clear the matrices whenever a connector is detached from one component and attached to a new one. Also, to simplify it further, we have decided to require connectors to always be connected to two components except during move operations.

5.4 Detailed Design

We will now describe the design in greater detail. We will in the following sections describe the design of the GUI, the calculations subsystem, the rule checking subsystem, and the storage subsystem.

5.4.1 GUI Design

An important feature of the main window is the drawing panel used for constructing the models. It is responsible for rendering a visual representation of the model and allowing the user to add, remove, and move the various modeling objects around.

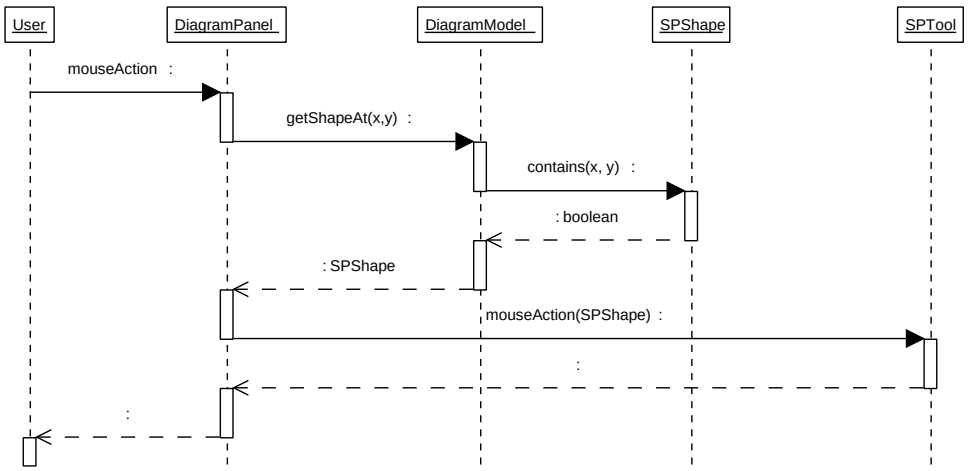


Figure 5.2: GUI mouse action sequence diagram

The sequence happening when clicking in the drawing panel is shown in Figure 5.2. SPShape is an abstract superclass for the objects used to represent the visual elements in the drawing panel. SPTool is the superclass for the various toolbox functions like moving elements in the diagram, adding a new component, and adding a new connector.

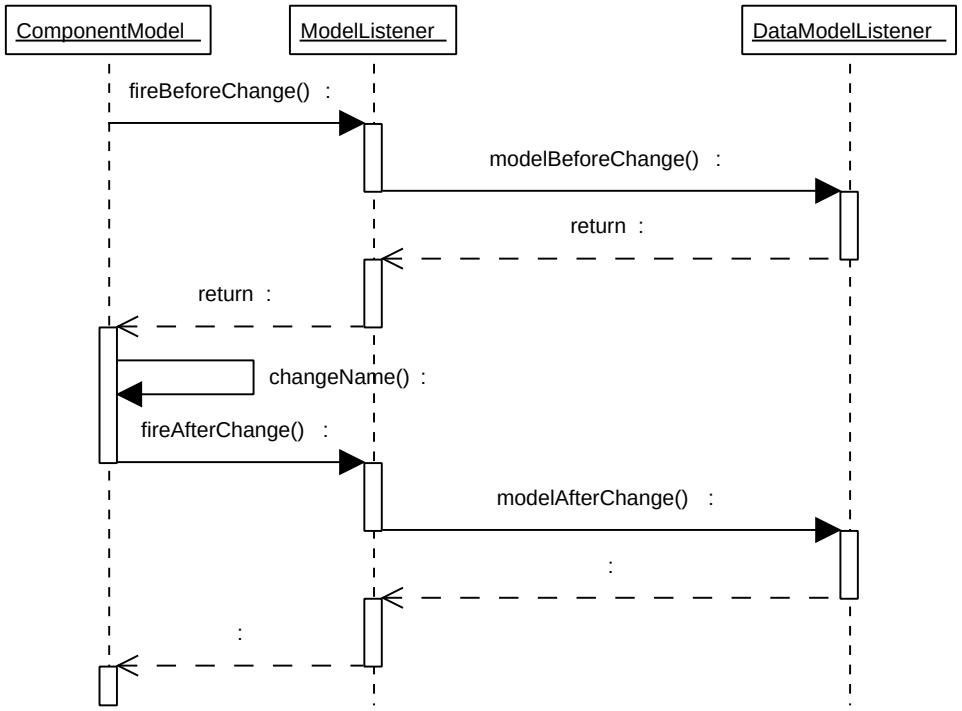


Figure 5.3: ComponentModel event sequence diagram

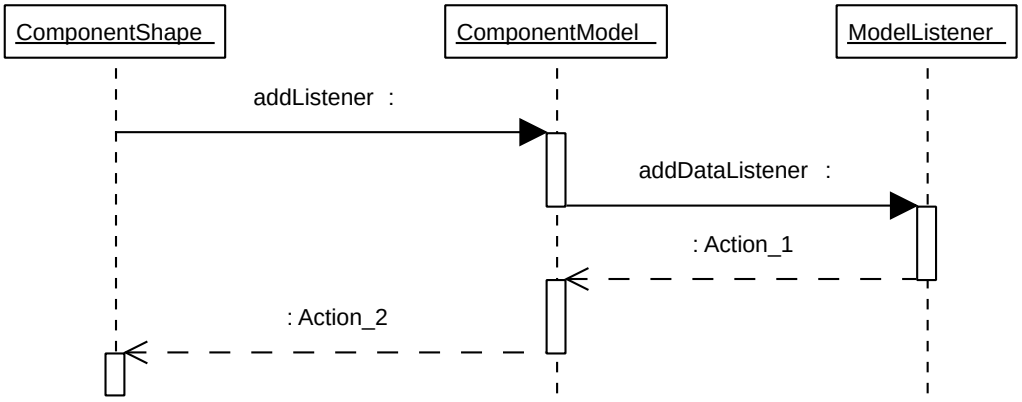


Figure 5.4: ComponentModel addListener sequence diagram

As mentioned earlier, we are using the MVC pattern. Figure 5.2 shows a sequence diagram for the change notification functionality provided by the data model objects. Before any change takes place to the object itself, a modelBeforeChange event is sent, followed by the change, and a modelAfterChange event. The modelBeforeChange event is intended used with the undo/redo functionality. Figure 5.4 shows a sequence diagram for when a ComponentShape adds a DataModelListener listener.

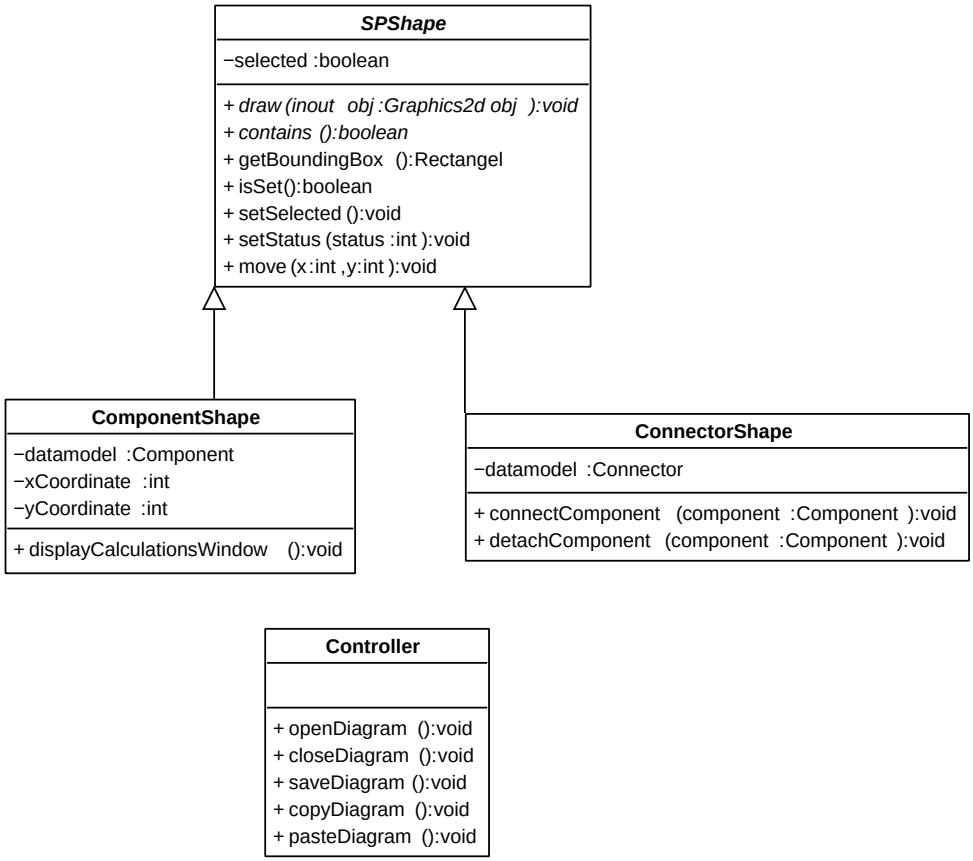


Figure 5.5: GUI drawing class diagram

The drawing panel uses descendants of the SPShape superclass to represent the various modeling elements in the diagram. The class diagram for the SP-Shape and its subclasses is shown in Figure 5.5

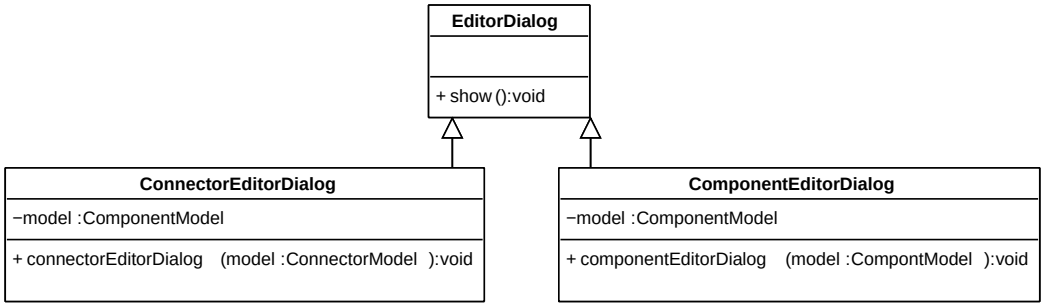


Figure 5.6: GUI dialogs class diagram

A class diagram for some of the dialogs is shown in Figure 5.6. The main point here is that the dialogs take a data model object in the constructor, but besides that they are fairly self-contained.

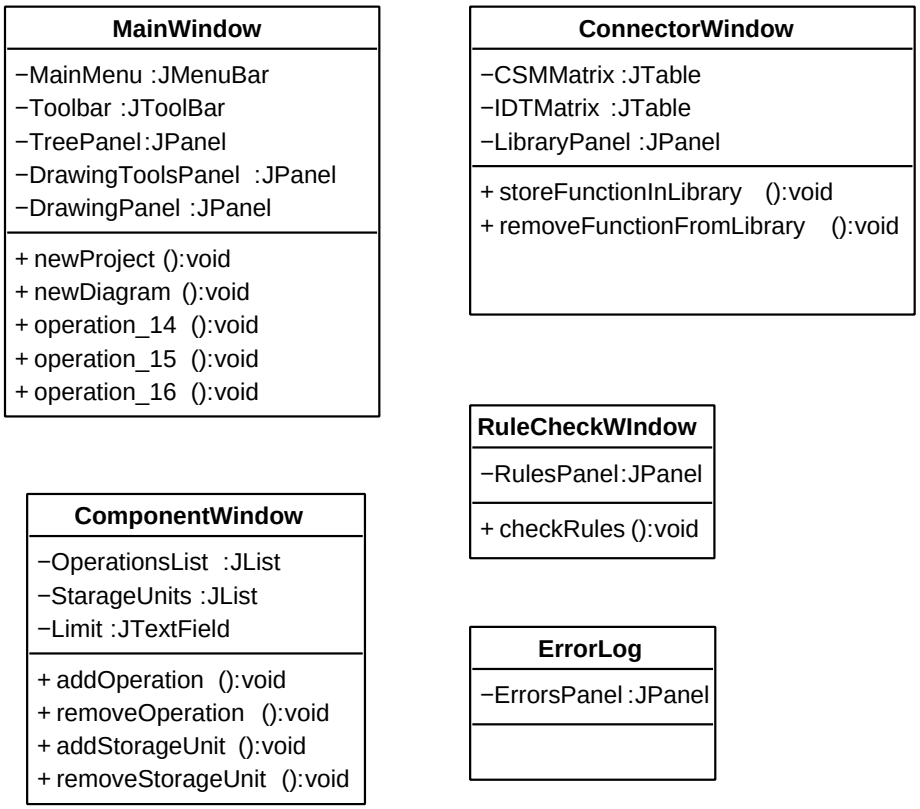


Figure 5.7: GUI windows class diagram

5.4.2 Rule Checking

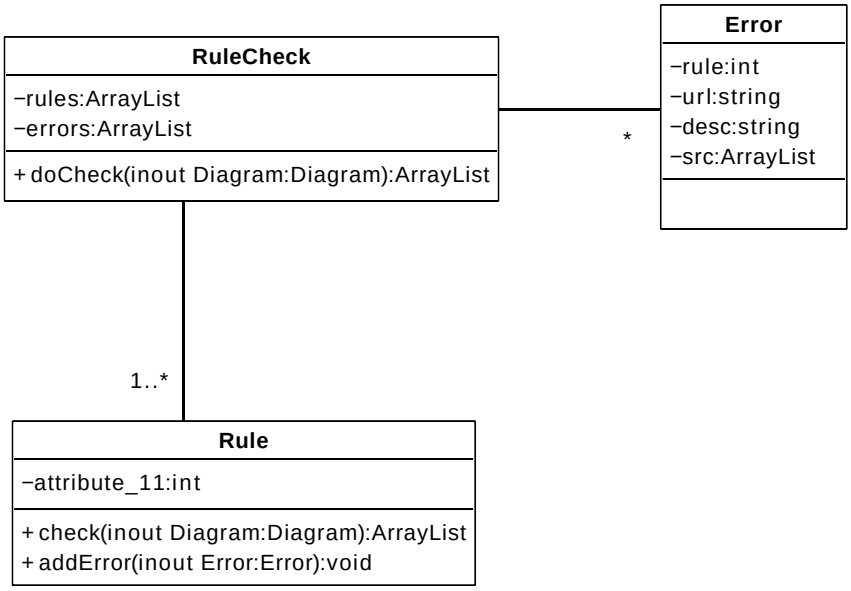


Figure 5.8: Rule checking class diagram

The rule checking is handled by a RuleCheck object that has a list of one or more instances of descendants of the Rule class. For each new rule a subclass of Rule has to be implemented.

5.4.3 Calculations

The UML class diagram in Figure 5.9 shows the design of the Calculations package.

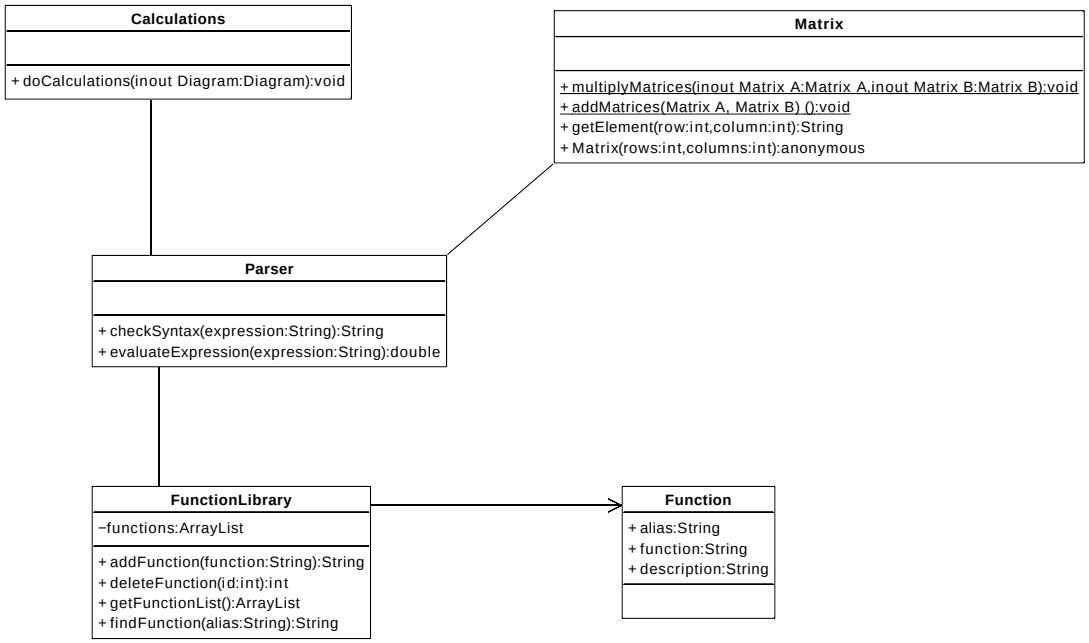


Figure 5.9: Calculation class diagram

The Calculations class is responsible for performing the actual calculations. A crucial part of performing the calculations is parsing mathematical expressions, which is the responsibility of the Parser class. In addition to being able to evaluate mathematical expressions, this class is also able to check the syntax of expressions the user types in.

This package also contains the function library, which stores commonly used mathematical expressions, together with an alias and description. When parsing an expression, the expression may contain references to external functions which the user has defined in the function library. The parser will therefore interact with the function library.

A typical call sequence is shown in Figure 5.10. Note that the Calculations object will typically call evaluateExpression in the parser many times during one doCalculations call, but in order to keep the diagram simple, we have not shown this explicitly.

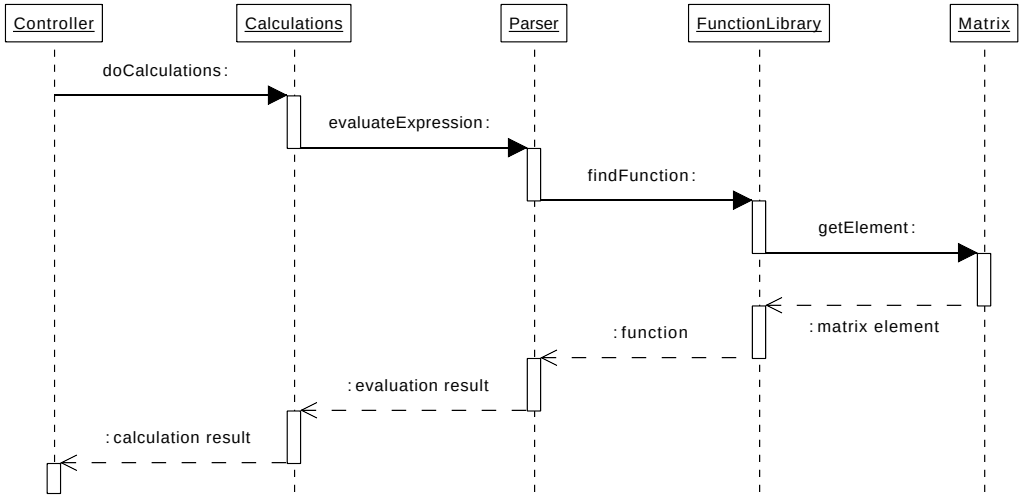


Figure 5.10: Typical calculation sequence

5.4.4 Storage Design

We will now explain how persistent storage shall be solved. Neither databases nor distributed networking will be used by SP Light. The customer wanted an XML-based storage solution, so that possible future applications can make use of projects made by SP Light. We have therefore chosen to base the storage on XML Schema ([18W]) and XMLBeans ([19W]).

Original Datamodels

We started with making the data model shown in Figure 5.11.

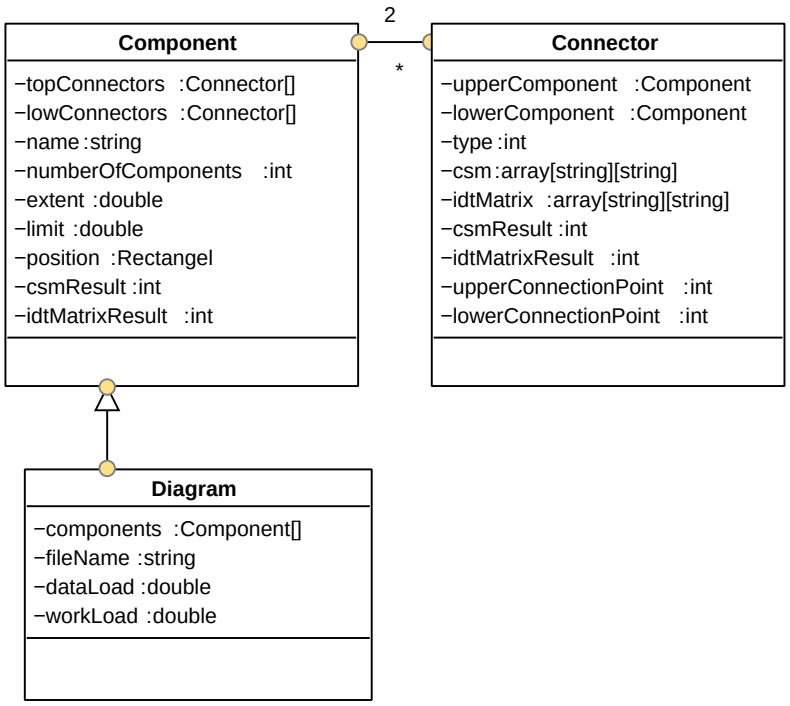


Figure 5.11: Original datamodels

We used these classes [1] as a basis for creating an XML Schema, which can be found in Appendix C.1. Each project file will be validated against this schema and in the end produce one XML-file. All diagrams, the project name, and library functions will be stored here.

To allow for easy save and load functionality we have chosen to use XML-Beans. XMLBeans takes the schema as input and outputs a JAR-file which includes the data models and corresponding methods. XMLBeans also handles validation against the schema.

Data Model Wrapper Classes

We have chosen to implement wrapper-classes to encapsulate the XMLBeans classes. The XMLBeans classes will be stored in a separate .JAR-file generated by XMLBeans. The wrapper-classes, as shown in Figure 5.12, also hold a list of listeners and other functionality needed by the rest of the program. We decided to use wrappers around the XMLBeans-classes to add listener functionality required by the MVC pattern.

[1] If we need to change the data models, only the schema will be changed, not the class diagrams.

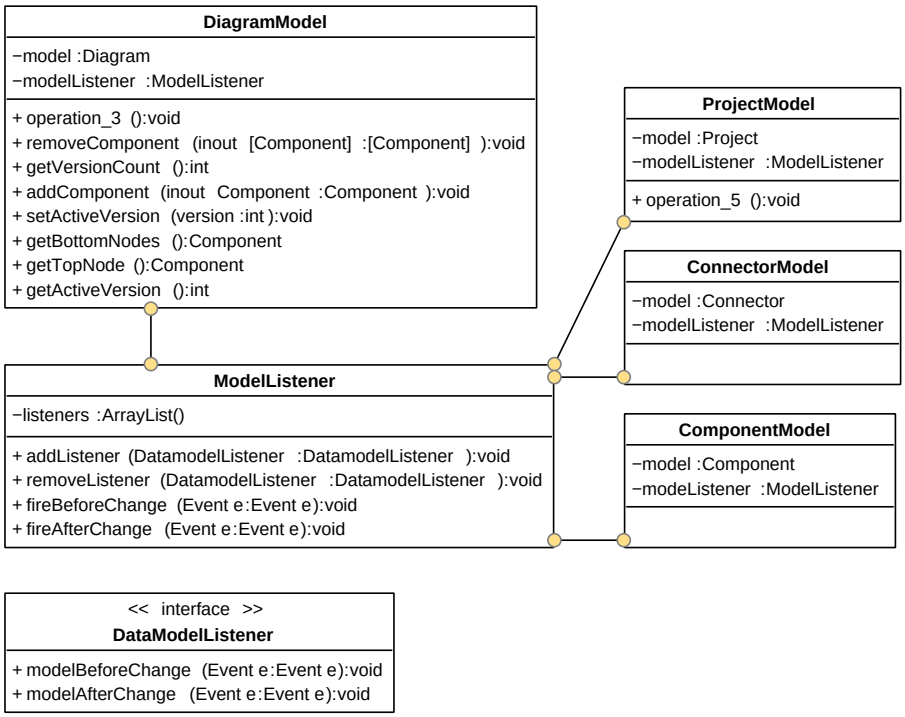


Figure 5.12: Storage class diagram

Please refer to the sequence diagram in Figure 5.13 for an illustration of a typical scenario. The user clicks the save-icon on the toolbar in the main window. The main window calls the Controller with a saveButtonEvent. The Controller calls saveProject on the ProJectModel. And finally the ProjectModel calls the save-function in the JAR-file.

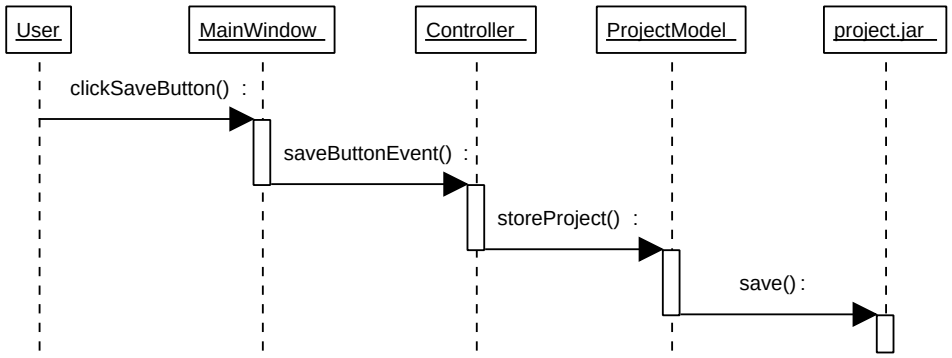


Figure 5.13: Typical save-scenario

5.5 Summary

We started this phase by making a Work Breakdown Structure. The approach was helpful in dividing the problem into smaller problems, and we learned where our understanding was lacking. After sketching the general architecture, potential problems, a least decisions we think can have profound effect on the delivered product, was then listed. Most uncertainty was attached to storage, so that was modeled first. Afterward the rest of the system was modeled using class- and sequence diagrams in parallel. Only essential parts were modeled, so most internal methods and attributes are left out.

We used some prototyping both in designing the GUI and when exploring XMLBeans. In addition we wrote the XML Schema. This gave a sliding transition between the design phase and the implementation phase, as planned. This process revealed some minor mismatches between the requirement specification and our design. Other potentially problematic decisions are explained in Section 5.3.

We believe that our design is complete and well elaborated. The most prominent problem for the next phase is the time aspect.

Chapter 6

Implementation

6.1 Introduction

In this part we will describe how we have implemented SP Light. The primary goal of this part is to describe in some detail how we have implemented our product. This goal will be attained through the following three sections:

- First, we will describe the technology and standards we have used in order to help us create readable and quality code. We will describe the programs and tools used during the implementation and also coding conventions.
- Second, we will describe certain parts of the code base in a more detailed manner. We will look at what changes we made from the original design, described in the design document, and we will also describe and discuss certain nontrivial algorithms and methods used in the application.
- Finally, we will explain every requirement that has not been implemented. We will also explain when a requirement has not been implemented as planned.

6.2 Technology and Standards

In this section we will describe the technologies, standards and tools we have used in the implementation phase.

6.2.1 Version Control

We chose to use CVS [2W] for version control. CVS was preferred over Subversion and Arch because it is supported at `login.stud.ntnu.no` and because all the members of the group were familiar with it. CVS has worked well and fulfilled our needs for version control without any noteworthy problems.

6.2.2 Integrated Development Environment (IDE)

We have chosen to use the Eclipse [3W] IDE. It provides very good CVS support, it is open source and it has basically the same functionality as for example IntelliJ. Eclipse has worked well except for some problems with cvs+ssh to `login.stud.ntnu.no` (Linux), but using `solarislogin.stud.ntnu.no`, instead solved this.

6.2.3 XML Schema

We have used the XML Schema standard (also known as WXS or XSD) to define the structure of the XML documents used by SP Light. An XML schema defines the data types that can be used in the XML document and also how many instances of a type that is allowed or required. It also makes it possible to do validation of XML files to check if they conform to the schema. See [18W] for more information about XML Schema.

6.2.4 XMLBeans

Apache XMLBeans 2.0.0 from [19W] has been used for the implementation of XML storage in SP Light. XMLBeans allows us to compile our XML schema into Java classes representing the types defined in the schema. These classes handle translation between XML and Java types and provide getters and setters for the types in typical JavaBeans [23W] fashion. XMLBeans makes saving and loading of XML data trivial.

The only drawback to this solution is the lack of an event system in XMLBeans. In order to get XMLBeans working with our intended MVC solution, without modifying the generated code, we had to program a wrapper layer between the XMLBeans classes and the rest of SP Light to handle events and listeners. See section 6.3.2 for further discussion of the storage package.

6.2.5 Java Coding Conventions

We have chosen to use Sun's own coding conventions as described in [1W] because it is close to the groups preferred style of programming. Since time is scarce we do not check that the conventions are strictly followed, but obvious mistakes are corrected. The use of Eclipse also helps consistent formatting of the code.

Please look at the code example Listing 6.1 for an illustration on how the conventions are followed.

```
private ArrayList<String> getLogicalStorages(Component component){
  ArrayList<String> result = new ArrayList<String>();
  int count = component.sizeOfLogicalStorageArray();
  for(int i=0;i<count;i++){
    result.add(component.getLogicalStorageArray(i));
  }
  return result;
}
```

Listing 6.1: Excerpt from splight.storage.ComponentModel.java

The complete source code can be found on the attached CD-ROM.

6.2.6 Documenting the Code

Early in the development of SP Light the customer expressed a desire to receive a product which could be modified and extended. The requirements specification also includes similar demands. We have done three things to satisfy these requirements:

1. Employed Javadoc
2. Added developer howtos to the wiki (see [30W])
3. Included the customer in the implementation process. [1]

Javadoc

Javadoc is Sun's recommended way to document Java code. It was chosen because it is easily read (HTML is generated with the Javadoc tool) and because it can be written along with the code. Javadoc tags have been written for all classes, as well as all non-trivial methods. Trivial methods are defined to be simple getters and setters, toString(), etc. Please look at Listing 6.2 for an illustration of how Javadoc is used.

[1] This does not mean that the customer has actually written Java code, but we have tried to explain our design decisions orally and demonstrated which packages and classes that handle the different parts of the program.

```
/**
 * This class is a wrapper for Diagrams from project.jar, which is ↩
  generated by XMLBeans.
 * It encapsulates load from XML and storage to XML.
 *
 * @author Erik
 *
 */
public class DiagramModel extends Model{


  /**
   * Deletes all connectors which are connected to the given ↩
    component.
   *
   * @param parentComponent ComponentModel to delete all ↩
    connectors from
   */
  private void deleteAllConnectors(ComponentModel parentComponent) {
    ArrayList<ConnectorModel> list = ↩
      getLowerConnectors(parentComponent);
    list.addAll(getUpperConnectors(parentComponent));
    while(!list.isEmpty()){
      removeConnector(list.get(0));
    }
  }


  /** It returns the ComponentType from the int stored in XML.
   * SUB_DIAGRAM=1, SUB_SYSTEM=2, DIAGRAM=3
   *
   *@return ComponentType
   */
  protected ComponentType getType() {
    int type = diagram.getType();
    switch(type){
      case 1:{
        return ComponentModel.ComponentType.SUB_DIAGRAM;
      }
      case 2:{
        return ComponentModel.ComponentType.SUB_SYSTEM;
      }
      case 3:{
        return ComponentModel.ComponentType.COMPONENT;
      }
      default:{
        return null;
      }
    }
  }
```

Listing 6.2: Excerpts from splight.storage.DiagramModel.java

6.3 Description of Implementation

A UML package diagram, showing the overall subdivision of the system, is shown below in Figure 6.1.

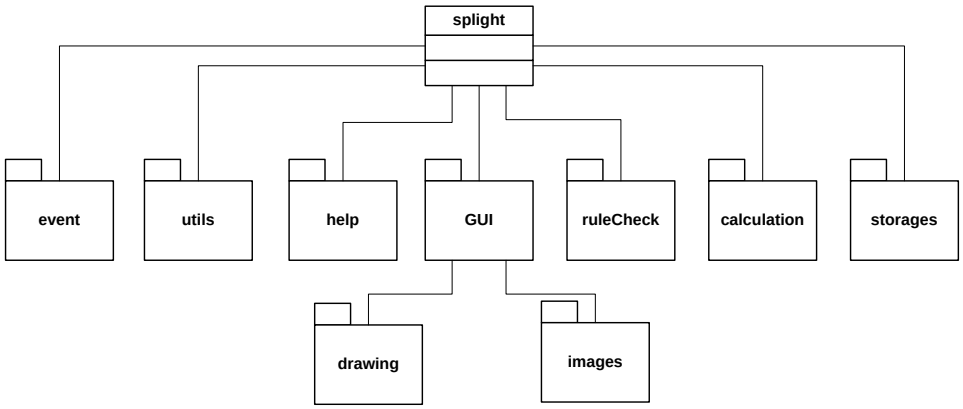


Figure 6.1: Package diagram

We renamed the `Debug` package to "utils", and we added a package "event" to hold all events. The GUI package is divided into drawing and images, where the latter includes only images used, while drawing includes all the code. Drawing is further divided into sub packages, but since this splitting is only essential for developers extending the functionality it is not shown here.

6.3.1 GUI

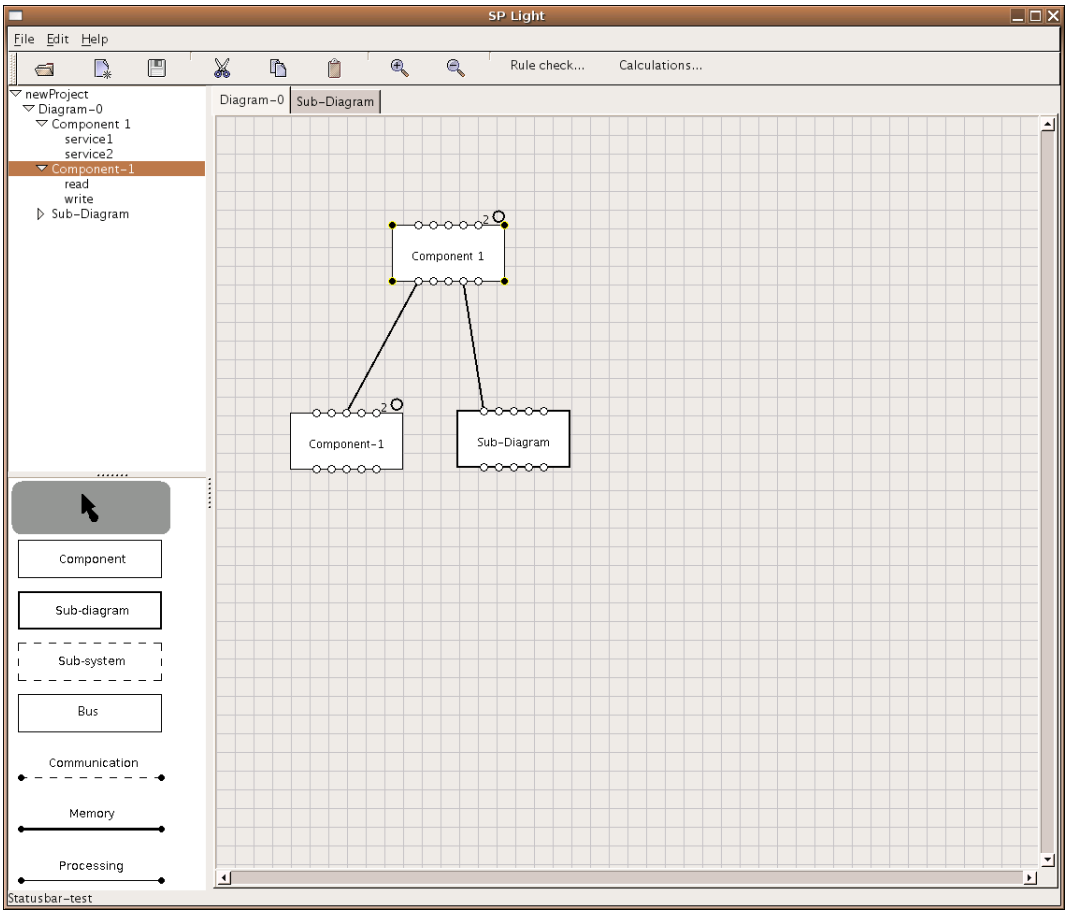


Figure 6.2: Screenshot: main window with tree structure, drawing area and toolbox.

Figure 6.2 shows the main window in the application. The window has a standard menu and tool bar, a tree structure displaying the open project, a toolbox containing drawing tools, and a drawing area. The graphical user interface with Java Swing components was not modeled very detailed before coding, since we needed some prototype experience with the components to know which component to use for different purposes. We also had to make some components manually. In the modeling phase, we only decided how the components were to communicate with each other in general. All graphical objects that needed to be updated when the underlaying model changed, registered themselves as listeners on the corresponding model object. When a model changed, it fired a change-event to all its listeners, including what had changed. This way, we ensured that all graphical objects were always displaying updated data. In some simple dialog boxes that only displayed a list of certain items, we forced the window to always be on top instead of using listeners. By not allowing the user to edit data elsewhere, we could be sure the dialog box data was consistent. This was one of the advantages of having a stand-alone application, since we did not have to consider more than one user.

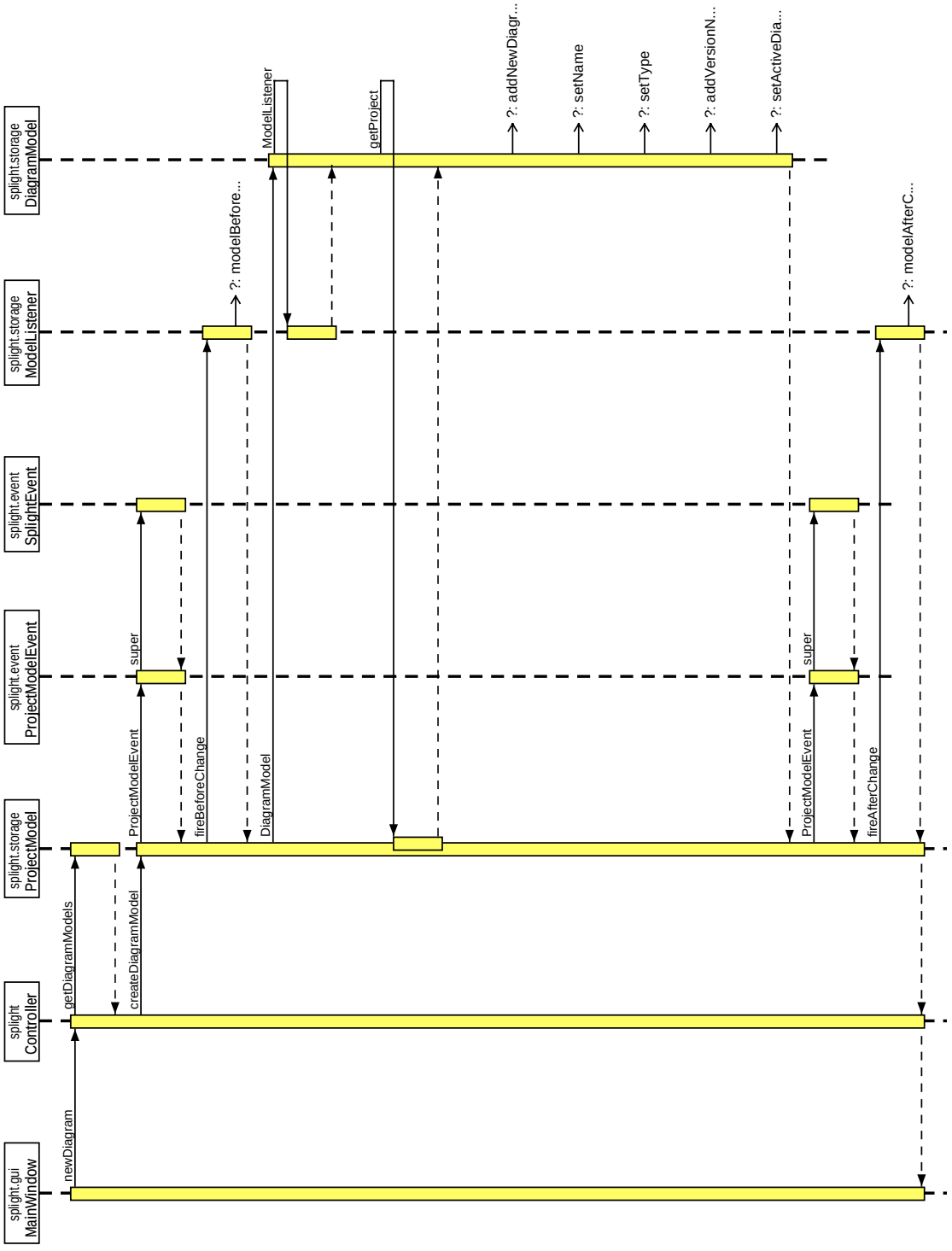


Figure 6.3: Sequence diagram for creating a new SP diagram

Figure 6.3 shows a sequence diagram for when a new SP diagram is added to a project.

Tree navigation

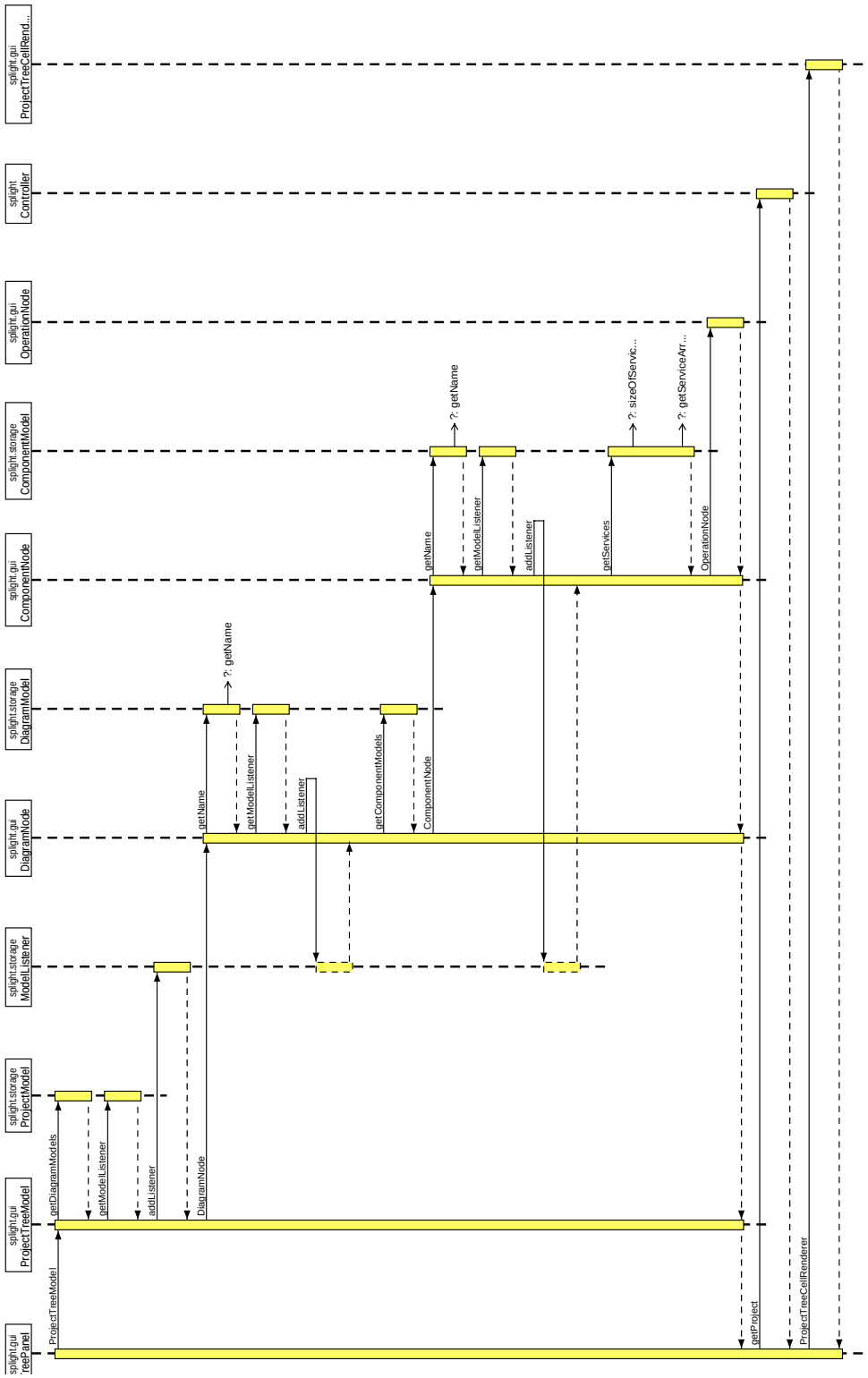


Figure 6.4: Sequence diagram: Constructing a tree model

The tree structure in the upper left part of the main screen (see 6.2) is used to navigate in the project, and gives an architectural overview of the project. This

tree structure was not a part of the original requirements specification given by the customer at day one, but was included in our final requirement specification as a replacement for three different views (architectural view, project view and design view). With the tree panel, it is possible to have a quick look at all the diagrams and components, but it is also possible to expand each diagram or component to show operations on each component or sub-diagrams with all their components. The tree structure is implemented in a separate class named `TreePanel`. `TreePanel` extends a normal drawable `JPanel` from Java Swing, and has a standard `JTree` swing component. A class named `ProjectTreeModel` serves as a model for the `JTree` component. Figure 6.4 shows how the tree communicates with other components. When loading a project, the following sequence is used:

1. A new `ProjectTreeModel` is made on the data from the project file.
2. In the constructor of `ProjectTreeModel`, all diagrams in the project are added as children of the tree model by creating new instances of the `DiagramNode` class.
3. In the constructor of a DiagramNode, the following things happen:
   (a) All components in the diagram are added as children of the diagram node by making new instances of the `ComponentNode` class.
   (b) All sub-diagram/sub-systems in the diagram are added as children of the diagram node by making new instances of the `DiagramNode` class.
4. In the constructor of a `ComponentNode`, all services in the component are added as children of the component node by making new instances of the `OperationNode` class.

Each tree node listens for events from the model it represents, and updates itself when it changes, or when a child is added or removed. The `TreePanel` class listens for mouse events and performs the right action when a node is being clicked, double-clicked or right-clicked, by finding the node at the clicked position. A click on a diagram node causes the diagram to open in a new tab if it is not already open, or focuses the existing tab if it is open. A click on a component node pops up the component editor window.

Drawing Panel

In the GUI implementation, the part that allows the user to construct the models has been named “drawing panel”. It is responsible for receiving user-input and rendering the model onto a JPanel’s Graphics2D object. It also provides listener functionality so that the GUI can open editor dialogs when a component is double-clicked, etc.

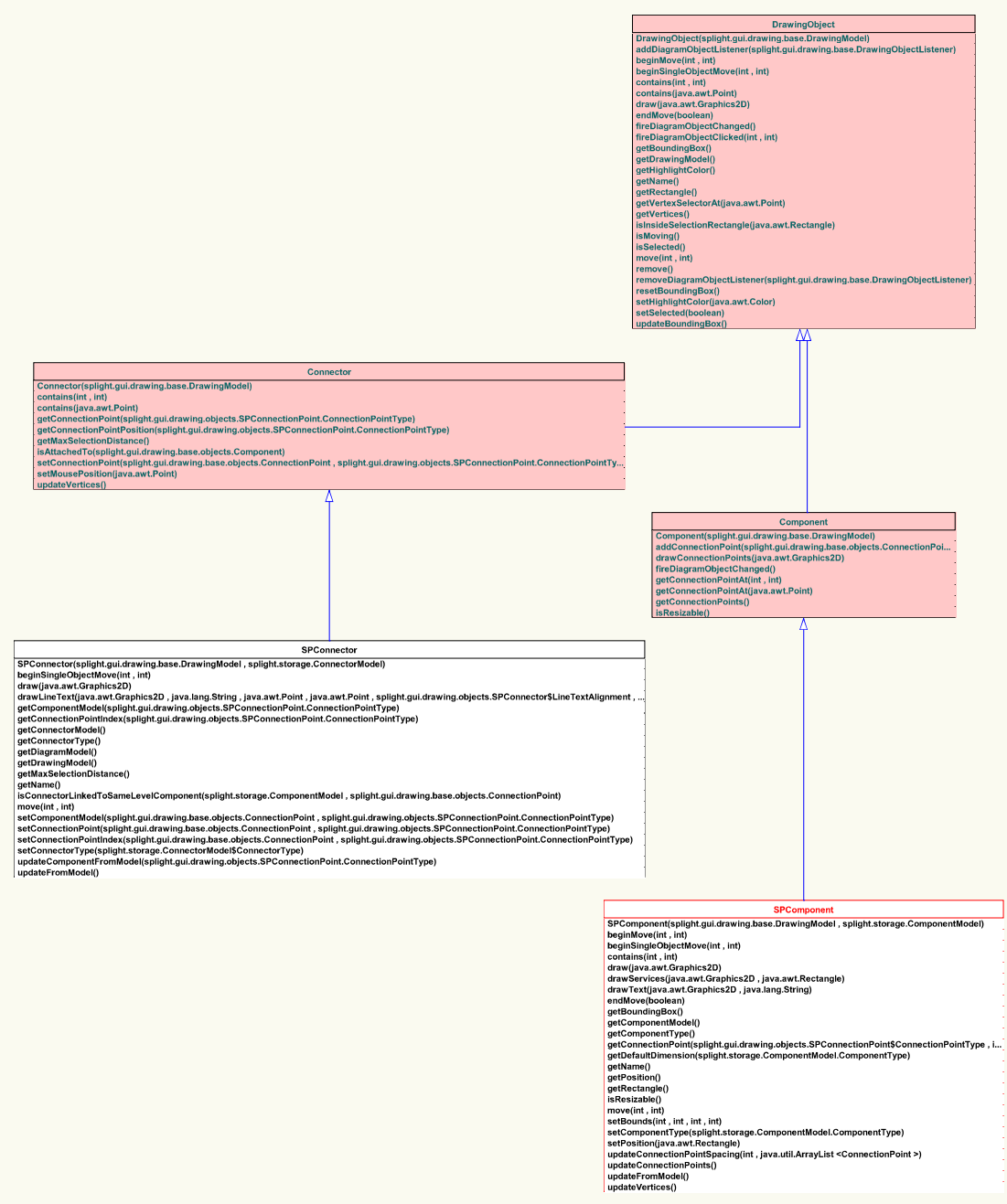


Figure 6.5: Class diagram for drawing objects

The most basic type in a drawing panel is the drawing object. It is the base class for all elements that are used with the drawing panel, like components and connectors. Figure 6.5 shows a class diagram of DrawingObject and its subclasses. Important features of a drawing object is its ability to be drawn, moved around, and do contains test to see if a given position is inside the object.

Both components and connectors are split into a generic base class and an SP

specific subclass. This is to make it easier to extend the drawing panel with more element types later that similar to components but do not use the same data models as the `SPComponent`.

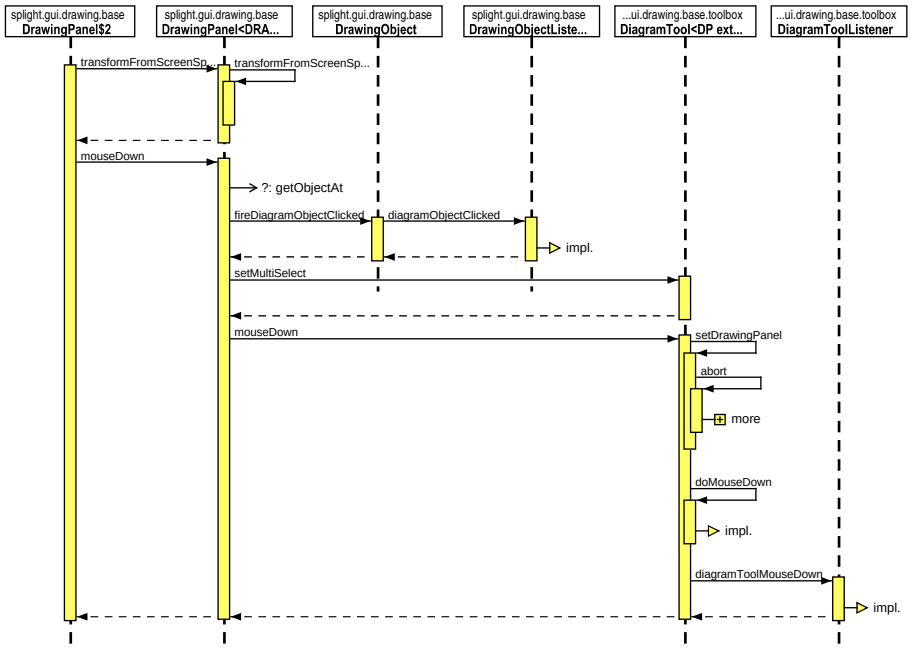


Figure 6.6: Sequence diagram for mouse interaction within the drawing panel

Figure 6.6 shows what happens when the user makes a mouse click on the drawing panel. First the coordinate must be translated from screen space to drawing panel space. This is to make sure the right drawing object is selected even when zoomed in/out. Next, it will check if there is a drawing object located at that position in the drawing panel. This method will first look for a matching component, and only if there is no matching component will it try to look for a matching connector. This check is done by first testing the against the drawing object's bounding box before a more complicated check is done to see if it is really within the bounds of the drawing object.

6.3.2 Storage

The storage package handles persistence. We use the classes compiled from the XML schema by XMLBeans as the interface to XML files and our own corresponding wrapper classes for interaction with the rest of the program. For each of the classes representing an XML type that is to be accessed from other parts of the program we have written a model class that handles type translation and events. Eg. the `Diagram` class compiled by XMLBeans is wrapped in our `DiagramModel` class. This allows us to use practical types where we want to, like an `ArrayList` instead of an array. It is also necessary to be able to trigger events used for updating the user interface.

All data models extend the `Model` class, to allow collecting different `Model` objects in one `ArrayList`. The `Model` class itself is empty. The real topmost object in the hierarchy is the `ProjectModel`, see Figure 6.7. Each `ProjectModel` has a list of one or more `DiagramModels` and a list of library functions. The library functions are just objects containing three simple strings; name, function and description, and will not be described further here.

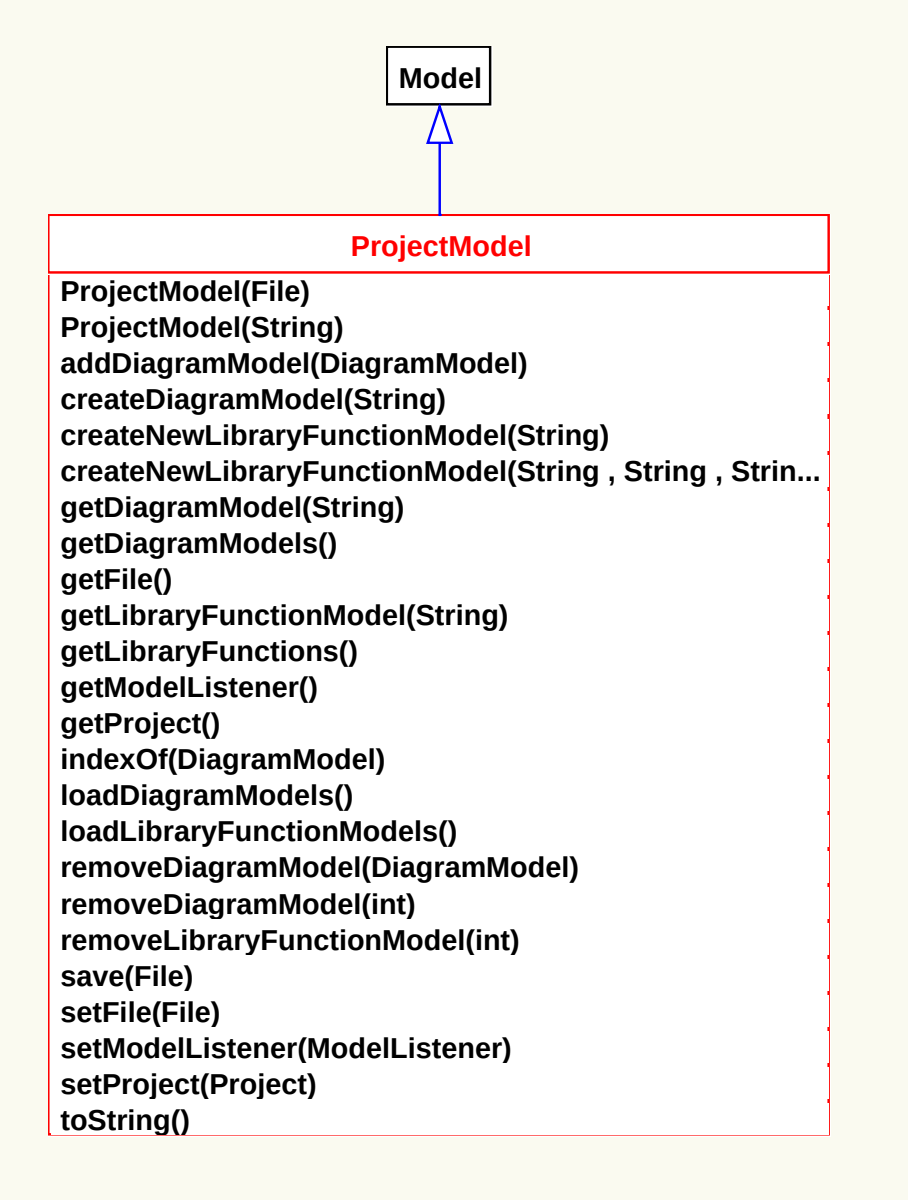


Figure 6.7: `ProjectModel` class diagram

Loading from and saving to XML can only be done at the project level. Figure 6.8 explains the sequence when saving. The XMLFilter class is used to force the `.xml` file extension. The `Controller` class can be used from different parts in the GUI to save the project. The `ProjectModel` class in turn calls methods provided by XMLBeans to store the contents of the project.

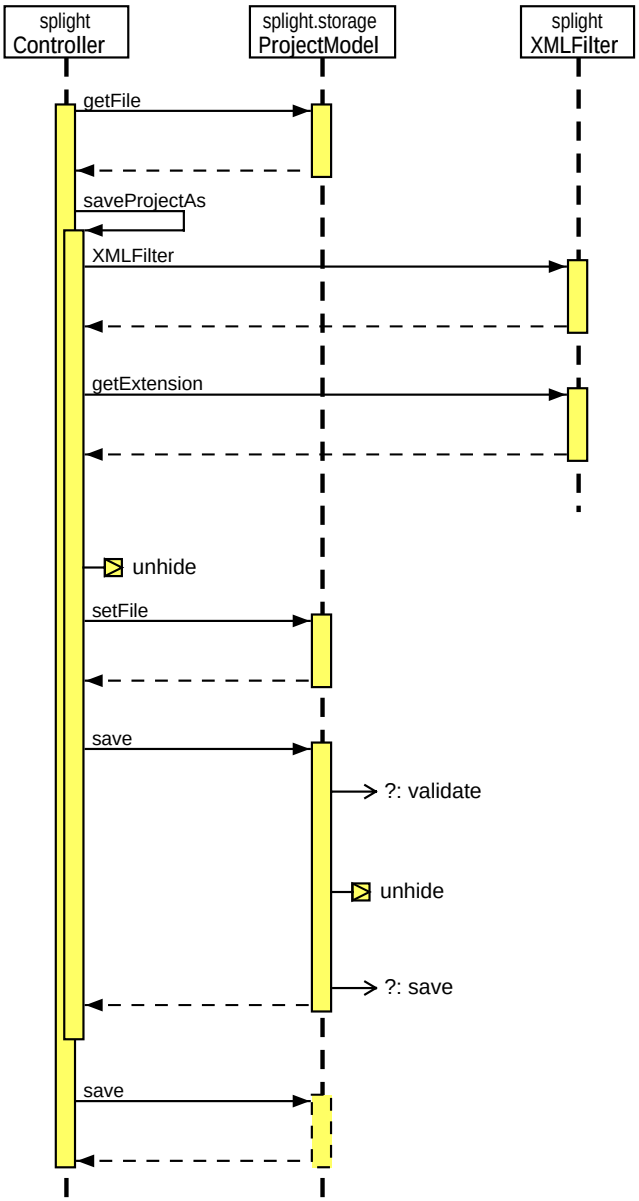


Figure 6.8: Sequence diagram for the save operation

`DiagramModel` is a rather complex class, as Figure 6.9 clearly shows.

Figure 6.9: `DiagramModel` class diagram

Each `DiagramModel` has a list of `ComponentModels`. See Figure 6.10 for more details. Basically, it holds two lists and provide methods to modify them. These lists hold connectors and components.

Figure 6.10: `ComponentModel` class diagram

Connectors store the relationship between two components. Each connector therefore holds a reference to two components, where the reference is the component's name. Functionality provided is listed in Figure 6.11.

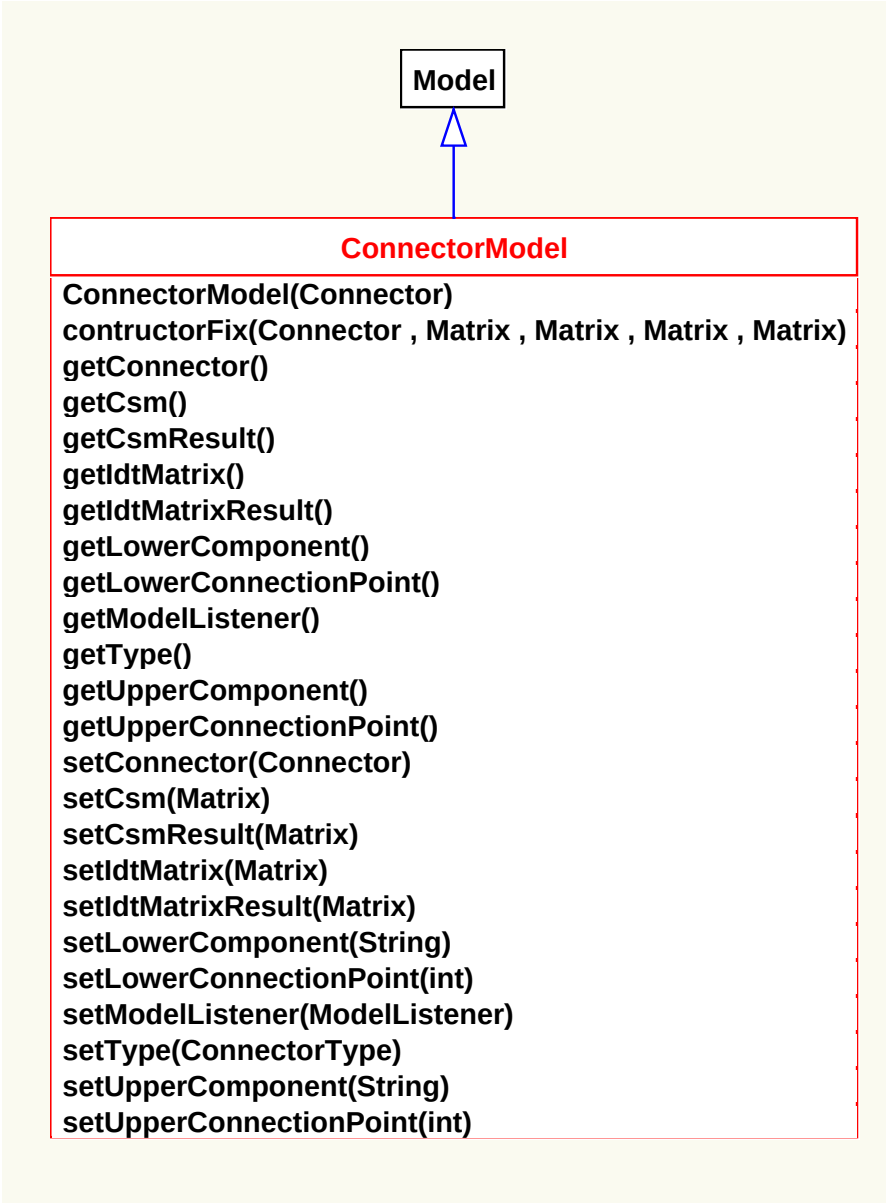


Figure 6.11: `ConnectorModel` class diagram

The relationship also entails two matrices; one for complexity (CSM) and one for compactness (IDT matrix). The sequence diagram 6.12 below shows how to set such a matrix. This call also updates the underlying XMLBeans-objects.

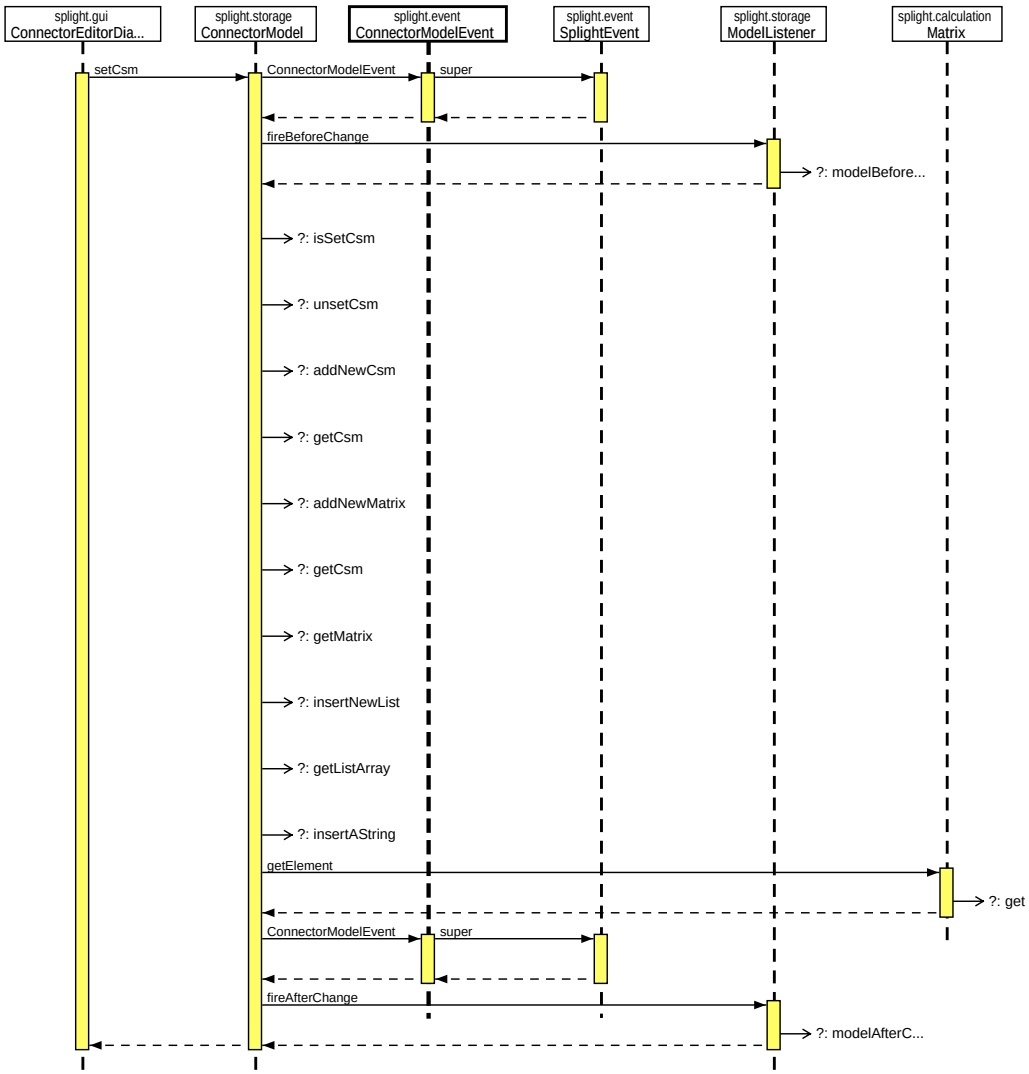


Figure 6.12: Sequence diagram for the `setCsm` method

Connectors and components are the basic elements of SP. A project has diagrams and diagrams has lists of connectors and components. When sub-diagrams and sub-systems are added, things get a bit more complicated. Both can contain components and connectors, and both can also be treated as a component. This relationship is not supported by XML, so we were forced to build a layer on top of XML to handle these relations. The chosen solution was designed to let a `ComponentModel` contain a `DiagramModel`, but only a reference is stored in XML. This reference was chosen to be the name of the component, giving the component holding a diagram the same name as the diagram. Hereafter names had to be unique within a project. Perhaps a better and/or simpler solution would be to use numerical ids instead of names, but at the time it did not seem to be a better solution. We are still not confident that ids are a better solution, because the relationship denoted by the same name of a component and a diagram would be lost. Further development of SP Light should look into this.

6.3.3 Calculation

Calculations in SP Light are handled by the `splight.calculation` package. This package contains a parser for expressions, a class representing matrices and providing some basic matrix operations, and a class handling calculation of complexity and compactness.

Calculation in SP Diagrams

Calculations are performed by `Calculations.java`. In SP Light calculations are initiated from the calculation window, and an instance of the Calculations class with a reference to the currently selected/active diagram is created. Complexity/Compactness calculations depend on the corresponding workload/dataload vectors to be set for the diagram, and these can be entered in the calculation window. If complexity, compactness or both calculations should be performed, is also specified in the calculation window. To successfully perform calculations, a unique top node must exist in the diagram, and all primitive components must be marked as such. Performing a rule check on the diagram will ensure that these requirements are met.

In addition to the CSM and IDT matrices that are specified for each connector, the connector and component models also have result variables that are used in calculations. These variables are temporary and not saved to disk together with the diagram/project, and results are only available when calculations have been performed. The calculation of complexity and compactness are based on matrix multiplication and addition, and the algorithms for performing the calculations are basically the same except that compactness is only calculated for memory connectors. The algorithm is based on recursion from the primitive (bottom) nodes and up the graph.

A high level description of the important steps in the algorithm:

- Clear possibly existing results on all components and connectors.
- Set result = input vector on the top node.
- Call the recursive calculate method on all primitive components.

The recursive method returns the result vector of the component, and this is found by adding the results of the multiplication of the specification matrix of each connector going into the component and the result vector of the corresponding upper component. The result variables for the connectors are also set after the multiplication is done, so that each connector's contribution to the devolved work on the component is shown. This approach works because the result on the uppermost component (the top node) is set, and when reaching this component the call will return. Because the result values are set on each connector and component as the algorithm executes the full path to the top node is not traversed with every call to the primitive nodes. Instead the result can be returned right away when it is set on a component.

The Parser for Mathematical Expressions

SP Light contains a parser which can syntax check, parse and evaluate mathematical expressions. The parser was implemented in three stages:

- First, an grammar was created for the language the parser will interpret.
- Second, a tokenizer was implemented in `ParserTokenizer.java`, package `splight.calculations`. This simply takes as input a string which is a mathematical expression, for instance `3*2+10`, and returns it as a list of tokens, which in this example would be the list `(3,*,2,+,10)`.
- Third, the actual parser was implemented using a *predictive recursive descent parser*. These types of parsers are very easy to implement, extend and understand, but are less flexible than other types of parsers (for instance *bottom-up parsers*). For general theory on parsers, we refer to [4B].

Let us now look at each element of the parser.

The Grammar The grammar of the parser in EBNF notation is shown in the JavaDoc for the class `Parser.java`. It is repeated below. We refer to [26W] for more information about EBNF notation.

```
expression      ::= empty | arithmetic_expr |
    bool_expr '|' arithmetic_expr ':' arithmetic_expr
bool_expr       ::= '!'? comp_expr ( BOP '!'? comp_expr )*
comp_expr       ::= arithmetic_expr COP arithmetic_expr
arithmetic_expr ::= fact_expr ( AOP fact_expr )*
fact_expr       ::= exp_expr ( FOP exp_expr )*
exp_expr        ::= primary_expr ( EOP primary_expr )*
primary_expr    ::= number | '(' expression ')'
                    | '[' function ']'
number          ::= (Java double)
function        ::= (Function from function library)
BOP             ::= '&&' | '||'
COP             ::= '>=' | '<=' | '<' | '>'
AOP             ::= '+' | '-'
FOP             ::= '*' | '/'
EOP             ::= '^'
```

The grammar above is subject to certain constraints which are not expressed in the grammar:

1. All operators are left associative. This is standard in mathematics [27W].
2. Empty expressions inside parentheses are not allowed.

The grammar is merely a formal description of a slight modification of the language used in standard mathematics. For instance, the strings

```
2*2
5*(5^3)+(2*(4-6)-(1-2)/2.1)
```

are strings of the language generated by the grammar, whereas

```
2*
5*(5^3)+(2*(4-6)-(1-2)/*2.1
```

are not strings of the language generated by the grammar.

The grammar describes the *syntax* of legal strings. The meaning of the strings, or *semantics*, are as in standard mathematics. However, the grammar has some possibilities that are not used in standard mathematics. In particular, references to functions in the function library are written inside brackets. Also, the grammar supports the *guard operator*, which is used as a conditional expression. The following expression

```
2 >= 3 && 5 <= [f(x)] | 1 : 0
```

is legal according to the grammar and is interpreted as follows: if 2 is greater or equal to 3 and 5 is less than or equal to the value of the function `f(x)` from the function library, the value of the expression is 1. Otherwise, the value of the expression is 0.

Now, all we need to do to recognize such expressions is to write a parser for the grammar. The next step in doing this is writing the *tokenizer*.

The Tokenizer As mentioned, the tokenizer takes an input string and returns a list of terminal symbols that build the string. We will not say much about its implementation here, since it is not particularly complicated. It is written in `ParserTokenizer.java` in the package `splight.calculation`.

The Main Parser The main parser takes a list of tokens from the tokenizer, recognizes the tokens as a part of the grammar, and evaluates the input string to a number. As we mentioned, the parser is a *predictive recursive descent parser*. This means that we have used the following strategy for implementing the parser:

- For each nonterminal symbol in the grammar, generate a new method which parses the symbol. Let R be the rule in the grammar which defines the nonterminal N. R will typically define the nonterminal symbol using other terminal and/or nonterminal symbols; let these be respectively $T_1, T_2, \ldots, T_n$ and $N_1, N_2, \ldots, N_m$. The implemented method will move forward in the token list until N is parsed. Each token is either one of the T_is or one of the N_js[2]. If the method detects a terminal symbol T_i, it checks that the detected symbol is the same as the symbol expected in R. If it is not, it reports a syntax error. If the method detects a nonterminal symbol N_j, the appropriate method for parsing that nonterminal symbol is called, and the method continues parsing the token after the tokens of the N_j symbol. This continues until the entire symbol is parsed.

Let us illustrate how we have used this strategy with an example from the `Parser.java`. This is shown in Listing 6.3 below.

[2]A rule may hav several alternative productions. If this is the case, the method for parsing N must decide what production to use. This is not a problem in the present grammar.

```
  /**
   * Parse a primary expression.
   *
   * @return The value of the expression
   */
  private double primaryExpression(String[] tokens, int startIdx,
    int endIdx) throws ParserException {
  /*
   * Parse a primary expression:
   *
   *  primary_expr     ::= number | '(' expression ')' | '['
   *    function ']'
   *
   */
  switch( tokens[startIdx].charAt(0) ) {
  case '(': // second alternative
    if( tokens[endIdx].charAt(0) != ')' ) {
      throw new ParserException("Syntax error: Parentheses
      in expression do not match:
      Found beginning left parenthesis, but no
      matching right parenthesis");
    } else if( endIdx == startIdx + 1 ) {
      throw new ParserException("Syntax error: Expression
      inside parentheses must be nonempty");
    } else {
      return expression( tokens, startIdx + 1, endIdx - 1 );
  }
  case '[': // third alternative
  if( tokens[endIdx].charAt(0) != ']' ) {
    throw new ParserException("Syntax error: Brackets in
      expression do not match: Found
      beginning left bracket, but no matching
      right bracket");
  } else if( endIdx == startIdx + 1 ) {
    throw new ParserException("Syntax error: Expression
      inside brackets must be
      nonempty (and refer to a function)");
  } else if( pm != null ){
    LibraryFunctionModel func =
      pm.getLibraryFunctionModel(tokens[startIdx + 1]);
    if( func == null ) {
      throw new ParserException("Invalid expression:
      Referencing non-existant library
      function '"+tokens[startIdx+1]+"'");
    } else {
      return parse( func.getFunction() );
    }
  } else {
    return 0.0; // for debugging
  }
  default: // first alternative
  return number( tokens, startIdx, endIdx );
  }
}
```

Listing 6.3: Excerpt from Parser.java

The function in Listing 6.3 parses the grammar rule:

```
primary_expr    ::= number | '(' expression ')'
                  | '[' function ']'
```

The rule says that a `primary_expr` is either a nonterminal number symbol, the terminal symbol '(' followed by a nonterminal expression symbol followed by the terminal ')' symbol, or the terminal symbol '[' followed by a function nonterminal symbol (i.e. a function from the function library), followed by the terminal symbol ']'.

The function in Listing 6.3 takes in a list of tokens and the indices `startIdx` and `endIdx` which gives the range in the token list the function shall parse. The function first checks the token at the start to determine whether this is an instance of the first, second or third case[3]. Next, it calls the appropriate routines for parsing nonterminal symbols.

For instance, when the parser discovers the symbol '(', it knows that the symbol in the token list at index `endIdx` must be ')'. If this is not the case, it reports an error. Also, the symbol in between the parentheses terminals must be the nonterminal symbol `expression`. For parsing this symbol, it just calls the routine `expression` which parses and evaluates an `expression` symbol. Note that this follows our strategy: Terminal symbols are parsed directly, and we have one method for each nonterminal symbol in the grammar.

This design allows the grammar to easily be extended with new operators, as we shall see in Section 6.4.3.

[3]This is why the parser is called *predictive*: While parsing, it predicts what types of symbols comes next. It is a *recursive descent* parser because it starts with the start symbol of the grammar and repeatedly parses the nonterminal symbols until there are only terminal symbols left

6.3.4 Rule Checking

A major component of SP Light is the ability to check that diagrams follow certain rules. The rules implemented are described in our wiki [30W], and we repeat them here:

1. Any non-primitive SP component must have at least one processing subcomponent and at least one memory subcomponent.
2. If processing in a non-primitive SP component is distributed, there should be one processing subcomponent for each distributed processing unit, e.g., CPUs.
3. If persistent storage for a non-primitive SP component is distributed, there should be one memory subcomponent for each distributed storage unit.
4. If either processing or persistent memory (or both) of a non-primitive SP component is distributed, a communication subcomponent is required. Only one communication subcomponent for each non-primitive SP component is allowed.
5. A non-primitive SP component cannot be a communication component if one of its ancestor components is a communication component.

In Figure 6.13 below, a screen shot of a running instance of SP Light is shown with a diagram that contains errors. The rules in error are colored red. The rule check error log generated by the program for the diagram in Figure 6.13 is shown in Figure 6.14.

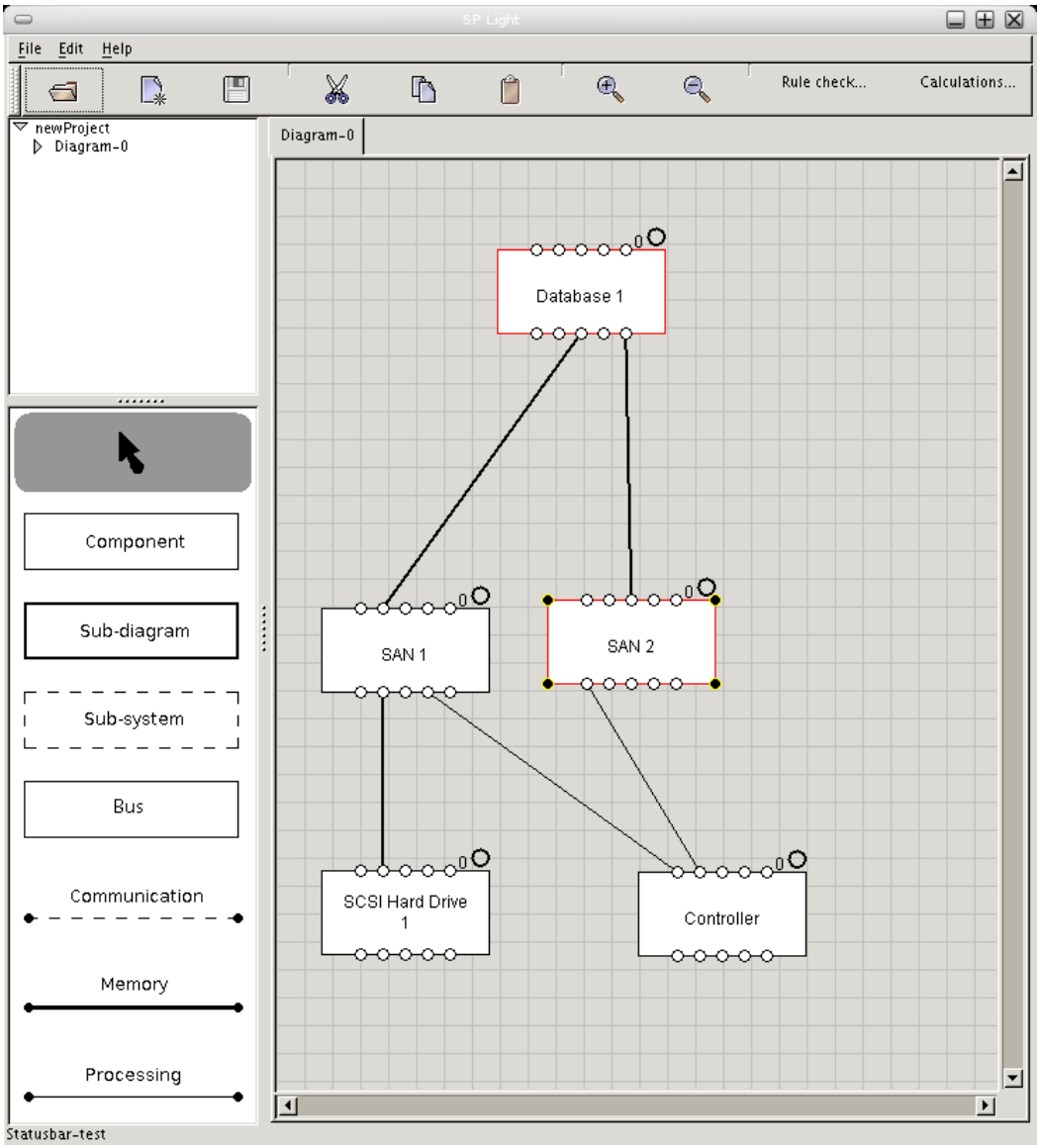


Figure 6.13: Rule checking in action in SP Light

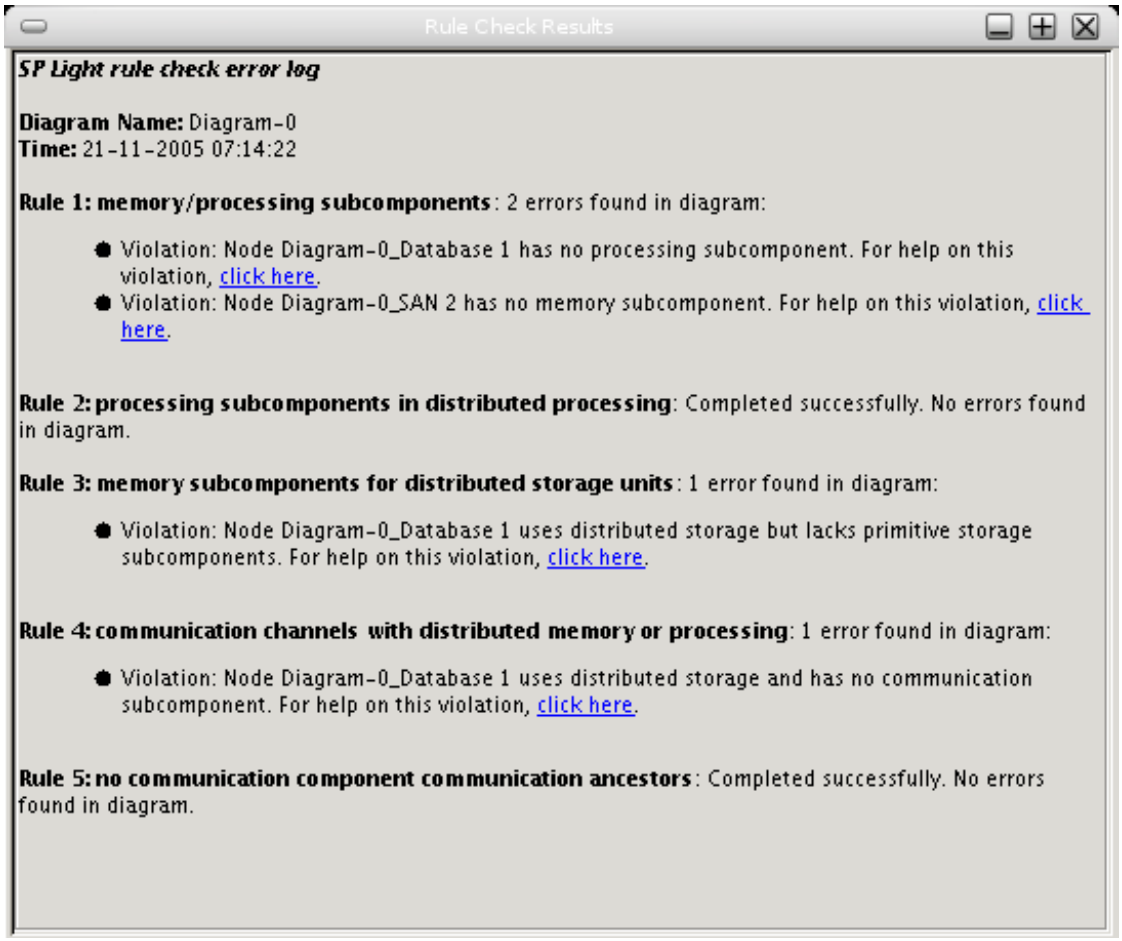


Figure 6.14: Rule check error log

The customer has decided that we implement these rules. As described in the requirements specification in Chapter 4, the customer wants developer users to easily be able to create and modify the rules the program checks. In this section, we will first give an overview of the rule checking system. Then we will look at how a specific rule check has been implemented. Finally, we will describe how developer users can easily modify and extend the rule checking system.

The Rule Check System

An UML class diagram of the rule check system is shown in Figures 6.15 and 6.16 below. Note that not all methods in all the classes in Figure 6.15 are shown for clarity reasons.

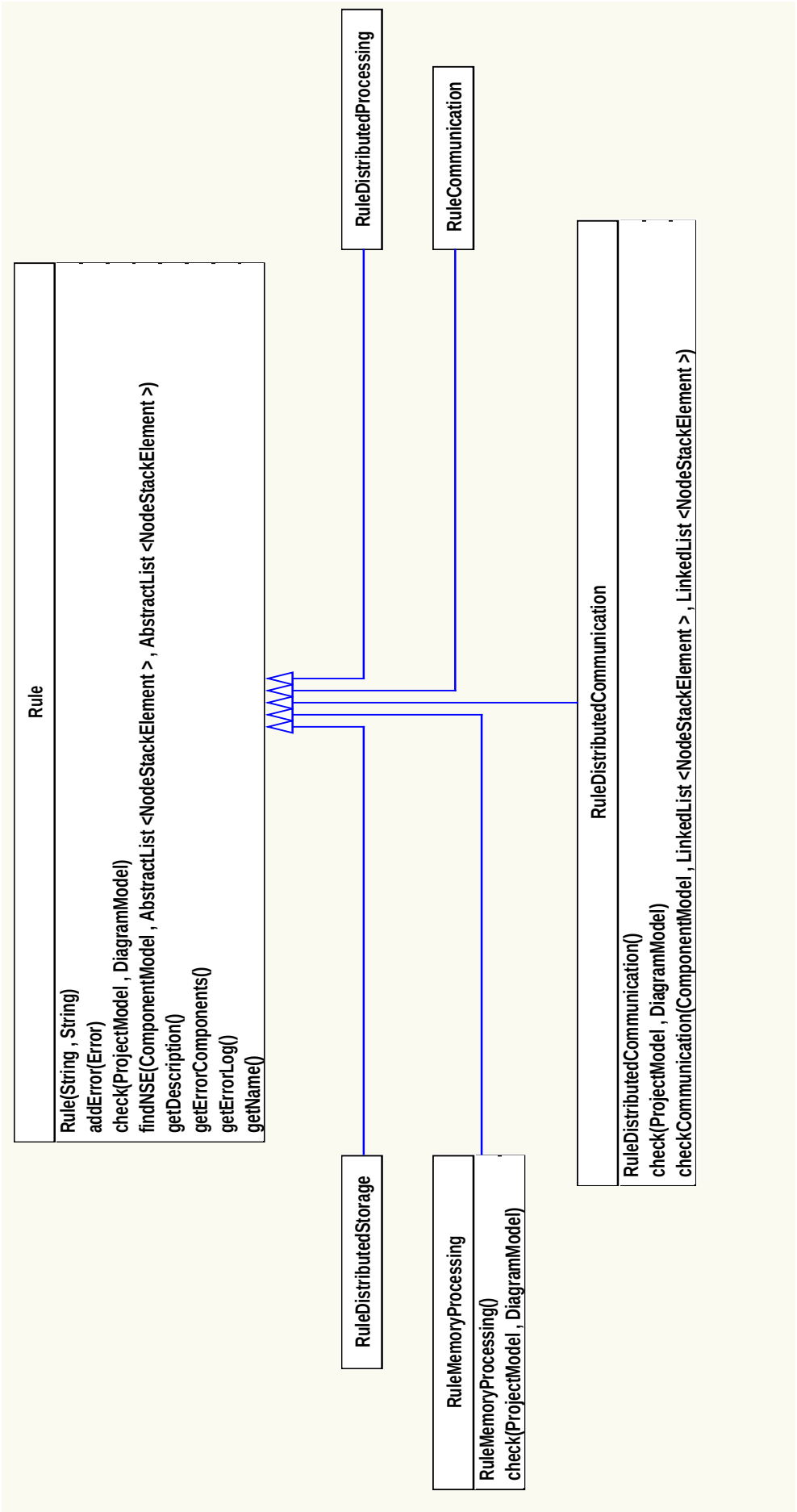


Figure 6.15: UML class diagram for the rule check package, part 1.

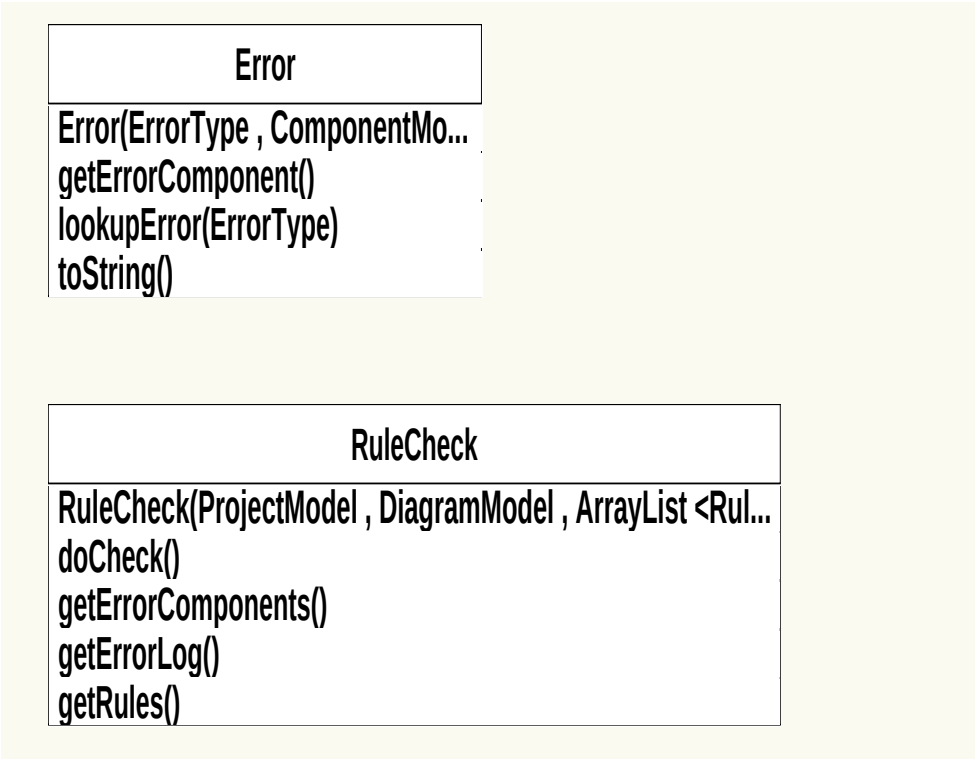


Figure 6.16: UML class diagram for the rule check package, part 2

The rule check system consists of the following classes:

- RuleCheck. This class serves as between the graphical user interface and the rule checking subsystem. It includes methods for performing the actual checking and also produces the error log shown in the GUI after the rule checking is complete.
- Rule. This is an abstract class containing methods common for all the specific rules. For example, it contains the addError method which is called whenever a violation of a specific rule is found.
- Specific rule classes. These classes, which are subclasses of `Rule`, contain the actual implementation of the rule checking. For instance, `RuleMemoryProcessing` contains methods to check if a diagram follows the memory/processing connectors rule.
- Error. This class represents a single error in a diagram. Typically, a specific rule will add Error objects whenever it finds an error. The Rule class keeps track of all the errors found with a specific rule.

Let us look at exactly what happens during a rule check. To understand this, consider the sequence diagram shown in Figure 6.17.

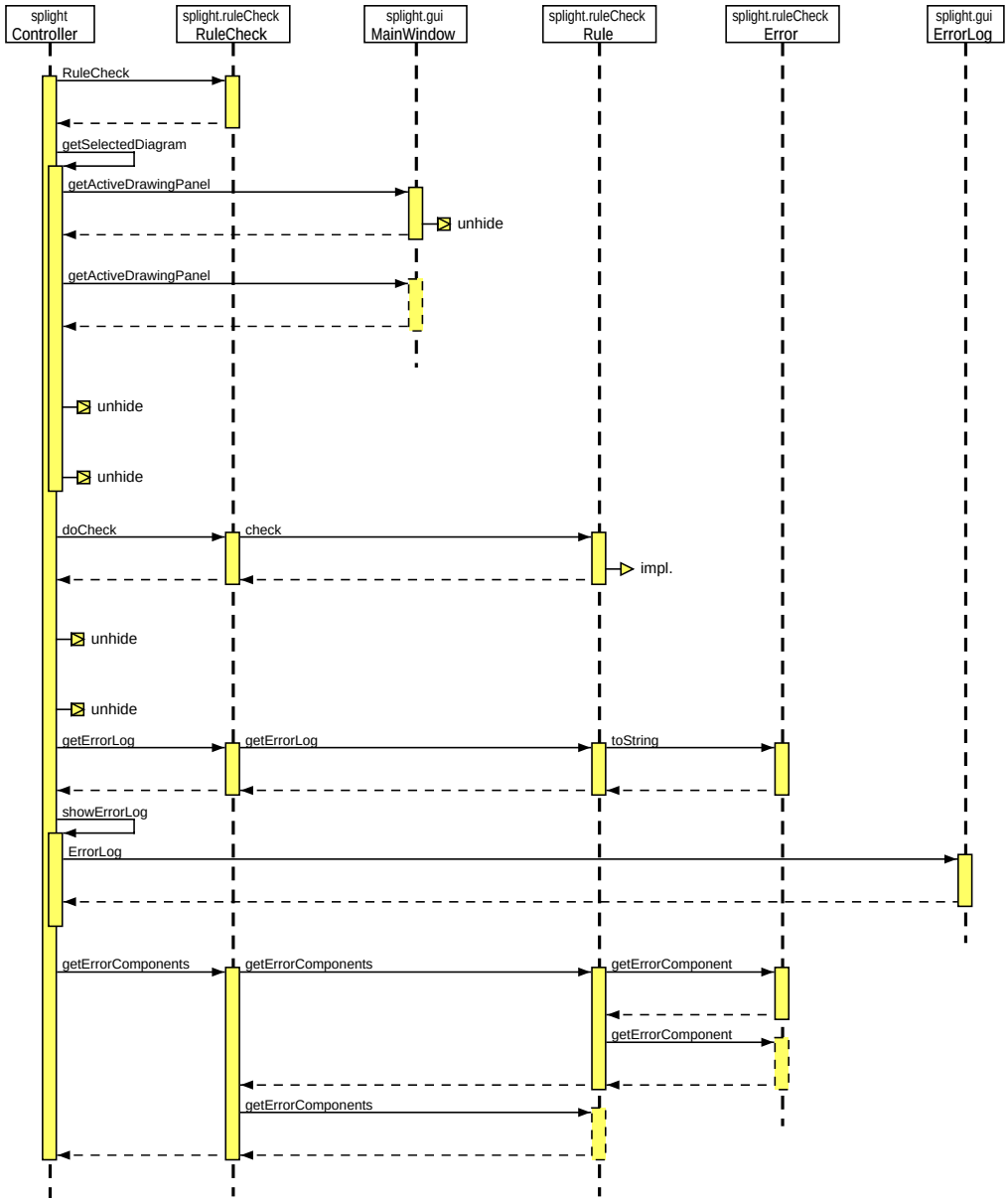


Figure 6.17: Sequence diagram for rule checking

First, the `Controller` class receives a list of instances of specific rule classes (e.g. `RuleMemoryProcessing`), in the method `ruleCheck`. This again calls the method `doCheck` in the `RuleCheck` class. The `doCheck` method calls the `check` method on the specific rule objects which are passed to it. The specific `check` methods check that the diagram does not violate the rule they represent. If any errors occur, they are added to the internal error list of the class, which is provided in the abstract `Rule` class. When all the checking is complete, control returns to the `Controller` class, which calls on `RuleCheck` to generate an HTML error log. This error log is subsequently shown to the user. Also, the controller retrieves a list of components which violate a rule

and changes their color to red in the diagram[4].

Let us now look at how checking a single rule is implemented.

A Specific Rule

In this section we shall look at the algorithms we used for implementing the first rule described in Section 6.3.4. This is implemented in `RuleMemoryProcessing.java` in the package `splight.ruleCheck`.

The heart of the rule lies in the `check` method. This method shall return `true` if the diagram has no errors, or `false` if it has at least one error. The code is quite long, so we recommend looking in the source code for detailed information. We now give a high-level description of the implemented algorithm.

The first rule described in Section 6.3.4 states that any non primitive component in an SP diagram must have at least one outgoing memory connector and at least one outgoing processing connector.

[1B] suggests an implementation of the rule. This implementation is recursive and also loops infinitely if the diagram contains cycles. Also, if there are N paths to a node, the algorithm in [1B] does exactly the same work N times. Thus it is very flow. We have implemented a quicker algorithm which solves all of these problems. The algorithm is iterative and does the following steps:

1. (**Initialize.**) Let Q be a first-in first-out (FIFO) queue containing all the non primitive top nodes of the diagram. Also create an empty queue F.
2. (**Check for termination**.) If Q is empty, the algorithm completes. If not, continue to the next step.
3. (**Check a node.**) Remove the first node N of Q and add the node to F.
4. (**Add undiscovered nodes to the queue and check for components.**) For all connectors c going out of N: If the node at the other end of c is non primitive and not already in F or Q, add it to Q. Also, if c is a memory or processing connector, mark N to have this type of outgoing connector.
5. (**Check for violations.**) If N is not marked as having a memory connector or is not marked as having a processing connector, report an error.
6. (**Repeat.**) Go to Step 2.

This algorithm is iterative, and works with general graphs instead of trees (which was not the case for the algorithm in [1B]). The algorithm is merely a slightly modified breadth first search — for further details on graph traversal algorithms, we refer to [6B].

The implementations of the other rules are variations on exactly the same theme. The exception is the implementation of the check for rule number 5. This algorithm first uses a breadth first search to detect the communication components in the graph, and then runs a breadth first search from each of the found communication components to determine if another communication component lies hierarchically above it in the diagram.

[4]The user may clear the color by pressing a button in the main window

Modifying and Extending the Rule Check System

It was very important for the customer to be able to modify and extend the rule check system. The reason is that the SP modeling language is still under development, and one might later want to either add new rules or change or remove some of the rules we have implemented. Thus, doing this should be easy, and it really is through the current design.

We have created a tutorial which, in addition to being in this document, is also available from our wiki ([30W]). In this section, we will take a look at how to create a new rule by giving a step by step guide. We will create a new rule check for a very simple rule. The step by step guide also gives the developer what he needs to know how to change or remove the existing rules.

Let us now look at what we need to do in order to create a new rule. We will explain this through an example: We are going to create a new rule check which checks the following:

Primitive components may not have any outgoing connectors.

Note that primitive components have to be marked explicitly as being primitive in SP Light. So it is perfectly possible to create a diagram which has primitive components with outgoing connectors.

Let us look at how we will do this, step by step.

Step 1. Creating a New Class First, we create a class RulePrimitive which extends the `Rule` class. Now, in order to implement this class, we need to implement the abstract methods in `Rule`. There is only one, shown in Listing 6.4:

```
public abstract boolean check(ProjectModel project,
  DiagramModel diagram);
```

Listing 6.4: Abstract Method of the `Rule` class

This method takes a project and a diagram, and returns true if no rule violations were found, and false if at least one rule violation was found.

In addition, we must provide a name and a description for the rule. The description is the text which will show up in the GUI, and the name is a short name for the rule used in the error log. Rule has a constructor which keeps the name and description. Therefore, setting the name and description is done by merely calling the superclass constructor.

Thus our initial skeleton for the new rule class looks as in Listing 6.5 on the next page:

```
import java.util.*;
import splight.storage.*;

public class RulePrimitive extends Rule {
  public RulePrimitive() {
     super("primitive components",
    "Primitive components can not have
     any outgoing connectors");
  }
  public boolean check(ProjectModel project,
    DiagramModel diagram) {
     return true;
  }
};
```

Listing 6.5: Initial Skeleton for a New Rule

Step 2. Displaying the Rule in the GUI Before writing the actual rule check, we need to make sure that the GUI knows about the new rule. This is very easy to do. Locate the method getRules in the RuleCheck class, shown in Listing 6.6 below:

```
    public static Rule[] getRules() {
        Rule[] rules = {
                new RuleMemoryProcessing(),
                new RuleDistributedProcessing(),
                new RuleDistributedStorage(),
                new RuleDistributedCommunication(),
                new RuleCommunication()
        };

        return rules;
    }
```

Listing 6.6: The `getRules` method

To make the new rule show up in the GUI, we merely add it to this list. Thus, this method becomes as in Listing 6.7 below:

```
     public static Rule[] getRules() {
        Rule[] rules = {
                // ....
                new RulePrimitive()
        };

        return rules;
    }
```

Listing 6.7: Adding a New Rule to The GUI

That is it! The rule will now show up in the GUI.

Step 3. Adding New Violation Messages This rule will produce new violation messages. In this specific example, there is only one new message: a

primitive component has outgoing connectors. In order to keep all the error messages in one place, they are located in the `Error` class. So let us add a new error message, as shown in Listing 6.8:

We first look in the `Error` class. We have to add a new constant for the `ErrorType` enumeration. So, we add the error `PRIMITIVE_COMPONENT_HAS_OUTGOING_CONNECTORS`:

```
public static enum ErrorType {
  // ....
  PRIMITIVE_COMPONENT_HAS_OUTGOING_CONNECTORS
};
```

Listing 6.8: Adding a Error Type

Next, we add our new error to the `lookupError` method. This provides a description of the error and a URL which will be clickable and which shows more information about the error. We modify it as shown in Listing 6.9:

```
private void lookupError(ErrorType error) {
  switch(error) {
    // ...
    case PRIMITIVE_COMPONENT_HAS_OUTGOING_CONNECTORS:
      description = "Primitive component "
        +(src.toString())+" has outgoing connectors";
      url = new URL("http://splight.idi.ntnu.no/Acyclic_Diagrams");
      break;
  }
}
```

Listing 6.9: Adding Information About a New Error

Note that the component which violates the rule is stored in the field `src`.

Step 4. Coding the Actual Rule Finally, we can write the actual rule check. To implement this rule, we will traverse the diagram until we find primitive nodes. But we want to use quick algorithms. For instance, we do not want to waste time visiting the same node twice. This is avoided by marking the node as visited.

But how can we do this? Again, the Rule class helps us out. It contains methods and structures for marking the nodes we visit during the graph traversal. It also helps us to write iterative instead of recursive functions (which are slower and require more memory). But note that you are in no way obligated to use the support it provides, although it will probably be useful.

The key structure is `NodeStackElement`s. This is an internal class of `Rule`, and contains information we might want to store during the traversal. For instance, the field `visited` of this class is used to check if we have visited this node before or not.

In addition, it is common to use two queues: One to hold the currently discovered but not visited nodes, and another to hold the nodes we have visited and are completely done with. `Rule` provides us with the method `findNSE` which as arguments takes two queues and finds a corresponding `NodeStackElement`

or `null` if the node has not been seen before. You might find this useful, but that naturally depends on what your rule is. We can unfortunately not give any further general hints about how you are going to design your rule checking algorithm, but we recommend looking at the already implemented rules for examples. Particularly, we would like to emphasize that, due to the nature of SP Light, it is very possible that a user has created a diagram with cycles in it. Your algorithm should be able to handle this.

Let us now implement the check method for RulePrimitive. We implement the checking method by traversing the graph using a breadth first search until we find primitive nodes. These nodes are then checked. The final class is shown below in Listing 6.10:

```
import java.util.*;
import splight.storage.*;

public class RulePrimitive extends Rule {
  public RulePrimitive() {
     super("primitive components",
       "Primitive components can not have any
        outgoing connectors");
  }
  public boolean check(ProjectModel project,
     DiagramModel diagram) {
     int nErrors = 0;
     LinkedList<NodeStackElement> queue
       = new LinkedList<NodeStackElement>();
     LinkedList<NodeStackElement> finishedNodes
       = new LinkedList<NodeStackElement>();
     ArrayList<ComponentModel> topNodes;

     // First let us find the top nodes.
     topNodes = diagram.getPossibleTopNodes();
     if(topNodes.isEmpty()) {
         addError(new Error(Error.ErrorType.NO_TOPNODE, null));
         return false;
     }

     // Add the top nodes to the queue.
     for(ComponentModel cm: topNodes) {
         queue.add(new NodeStackElement(cm));
     }

     // Now, traverse the diagram.
     while(!queue.isEmpty()) {
         // Remove the node from the queue
         NodeStackElement thisNode = queue.remove();
         // .. and add it to the finished queue
         finishedNodes.add(thisNode);

         // Also add all the non discovered descendants to the queue

         ArrayList<ConnectorModel> connectors
           = diagram.getLowerConnectors(thisNode.component);

         if(!connectors.isEmpty()) {
            for(ConnectorModel conn: connectors) {
```

```
                    ComponentModel endComponent
                        = diagram.getLowerComponentModel(conn);
                    NodeStackElement newNode
                        = findNSE(endComponent, queue, finishedNodes);
                    if(newNode == null) // Not seen before; add it
                        queue.add(new NodeStackElement(endComponent));
                }
            }

            // Now do the actual check

            if(thisNode.component.isPrimitive() &&
                    !connectors.isEmpty()) {
                addError(new Error(
                    PRIMITIVE_COMPONENT_HAS_OUTGOING_CONNECTORS,
                    thisNode.component));
                ++nErrors;
            }
        }

        return nErrors == 0;

    }
};
```

Listing 6.10: A New Rule

That is all — a new SP Light rule check has been created.

6.3.5 Help, Utils and Event

These packages are small and provide basis functionality.

Help

The help-package contains functionality which lets the user right-click on elements in SP Light og use “Help on <ElementName>”. When clicking this option the computer’s default browser will open the page `http://splight.idi.ntnu.no/index.php/<elementName>`. Difficult aspects in SP Light is in other words only two mouse clicks away. And since the information is in a wiki, the customer can easily modify the explanations according to how users perceive the element.

To enable this feature for a new `JComponent`, setting its popup menu to an instance of the `SPPopupMenu` and its name to `<ElementName>` is all that is required. Listing 6.11 shows how this can be done for a `JButton`, given that you have a reference to an existing `SPPopupMenu` object.

```
public void addHelpToButton(SPPopupMenu menu) {
  JButton button = new JButton("New Diagram");
  button.setComponentPopupMenu(menu);
  button.setName("New Diagram");
}
```

Listing 6.11: A New Button with Help

When converting `<ElementName>` to a URL spaces are changed into underscores, i.e. the URL for the JButton listed above will be `http://splight.idi.ntnu.no/index.php/New_Diagram`.

Utils

This package includes only `Debug.java`. It is a class which allows us to get debug output from the program in an orderly manner. Its main function is to remove the need for `System.out.println` calls, by providing other methods for writing to the console. When delivering the product, these alternative methods can be set to do nothing, easily disabling the debug output.

Event

This package includes the `splightEvent` and its subclasses. These events are used when the data models change to notify the GUI.

6.4 Mapping between Code and Requirements

The requirements specification states which requirements to implement and which priority they are given. The system test, see 7.6, is divided into sections according to what is being tested and can be seen as a mapping between code and requirements. This section will explain changed requirements and requirements which are implemented in another way than described in design.

6.4.1 Priority High

Requirement F-6 states:

- **H F–6** It shall be possible to store different versions of a diagram.

The purpose of this requirement is to allow the user to make multiple versions of the same diagram and make them appear as one diagram to its surroundings. The user would be able to switch between the versions by right-clicking a sub-diagram/sub-system component and selecting the version. Unfortunately the feature did not end up working as we had anticipated and we had to remove it due to lack of time to make it working.

Moving Connectors

The requirements F–90, F–97 and F–98 in the requirements specification state:

- **H F–90** When detached from a component, a connector shall retain all data.
- **H F–97** When reattaching a nonempty connector, the tool shall check that the attachment is valid.
- **H F–98** When a connector is moved to a component with different properties, the user shall be able to choose “abort” or “continue as new connector”

These requirements state what should happen when moving connectors between components. This is implemented as follows: When moving between different connection points on the same component, the matrices remain unchanged. When moving the connector to another component and if the matrices are non-empty, the user is asked to choose between deleting the matrices or aborting the move.

6.4.2 Priority Medium

Drag-and-Drop

Requirement F-92 states:

- **M F–92** It shall be possible to perform drag-and-drop of the symbols from the sidebar menu.

This has been implemented slightly different from the original requirement. Instead of drag-and-drop, we ended up with a select-and-click solution. The user selects a tool from the toolbar using a single click followed by a click in the diagram panel to place the new component/connector. This should solve the problem just as fine, and it is similar to how other programs handle toolbar/-drawing panel interaction. We prioritized making other requirements work instead of making this one comply exactly with the requirement specification.

Undo/Redo

Requirement F-11 states:

- **M F–11** It shall be possible to undo (and redo) at least 20 user operations.

This turned out to be more complicated to implement than we first anticipated due to dependencies between diagrams. At first we wanted to implement undo/redo at the diagram level. This would let the user perform undo/redo on a diagram independent of any changes to other diagrams. The complications occurred when you have a sub-diagram or sub-system inside a diagram with a connector going from the top-level diagram to a component in the sub-diagram or the sub-system. Removing a component from the sub-diagram would also make the connector disappear. If the user then changes the component in the top-level diagram that this connector was originally attached to, then undoing the lower-lower diagram would be complicated as the user might expect to have the connector undo'ed as well.

An alternative undo/redo solution would be to only offer it on a project level. Unfortunately we had to prioritize the more important core features.

Referencing cells in matrices

Requirements F–71 through F–73 and F–114 state:

- **M F–71** Cells in CSMs or IDT matrices shall be addressable.
- **M F–72** If a cell has been given a name, it shall be possible to refer to it in other matrices in the project.
- **M F–73** Cells in CSMs or IDT matrices shall be able to have a textual description of the cell content. The user shall be able to choose whether or not to provide a description of the cell.
- **H F–114** In the matrix cells, it shall be possible to enter constants or functions, with possible external references (i.e. references to functions in the function library or references to elements in some CSM or IDT matrix).

For usability reasons, these requirements have been implemented in a slightly different manner than what was originally intended.

Let us first look at F–71 and F–114. In the program, it is not possible in a cell to refer to another cell in another IDT or CSM matrix. This is not difficult to implement, but we have chosen another solution for achieving the same goal in a more user friendly manner. Let us look at what the user would have to do to reference a CSM or IDT matrix tied to another connector than the connector on which the user is currently working. To uniquely specify an element in another matrix, the user would have to specify the two components which are linked by the connector which has a matrix to which the user wants to refer.

But suppose that the two components have several different connectors between each other. The user would then have to choose which connector he or she wants to refer to, in addition to specifying the correct row and column of the matrix of the connector. It gets even more complicated if the user wants to refer to a matrix in a subdiagram. This complexity increases the chance that the user might make an error. Also, the user might delete rows or columns which have references to them. It would contradict the purpose of SP Light — to provide a tool which does not put any particular restrictions on what the user can do — to disallow the user to do such operations. Another problem is that if the user changes a matrix cell which has references to it, he or she might not know that this is the case.

Instead, we opted to do this in a simpler manner. There is really no need to be able to be able to refer to elements of CSM and IDT matrices since their contents are merely constants. So if a user wants to use a particular value in two matrix cells, it is a *lot* easier to just add a new function to the function library and use that function inside the two matrix cells. This more user friendly since the user can access the function library easily when entering matrix cell contents. The user can set this function to have a name and description indicating the matrices in which it is used.

Regarding requirements F–72 and F–73, we have also decided to do this via the function library. We feel there is little need to give cells names and descriptions. There are three primary reasons. First, the purpose of CSM or IDT cells is pre-defined: It is the relationship between the service or logical storage in the column header and the service or logical storage in the row header. The rows and columns already have names. Second, if a value in a CSM or IDT matrix is important enough to deserve a description, it should instead be added to the function library and instead the matrix cell should contain a reference to it. In the function library, the user can enter a name and detailed description. Third, the entire description would disappear if the user (accidentally or not) removed a service from a component. Again, using the function library solves these problems.

We summarize this section by stating that in our view, these requirements only add complexity and reduce usability. The function library solves these problems and is a better solution. We admit that these requirements in the requirements specification should have been more clearly written and more thought through regarding these points.

Color Coding in Rule Checking

Requirement F–39 and F–40 states:

- **M F–39** If a rule check fails for a connector, it shall be colored red.
- **M F–40** It shall be possible to enable or disable color coding.

These requirements have been implemented. However, at the time these requirements were written, we did not know exactly which rules we were going to implement. In light of the rules in described in Section 6.3.4, we quickly decided that it would be more natural to color code the *components*, and not the *connectors*. Hence in the program, *components* at which a rule violation is detected are colored red.

Regarding requirement F–40, we chose not to let the user disable or enable color coding when checking the rules, because new users could potentially be confused as to what the color coding implied. Advanced users would also prefer to always enable color coding to mark the errors in their diagrams. Therefore, the rule check will always color code the components having errors. But the user may clear the color coding by right-clicking on the diagram and pressing "Reset color coding".

Progress bars

Requirement F–75 in the requirements specification states:

- **M F–75** Every process expected to take more than ten seconds shall have a progress bar.

We have not found an operation in SP Light that takes more than ten seconds. The most CPU-intensive parts of the program, namely storage, calculations and rule checking, are all blazingly quick even on relatively old machines. Just creating the progress bar might actually take more CPU time than doing these tasks! Therefore, progress bars are not used.

6.4.3 Priority Low

Requirements with low priority has only been implemented when they were conveniently implemented along with other requirements. Almost all of these has been implemented and some is described below.

- **L F–14** Every diagram shall have a unique identifier.

This is implemented by enforcing unique names. If the user provide duplicate names, the program adds a postfix to make the name unique. This is further discussed in the description of storage, see 6.3.2.

Documentation

The wiki is structured according to the following requirements:

- **L F–29** User documentation shall include information on the SP language.
- **L F–30** User documentation shall include information on how to use SP Light.
- **L F–31** User documentation shall include tips about how to create good SP diagrams.

These requirements are implemented through the wiki at [30W]. We believe this is a very good solution for the following reasons:

1. The wiki can act as a source from which everyone — not only users of SP Light — can retrieve information about the SP modeling language. Had we made the documentation only available from the SP Light program, this would not be the case. But the customer can also restrict access to the wiki, if he wishes to do so.
2. The wiki takes care of keeping track of help documentation revisions. It is very simple to see the differences between two document revisions in the wiki, and to find out who wrote what.
3. The wiki is easy to update. If we had made an internal help system in SP Light, it would likely be a lot harder to update than the wiki, which is simply a normal web page accessible from anywhere. This also implies that the customer need not release a new version of SP Light to include new help information; users can simply visit the wiki and always have the latest documentation available.

The wiki is not complete, and since we are not experts on the SP language and how it should be developed, used and presented, the customer will have to put an effort into expanding the wiki. The customer has already started creating pages in the wiki. We have also contributed with a great deal of information in the wiki. This includes both developer howtos (e.g. how to add new rules to the rule checking system as described in Section 6.3.4) and user howtos (e.g. how to use the function library and mathematical expressions in SP Light). We have focused on adding information required to understand SP Light as a developer, or to understand concepts in SP Light which are not intuitive or obvious. We refer to the wiki ([30W]) for further details.

Extending the Parser

SP Light contains a parser for mathematical expressions. It was described in Section 6.3.3. One of the low priority requirements, F–45, was not implemented. We will explain why it is not difficult to extend the program with F–45, and also take a look at how we have solved F–46.

- **L F–45** The parser shall support the guard operator.
- **L F–46** Developer users shall be able to expand the parser with new operators.

The guard operator was described in Section 6.3.3.

This has not been implemented in the parser, but the parser has been designed with extension in mind. It is not a great deal of work to add the guard operator to the parser, and the EBNF grammar for the parser syntax described in Section 6.3.3 already supports the guard operator and boolean expressions.

The reason we are confident that extending the parser with new operators (whether or not they are in the suggested grammar) is easy is due to the structure of the recursive descent parser. Each nonterminal symbol in the grammar has its own method. Consequently, adding support for a new operator is done in four steps:

1. Modify the grammar — this has already been done for the guard operator, boolean expressions (with logical AND, OR and NOT operators) and comparison operators (greater than, greater than or equal, less than, less than or equal).
2. Make the tokenizer in `ParserTokenizer.java` recognize the new operator symbols. This is just adding a small amount of code to a case statement.
3. Make a new method for each new nonterminal symbol in the grammar in `Parser.java`.
4. Modify the existing methods for the other nonterminal symbols to handle the new rules introduced in the grammar involving the new operator.

Typically, the new methods will just be small modifications of the already implemented methods for parsing nonterminal and terminal symbols. The `Parser` class also contains some general support methods for handling left associative operators (which is one of the difficulties when dealing with recursive descent parsers). Thus, extending the parser is a relatively easy task. We refer to the source code and Section 6.3.3 for more detailed information.

Extending Other Functionality

We finish the mapping requirements section by considering the two requirements:

- **L F–15** It shall be possible to set the zoom at a given percentage.
- **L F–46** It shall be possible to draw a connector by drawing connected straight line segments (a "polyline") rather than a single straight line.

Requirement F–15 has been implemented in the following way: Zoom is available using the zoom buttons on the main toolbar. The user may zoom in increments of 10user can not set the zoom to be, for instance, 78restriction to any type of user.

Requirement F–46, we unfortunately did not have the time to implement this. Currently, all connectors go in straight lines. However, SP diagrams directed acyclic graphs and therefore it should be very easy to draw the diagrams with little or no clutter between the connectors.

6.5 Installation and User Manual

In this section we will describe an installation manual for SP Light. We will also describe how user documentation is available.

6.5.1 Installation Manual

To run SP Light you need to have Java 1.5 Runtime Environment installed and a computer running Windows (2k or XP) or Linux (2.6 or newer). The program is distributed as a self-extracting `rar` archive for Windows and a `tar.bz2` archive for Linux. The archives contain some `jar`-files and startup script (`start.bat` for Windows or `start.sh` for Linux). Running `start.bat` or `start.sh` starts the program. No further installation is required.

Run On Windows

We will now describe how to run SP Light on Windows.

1. Get SP Light Windows package (`splight.exe`) from the attached CD-ROM or download it from `http://splight.idi.ntnu.no/downloads`.
2. Run the file (an archive will be extracted)
3. Go to the directory which the archived was extracted to and run `start.bat`. (Default path is c:\splight.)

Run On Linux

We will now describe how to run SP Light on Linux.

1. Get SP Light Linux package (`splight.tar.bz2`) from the attached CD-ROM or download it from `http://splight.idi.ntnu.no/downloads`.
2. Open a Linux shell or terminal, change the directory (`cd`) to the directory where the file is located, and extract the file with the command `tar -jxvf splight.tar.bz2`.
3. A directory named `splight` will now be created. `cd` into the directory, and type `./splight` to start SP Light.

6.5.2 User Manual

We have chosen to use a wiki (running the Mediawiki [29W] software package, hosted by IDI) to document the program. The reasons for using a wiki were described in Section 6.4.3. It can be found at `http://splight.idi.ntnu.no` and consists of three parts: Structure and Performance, SP Light and howtos.

The first part has been used during the project to communicate difficult aspects of SP within the group and with the customer. Further extensions will provide an introduction to SP and information about common problems for new users.

The second part is about the program itself, SP Light. It is divided into modeling, calculations and rule checking, since these are the three main areas of the program. The different elements used, as components and connectors, and their use is the most important content here.

The last part is howtos for developer users and howtos for regular users. These are quick guides on how to perform certain procedures, or where we think others may benefit from a guide. For example the "import diagram"-function is explained.

The customer wants complete control over any publication, so editing the wiki is restricted. See 8.5.1 for the plan for further development.

6.6 Summary

The implementation phase was short and intensive. SP Light took shape and satisfies nearly all of the requirements. Since time was scarce little time was used to fix bugs and improve the GUI. SP Light is therefore precisely that, a lightweight framework. This is better than the goals set for Customer Driven Project, which states that a *prototype* shall be developed. When taken into account that the project was originally planned for 6-7 person, we believe this is satisfactory.

Chapter 7

Testing

7.1 Introduction

This document includes how to test, when to test, what to be tested, and the results of the tests.

The first thing we tested was usability. Since the program is planned to be used by students not knowing very much about SP, it is important to have help functions available, and to have a user interface that is easy to use. It was important to test the user interface as early as possible, since modifying it later would be much more difficult. We started by making a graphical user interface prototype, just to have something to test. This draft did not contain much data, and the code was not tested with unit tests. After the usability test, we rewrote the prototype and used checklists for testing each class. We also added much more functionality and all modules were checked with checklists to make sure they communicated with other modules in the right way. When most of the code was written, we used a system test to check what needed to be fixed, and the customer did an acceptance test.

Below is a summary of all test types. The main test plan in Table 7.1 shows an overview of the test process.

- Unit test: Testing the code of each method and class while programming.
- Module test: Testing of interfaces between different code packages while coding.
- Usability test: Testing to make sure the graphical user interface is understandable.
- System test: Testing to see if the tool does what it is expected to do.
- Acceptance test: Testing to check whether the customer is satisfied with the product or not.

Activity	**Test control**	**Run by**	**Time**	**Responsible**	**Report to**
Usability test	Detailed test specification	Students from another kpro-group	07.11.2005	Magne	Erik
Unit test	Checklists	Developers	During implementation	Each developer	Magne
Module test	Checklists	Developers	During implementation	Magne	Erik
System test	Detailed test specification	Developers	20.11.2005	Erik	Magne
Acceptance test	Detailed test specification	Customer	21.11.2005	Erik	Magne

Table 7.1: Main test plan

7.2 Test Requirements

This section will focus on test requirements. Tests should be based on requirements, not the finished product. To avoid making tests based on a finished product, every test requirement should be made before designing the unit being tested. We have used a simplified version of the V-model [13W] to make sure the test development is performed in the right order.

Figure 7.1 shows the V-model. The development process starts with requirements from the customer, moves down to unit implementation and then up to acceptance test, like a V. When moving downwards, test specifications should be made for the horizontal corresponding upwards activities.

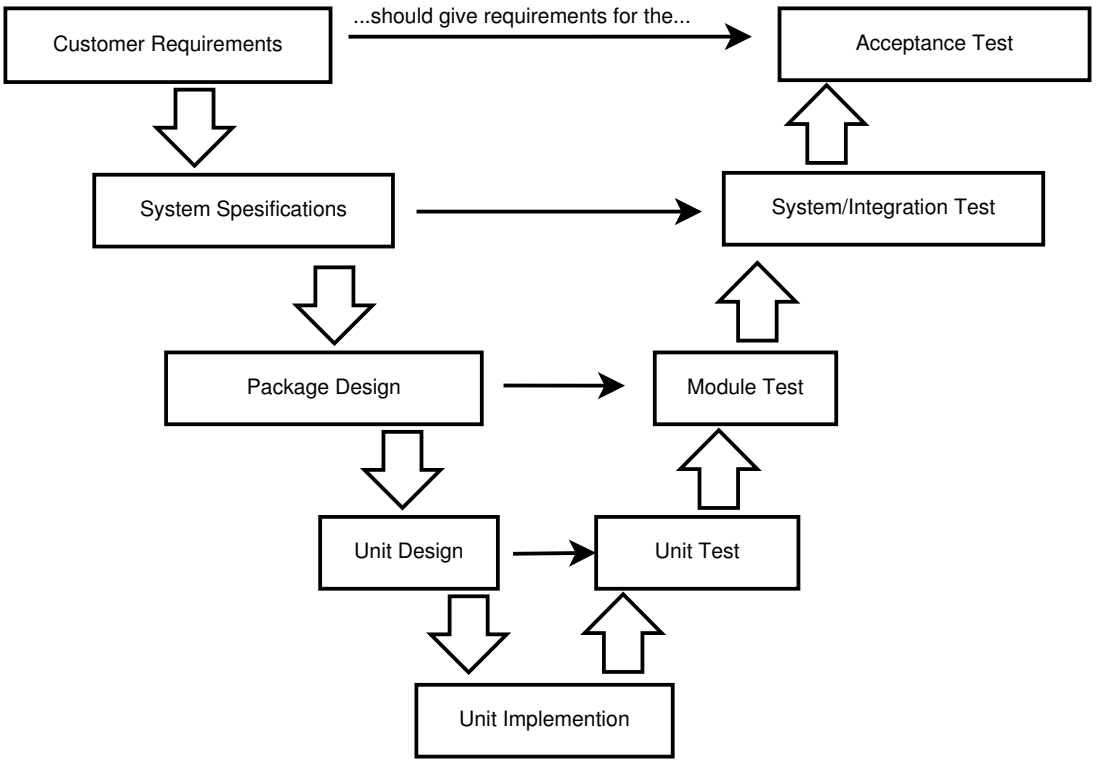


Figure 7.1: The V-model.

The usability test is not a part of the V-model, and was done before unit testing, after making a small prototype of the program. Below is a list of testing requirements for all test types except the usability test. Specific test procedures for each test level are described in the beginning of the accompanying test sections.

- All test plans shall be made before implementation.
- All tests shall be made so that the test result is either “OK” or “NOT OK”.
- If checklists are used, all errors shall be fixed before setting it OK.
- If detailed test specification is used, failure reports shall be made for every “NOT OK” test result.

- A failure report is closed if the error is proved corrected in a new identical test, or if the customer approves the functionality loss.
- Every failure report from one test level in the V-model shall be closed before performing tests on the next level.

7.3 Unit Test

The unit testing is done during implementation, and consists of checklists that the programmers use to make sure each java class is okay. Checklists were made for code documentation, data access, error prevention and graphical representation. All checklists were made in the beginning of the implementation phase.

Checklist for code documentation is shown in Table 7.2.

Check number	Check	Result
UT–1	All variable names are representative for their content	
UT–2	All method names are representative for the function they perform	
UT–3	All non-trivial methods are commented	
UT–4	All public methods are commented in Javadoc	

Table 7.2: Code documentation

Checklist for data access is shown in Table 7.3.

Check number	Check	Result
UT–5	All access modifiers are as restrictive as possible	
UT–6	Set- and get-methods for every data object which shall be available for other classes.	
UT–7	GUI elements are listening to the correct data models	

Table 7.3: Data access

Checklist for error prevention is shown in Table 7.4.

Check number	Check	Result
UT–8	No infinite loops	
UT–9	All loop extremities have been tested	
UT–10	All exceptions handled	

Table 7.4: Error prevention

Checklist for graphical representation is shown in Table 7.5.

Check number	Check	Result
UT–11	All dialog boxes have cancel buttons, except purely informative ones	
UT–12	All actions which take more than 10 seconds have progress bars	
UT–13	No output to System.out	

Table 7.5: Graphical representation

7.3.1 Results

All classes were checked, and code was corrected when errors were found. Progress bars were not implemented for any process, since none of the processes took more than ten seconds, even for very complicated diagrams.

7.4 Module Test

Module testing is done during implementation, and consists of a checklist that the programmers use to make sure each java class is okay, and to make sure the module communicates with other modules the way it is supposed to. The checklist was made in the design phase.

The checklist is shown in Table 7.6.

Check number	Check	Result
MT–1	The purpose of every java class is described in javadoc.	
MT–2	All class names are representative for the function they perform.	
MT–3	No GUI functions in core modules.	
MT–4	No global variables edited.	
MT–5	Interface to rest of the system is according to the design specification.	

Table 7.6: Module checklist

7.4.1 Results and comments

We used XMLBeans [19W] for XML storage. The classes generated by XMLBeans are not commented, since they were auto-generated as a jar-file, and we never needed to edit any code in them. For all other modules, the above checklist has been used. When variables were needed globally, they were included as private variables in the Controller class, made set- and get-methods, and gave the needing object a reference to the controller.

Module tests was performed at a very early stadium. After integrating GUI and storage, the checklist was used primarily after refactoring, to check that no violations had been introduced.

7.5 Usability Test

Usability testing is done to find errors in how the graphical user interface is presented. It is difficult for the developer to detect errors while creating the interface, since the developer gets very used to things and do not necessarily detect that graphical objects do not work like a normal user would expect. The test was written when the requirement specification was ready and tested as soon as we had a testable prototype showing the graphical user interface. Many core functions were not implemented at this stage, but this was not essential, since we could fake storage and other complicated actions and just tell the test persons what the system would do.

The tests took place in IDI's usability lab. The program was tested on another group taking the Customer Driven Project course. They were all students of Computer Science, 4th grade, so their technical knowledge was similar to what we can expect from a real user. Our group provided a test leader and two observers, and we used the video equipment in the lab to record video of the screen with mouse movements and audio of what they said. The test subject could opt not to be filmed.

We performed the test using the following guideline:

1. Introduce yourself and the testing team.
2. Describe the purpose with the test.
3. Explain that it is possible to abort at all times.
4. Explain the equipment (camera, computer, etc).
5. Teach the test persons to think aloud.
6. Explain that no help can be given during the test.
7. Describe the task and introduce the product.
8. Ask if the test person has any questions, and run the test.
9. After the test is finish, ask for comments on the program.
10. Use the results to improve the program.

7.5.1 Success Criteria

Since the test was carried out before much of the program was made, the main purpose of the test was to find improvements that we could implement. Success criteria were based on how well the test users managed to navigate in the program and make diagrams. 6 persons tested the program, and if two or more of them had problems with a function, the function should be improved. The test focused on navigation, drawing and help functions. Before the test, we explained basic concepts in SP to the test persons.

7.5.2 Tasks to Perform

The tasks the test persons tested are listed below. This list was also handed out to the test persons, and can be found in Norwegian in Appendix E.1, with a short introduction used in the actual test.

- create a project
- create a new diagram
- create two components and connect them with a connector
- switch the type of the connector
- set the extent of a component
- save the project
- close SP Light and load the previously saved project
- find two ways to access the help documentation
- read the help documentation for the import function
- import a diagram to the existing project
- find the version of the program

7.5.3 Results

The test unveiled some errors. Table 7.7 shows a summary of the errors we found after studying the recording, and possible solutions to them.

Problem	**Solution**
When connecting two components, many users tried to attach the connector to the corner points of a component or to the circle showing number of services, instead of connecting it to one of the connection points.	Make corner points smaller. Move the service circle some pixels away from the components. Make connection points green when they are legal to connect to, otherwise red.
When deleting a component, some test persons right-clicked on the connector and expected to find a "Delete"-button somewhere instead of just pressing the delete key on the keyboard.	Adding a delete item on the right-click menu.
Some test persons wanted to use drag-and-drop to draw components, and did not understand that they had to first click in the toolbox to select tool and then click on the drawing area to make the component.	Implement drag-and-drop or make the mouse cursor change to a drawing symbol when a tool is selected.

Table 7.7: Usability test results

7.6 System Test

The system test was made when the requirements specification was done. A system test is performed to check if all these requirements are implemented.

7.6.1 Test procedures

- Test specifications shall be based on the requirements specification.
- The system test is performed by the developer, and developers watch and take notes of the results.
- Failure reports from the system test shall be closed before acceptance test.
- The test specifications are listed below. We have tried to combine multiple requirements into each test.

7.6.2 Detailed Test Specifications and Results

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Drawing	
Requirements tested	F–1, F–2, F–4 through F–7, F–9, F–10, F–12	
ST–1		
Make a new Project with an empty diagram and insert three normal components, one multi-component, one sub-system and one sub-diagram by dragging them from the drawing toolbox.	The drawing area shall have scrollbars and a grid. It shall be possible to insert all components without receiving any error message. The components are given standard names, possible to change.	OK (See 7.6.3)
ST–2		
Connect all the components with all three types of connectors.	It shall be possible to insert connectors without receiving any error message.	OK
ST–3		
Move a component.	Connectors shall remain attached when moving the component.	OK
ST–4		
Double-click a normal component and add two operations. Repeat for the other normal components.	The operation number on each component shall be updated.	OK
ST–5		
Double-click the memory connector between the two normal components and add a function to both matrices.	The operation number on each component shall be updated.	OK
ST–6		
Move one end of the memory connection from the normal component to the multi-component.	Connectors shall not attach to the multi-component. Error message where the user can select whether to abort or continue as a new connector.	OK

Table 7.8: Drawing test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Rule checking	
Requirements tested	F–33 through F–41	
ST–7		
Create an SP diagram with a non-primitive component having one primitive communication subcomponent. Perform a rule check with the test for processing and memory subcomponents selected. Disable color coding.	Rule checking reports that the component is missing a processing *and* a memory subcomponent. The rule check window has a clickable link which, upon clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK (See 7.6.3)
ST–8		
Repeat previous test, but enable color coding.	As previous test, but in addition the components missing the subcomponents is highlighted with the color indicated in the error log window.	OK
ST–9		
Repeat previous test, but do not select the test for processing and memory subcomponents.	Rule checking completes with no error. Component does not change color.	OK
ST–10		
Create an SP diagram with a non-primitive component having primitive processing and memory subcomponents. Perform a rule check with the test for processing and memory subcomponents selected.	The processing and memory subcomponent test succeeds. Component does not change color.	OK
ST–11		

Table 7.9: Rule check test *(continued on next page)*

Test case	Expected result	Result
Create an SP diagram with two non-primitive components, and add respectively distributed processing and distributed persistent storage subcomponents to the components. Let at least one of the subcomponents of the distributed processing component have no primitive processing subcomponent and let also at least one of the subcomponents of the distributed storage component have no primitive storage subcomponent. Perform a rule check with the checks for distributed processing and distributed persistent storage checks enabled. Disable color coding.	Rule checking reports that a distributed processing component is missing a processing subcomponent and that a distributed persistent storage unit is missing a memory subcomponent. The rule check window has a clickable link which, upon clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK (See 7.6.3)
ST–12		
Repeat previous test, but enable color coding.	As previous test, but in addition, the components change color to red.	OK
ST–13		
Repeat previous test, but add the missing processing and memory subcomponents.	Rule checking completes with no errors.	OK
ST–14		
Repeat ST–7, but de-select the tests for distributed processing and memory subcomponents in the rule check.	Rule checking completes with no errors.	OK

Table 7.9: Rule check test *(continued on next page)*

Test case	Expected result	Result
ST–15		
Create an SP diagram with two components having, respectively, distributed persistent storage and distributed processing, but no communication subcomponents. Perform a rule check with the check for communication subcomponents with distributed memory and/or distributed processing enabled. Disable color coding.	Rule checking reports that a communication subcomponent is missing. The rule check window has a clickable link which, upon clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK (See 7.6.3)
ST–16		
Repeat previous test, but enable color coding.	As previously, but in addition, the components having distributed persistent memory and distributed processing but missing communication subcomponents change color to red.	OK
ST–17		
Repeat previous test, but create a communication subcomponent below the component having distributed processing.	Rule checking completes with same error as before, but only highlights the component having distributed persistent storage. The rule check window has a clickable link which, when clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK
ST–18		
Repeat previous test, but create a communication subcomponent also below the component having distribtued persistent storage.	Rule checking completes with no errors.	OK
ST–19		

Table 7.9: Rule check test *(continued on next page)*

Test case	Expected result	Result
Repeat previous test, but create another communication subcomponent below the components having distributed persistent storage and distributed processing.Enable color coding.	Rule checking completes with an error stating that only one communication component is allowed below the components. The distributed processing and persistent storage components change color to red. The rule check window has a clickable link which, when clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK
ST–20		
Repeat test ST–11, but do not enable the test for communication subcomponents.	Rule checking completes with no errors.	OK
ST–21		
Create a communication component. Below, create three other components which are not communication components linked under each other. Below these three components, create a communication component. Perform a rule check with the check for ancestor communication components enabled. Disable color coding.	Rule checking completes with error stating that the lowermost communication component has an ancestor communication component. The rule check window has a clickable link which, upon clicked, opens the local web browser and directs it to a page with more detailed information about the failure.	OK (See 7.6.3)
ST–22		
Repeat previous test, but enable color coding.	As in the previous test, but in addition, both communication components are highlighted by the color indicated by the rule check error log in the diagram.	OK
ST–23		
Repeat previous test, but do not create the lowermost communication component.	Rule checking completes with no errors.	OK
ST–24		

Table 7.9: Rule check test *(continued on next page)*

Test case	Expected result	Result
Repeat test ST–17, but do not enable the test for ancestor communication components.	Rule checking completes with no errors.	OK
ST–25		
Create an arbitrary SP diagram. Perform a rule check with all the tests enabled. When the rule check window appears, minimize it and attempt to do an operation (e.g.. add a new component) in the diagram.	Rule checking completes, reporting possible errors (what errors it reports is not important in this test). The rule checking window shall appear and be minimizable. The diagram shall operate as normal even though the error log window is open or minimized. The error log window shall display the name of the diagram, and current date and time.	OK

Table 7.9: Rule check test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Help and User Documentation	
Requirements tested	F–3, F–18 through F–29	
ST–26		
Click the "Help" menu, and click "Documentation".	The local web browser opens up and a web page with SP Light documentation is displayed.	OK
ST–27		
Click the "Help" menu, and click "About".	An About box opens up. The box contains (at least) the program name, version number, developer names and contact information, a clickable link to the SP Light home page, license agreement and copyright information.	OK
ST–28		
Create an SP diagram containing all possible components and connectors. Successively right-click each component and connector in the diagram.	The pop-up menu has an item "Get help...". Choosing this item will open up the local web browser and direct it to a page containing more information about the component or connector in question.	OK
ST–29		
Repeat previous test, but disconnect the local computer from the Internet first.	Clicking "Get help..." opens a dialog box reporting that the web page is inaccessible, and displays the URL for the help information.	OK
ST–30		
Hover the mouse pointer over each icon (with no textual information) in SP Light for at least 5 seconds.	After a few seconds of hovering, a tool tip with a textual description of the action of the icon appears.	OK
ST–31		
Try accessing the help documentation, and try to modify the help documentation without logging in.	The help system reports that you do not have access to modify the help documentation.	OK
ST–32		

Table 7.10: Help and User Documentation Test *(continued on next page)*

Test case	Expected result	Result
Attempt to change the help documentation for a specific component or connector in the system.	Changing the documentation requires at most changing an URL in the source code of SP Light, or nothing at all.	OK
ST–33		
Access the help system from the “Help” menu.	Whenever terms from SP Light, such as “connector”, “component”, “rule check”, “subsystem”, are used, these terms shall be clickable and linked to further information about these terms.	OK
ST–34		
Access the help system from the “Help” menu. Attempt to search for at least 5 terms used in SP Light (see the results of the previous test for examples).	Search is successful. The help system provides links describing and providing help on the terms searched for.	OK

Table 7.10: Help and User Documentation Test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Parser test	
Requirements tested	F–42 through F–45, and F–62, and partially F–68, F–69, F–70, F–74	
ST–35		
Create an SP diagram with two components and a memory link between them. Open up the Calculations window by right-clicking the link. Add two storages to each component. In the four matrix cells, enter the following expressions: $2 + 3 * 2$, $5 - 6 - 7$ (in the first row), $5 - (6 - 7)$, and $2 * 3\text{^}(2 + 9.2) + 3/4.0 + 3\text{^}9$ (in the second row). Click OK. Then specify the data load of the uppermost component to 1. Then click the Perform Calculations button.	No syntax error is returned. The resulting workload on the lowermost component is calculated by SP Light to be $(14, 461030.7476)$.	OK
ST–36		
Repeat the previous test, but instead of writing the last element in the test directly, add a function to the function library $f(x)$ with content $3/4.0$ and a function $g(x)$ with content $3\text{^}(2 + 9.2)$, and write the element as $2 * g(x) + f(x) + 3\text{^}9$.	As in the previous test.	NOT OK (See 7.6.3)
ST–37		

Table 7.11: Parser test *(continued on next page)*

Test case	Expected result	Result
Repeat the previous test, but add an extra parenthesis to the first element so it becomes $2 + 3 * 2)$.	After typing the element in, an error is reported saying that the syntax is invalid and also possibly that the reason for the error is the superfluous parenthesis. During calculations the same error is reported.	NOT OK (See 7.6.3)
ST–38		
Repeat the previous test, but change the first element so it becomes $/3 * 2$.	After typing the element in, an error is reported saying that the syntax is invalid and also possibly that the reason for the error is that the / operator is missing an operand. During calculations the same error is reported.	NOT OK (See 7.6.3)
ST–39		
Repeat the previous test, but change the first element so it becomes $7/0$.	During the calculations an error message appears saying that it could not complete due to a division-by-zero error in the specific element.	NOT OK (See 7.6.3)

Table 7.11: Parser test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Genaral GUI	
Requirements tested	F–11, F–13, F–14, F–74 through F–83	
ST–40		
Start SPLight.	Icons shall look like those in standard MS Windows software. The main menu shall have File, Edit and Help as first level menus.	OK
ST–41		
Make a new project and insert a new diagram and a new component. Try all the functions in the Edit menu.	It shall be possible to copy, paste and cut diagram objects. It shall be possible to undo 20 operations. It shall be possible to zoom in and out on the diagram.	NOT OK (See 7.6.3)
ST–42		
Make a diagram with more than ten connected components and add advanced functions to the complexity matrices.	Mouse cursor shall turn into an hourglass while computing. A progress bar shall show the progress.	NOT OK (See 7.6.3)
ST–43		
Try all menu functions.	Every function displaying a dialog box shall be possible to abort by pressing "Cancel".	OK
ST–44		
Try all functions in the main window.	Every function displaying a dialog box shall be possible to abort by pressing "Cancel".	OK
ST–45		
Try all functions in the component window.	Every function displaying a dialog box shall be possible to abort by pressing "Cancel".	OK
ST–46		
Try all functions in the main connector window.	Every function displaying a dialog box shall be possible to abort by pressing "Cancel".	OK
ST–47		

Table 7.12: Rule check test *(continued on next page)*

Test case	Expected result	Result
Start SPLight, add a new project, a new diagram, a new component and add a operation to the component.	The tree structure shall be updated and show the project, the diagram, the component and the operation.	OK
ST–48		
Make two diagram in a project. Close them, and reopen them by double-clicking them in the tree structure.	The diagram panel shall be tabbed, and each diagram shall have its own tab when it is open. Double-clicking in the tree structure shall open the diagram in a new tab, or focus the tab if the diagram is already opened.	NOT OK (See 7.6.3)
ST–49		
Click on the Rule Checking button.	A window for rule checking shall appear.	OK

Table 7.12: Rule check test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Storage	
Requirements tested	F–1, F–2, F–4 through F–7, F–9, F–10, F–12, NF–12	
ST–50		
Start SPLight, make a new project and add two new diagrams with some components and connectors with data. Save the project. Restart SPLight and open the project again.	The project file is saved in XML format, and all data is available when reopening the project.	OK
ST–51		
Make a diagram with some connected components. Select Print from the File menu.	A printing dialog box shall show. When pressing ok, the diagram shall be printed if the printer is ok.	OK
ST–52		
Make a new project and select "Import Diagram from the File menu. Select another project in the dialog box and select a diagram from it.	The diagram shall be imported and appear in the tree structure.	OK
ST–53		
Make a legal diagram. Press the "Calculate" button.	Performance data from each level of the model is calculated and presented in a tab separated list, possible to copy and paste into Excel.	OK
ST–54		
Let the diagram remain open for 11 minutes.	After the 11 minutes, a backup file shall appear in the directory where the project file was previously saved.	OK

Table 7.13: Rule check test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	SP components and connectors test	
Requirements tested	F–48 through F–58, F–60, F–61, F–63, F–64	
ST–55		
Create an SP diagram with two components. Add a memory and processing link between the two components. Give each component the names "My Component 1" and "My Component 2".	Components and links are created, and components are displayed with correct names in the diagram.	OK
ST–56		
Extend the model of the previous test by adding four services and four logical storages to each component. Give each of the 16 services and logical storages distinct, arbitrary names.	Naming the services and logical storages completes successfully.	NOT OK (See 7.6.3)
ST–57		
Repeat previous test, but give two services and two logical storages the same name.	An error is reported, stating that the names conflict.	NOT OK (See 7.6.3)
ST–58		
Double-click the memory connector between the components in the diagram of the previous test. Close the window and perform calculations.	A window with CSM and IDT matrices opens up. The rows of the IDT matrix are the logical storages of the uppermost component of the link, and the columns are the logical storages of the lowermost component of the link. Similarly, for CSMs, rows are the services of the uppermost component of the link, and columns are the services of the lowermost component of the link. Both the CSM and IDTs are filled with empty cells with no name or description. All calculations evaluate to zero.	NOT OK (See 7.6.3)
ST–59		

Table 7.14: SP components and connector test *(continued on next page)*

Test case	Expected result	Result
Double-click a component from the diagram of the previous test. Assign an extent and a data capacity (limit) to the component.	Data capacity (limit) was assigned successfully.	OK
ST–60		
Add a new sub-diagram to the diagram of the previous test. Let the sub-diagram have three components which are hierarcically at the same level. Return to the original diagram, and add a link from the uppermost component in that diagram to the sub-diagram. Then return to the sub-diagram and add a connector from the sub-diagram to the lowermost component in the original diagram.	Connecting succeeds, and when adding the connector from the original diagram to the sub-diagram, a pop-up menu appears where it is possible to choose which component(s) in the sub-diagram the connector connects to. When adding a connector from the sub-diagram, a pop-up menu also appears, where it is possible to choose which component(s) in the original diagram the outgoing connector shall connect to.	OK
ST–61		
Open up the sub-diagram from the previous test, and add 20 components and 20 connectors to it.	Creation works, and the program does not limit the number of components or connectors in the sub-diagram in any way.	OK
ST–62		
Create a new abstraction in the original diagram from the previous test. Attempt to specify extent and limit (by right-clicking the abstraction).	It is not possible to specify extent and limit for the abstraction.	NOT OK (See 7.6.3)
ST–63		

Table 7.14: SP components and connector test *(continued on next page)*

Test case	Expected result	Result
Attempt to specify extent and limit (by right-clicking) for the sub-diagram of the previous test.	Extent and limit are specified successfully.	OK

Table 7.14: SP components and connector test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	Component and connector GUI	
Requirements tested	F–103 through F–110, F–112 through F–115	
ST–64		
In a diagram with two memory-connected normal components, double-click one of the components in the drawing area or in the tree structure.	The component window shall open and contain two tabs; one for operations and one for data structures. It shall be possible to specify speed rate and data size of the component.	NOT OK (See 7.6.3)
ST–65		
Add an operation in the operations tab in the component window for a normal component.	The operation shall be added to the operation list in the operations tab and in the tree structure.	OK
ST–66		
Remove an operation in the operations tab in the component window for a normal component.	The operation shall be removed from the operation list and in the tree structure.	OK
ST–67		
Insert a multi component in a diagram. Double-click on it.	The normal component window shall open, but with an extra textfield to specify how many components the multi component contains.	NOT OK (See 7.6.3)
ST–68		
Insert a sub-diagram component in a diagram. Double-click the sub-diagram component in the diagram or in the tree structure.	The sub-diagram shall open in a new tab.	OK
ST–69		

Table 7.15: Rule check test *(continued on next page)*

Test case	Expected result	Result
Insert a two normal components, add operations to them and connect them with a memory connector. Double-click on the connector.	The connector window shall open, containing two tabs; one for complexity and one for compactness. Both tabs shall contain a matrix, consisting of rows being operations in one of the connected components and columns being operations from the other connected component.	OK
ST–70		
Delete one operation from one of the components in in the previous task.	The matrices shall be updated.	OK
ST–71		
Add one operation to one of the components in the previous task and delete one operation from the other component.	The matrices shall be updated in the connector window.	OK
ST–72		
Insert a function consisting of two constants in the matrix from the previous task.	No error messages.	OK

Table 7.15: Rule check test *(continued from previous page)*

Testing done by	Christian L	
Date	21–11–2005	
Responsible	Erik D	
Test name	SP components and connectors test	
Requirements tested	F–62, F–65, and F–66 through F–74	
ST–73		
Create an SP diagram. Let one component be at the top level, with memory, communication and processing links to two components on the next level. Below these two components, add processing and storage components. Then add four services and four logical storages with arbitrary names to each of the three non-primitive components. Then, double-click each of the three memory connectors and give each element in each CSM and IDT matrix a specific name. Give some (but not all) of the elements a description. Do not give any content.	When double-clicking a connector, a window with CSM and IDT matrices opens up. The elements of the matrices have no name and no description. It is possible to specify the information to be stored in each cell of the matrices. Names and descriptions are assigned to the elements of the matrices successfully.	NOT OK (See 7.6.3)
ST–74		

Table 7.16: Calculations and matrices test *(continued on next page)*

Test case	Expected result	Result
Double-click some memory link in the diagram of the previous test. In the window that appears, attempt to add five mathematical expressions each involving the operators $+$, $-$, $*$, $/$, exponentiation, and grouping using parentheses. Also add two functions whose content refer to the IDT matrix elements $(2,2)$ and $(3,3)$ of the lowermost matrix of the chosen memory link. The names of the elements were assigned in the previous test.	Functions are successfully added to the function library.	NOT OK (See 7.6.3)
ST–75		
In the same diagram as in the previous test, specify work load and data load on the top level component.	Work load and data load are specified successfully.	OK
ST–76		
Now give content to each cell in the CSM and IDT matrices of the previous diagram. In each cell, use the functions and aliases already specified together with some mathematical operations. Then perform calculations.	Calculations complete successfully.	OK

Table 7.16: Calculations and matrices test *(continued from previous page)*

7.6.3 System Test Comments

Any test which did not pass as it was, has been denoted "NOT OK". All these tests are described here.

Failed Tests

These are tests that failed due to bugs in the code or requirements not implemented. Preferably these should have been fixed, but there is unfortunately no time left to do so.

The General GUI Test Undo did not work. This requirement was dropped, as described in Section 6.4.2. Progress bars were also not implemented because they were not needed — see Section 6.4.2. Also, an error was discovered when closing the last open diagram.

Invalid Tests

All tests which are no longer valid are listed here with a comment. Invalid meaning that the test contained errors, or that the concept has been implemented in another way than originally intended.

The Parser Test Two parser tests failed, because the tests was erroneous. The functions must be surrounded by "[" and "]".

SP Components and Connectors Test Some tests failed because input data requirements have changed since the test was written. Extent shall be calculated, and it shall not be possible to give each cell a name. The cell name functionality is implemented with the function library, as described in Section 6.4.2. The program has also changed in the way it deals with connecting connectors into a sub-diagram. It is not possible to connect a connector from a sub-diagram to an outside component from within the sub-diagram. This has to be done from the diagram that contains the sub-diagram.

Component and Connector GUI The requirements on the component and connector have been changed. The multi component diagram element has been implemented in another way than the test presumes, and size is no longer used for what it was originally supposed to in the requirements. Naming matrix cells is implemented through the function library.

Comments to Passed Tests

If something did not go precisely as planned and the tests still were set to "OK", then they are commented below.

The Rule Checking Test As we described in Section 6.4.2, rule checking always enable color coding. There is a context menu choice to clear the color of the components, instead of forcing the user to choose whether or not to enable color coding. This affected the test, but since it is a minor change and since color coding and resetting the colors worked, the test was passed.

The Drawing Test Some tests were written assuming we would use drag and drop. We opted for "click and drag" rather than "drag and drop". However, this was such a minor change and the intention of the test was to test that placing components worked and not the particular interaction style used in the program. For a further discussion see Chapter 6.

7.7 Acceptance Test

The acceptance test was the first one made, and the last one performed. We used the functional requirements from the user when we made the acceptance test. An acceptance test is performed to check if the customer is satisfied with the product, and the customer decides whether to use the product or not based on this test. The functions being tested are very general, and require that both graphical user interface and the underlaying system works as expected.

7.7.1 Test Procedures

- Test specifications shall be based on typical use cases.
- The acceptance test is performed by the customer. Developers watch and take notes of the results.
- Failure reports from the acceptance test shall be closed before deliverance of the product.

7.7.2 Detailed Test Specifications and Results

Test type	**Acceptance test**	**Date:**
Testing done by		
Test responsible		
Test	**Use case**	**Result**
AT–1	Create a new project, make a new diagram and insert some components with connectors between them.	
AT–2	Save and reopen the project.	
AT–3	Print the diagram.	
AT–4	Perform rule check for the diagram.	
AT–5	Add functions to the matrices, store and load functions form the library.	
AT–6	Calculate performance.	
AT–7	Export performance data to Excel.	

Table 7.17: Acceptance test

7.8 Test Summary

We used five different test types; usability test, unit test, module test, system test and integration test. This section summaries each test type.

7.8.1 Usability Test

This test was done with an early prototype to find out how we should design the graphical user interface. The test unveiled some errors that we had to deal with when making the real program.

7.8.2 Unit Tests

Unit tests were done using checklists for each java class. All classes were checked, and all detected errors were removed.

7.8.3 Module Tests

Module tests were done using checklists for each module. These tests were made to unveil communication errors between classes, and all errors were corrected.

7.8.4 System Test

Some of the system tests are denoted "NOT OK". This was mainly due to invalid tests, since the requirements changed after we wrote the first tests. It was a problem that requirements changed after we had implemented them, and this caused some errors when we tried to replace already implemented features. After agreement with the customer, some of the requirements were decided not to implement, since time was limited. Undo/redo is a typical example. Functionality not finished is discussed further in Evaluation, 8.5.1.

7.8.5 Acceptance Tests

The acceptance test was done with the customer. Two customers tested the program with data they knew the results of, and they also tested general functions like adding and deleting diagrams and components. Some small bugs in the graphical user interface were still present in the acceptance test, but these will be fixed before deliverance to the customer. The acceptance test also unveiled a misunderstanding about a new requirement concerning calculation and viewing of temporary matrices. This will be dealt with by going back to the last working algorithm we made, and deliver the current algorithm in another source file, so that the customer can implement the other algorithm or a combination of these later.

Chapter 8

Evaluation

8.1 Introduction

In this document, we will evaluate the project. We will look at the following things:

1. The customer and task. Here we will discuss how we worked with our customer and with the task we were given.
2. Customer Driven Project as a course. This section concerns the course TDT4290 Customer Driven Project. We will evaluate our supervisors, the organization of the course and our experiences with the combination of the project and other courses.
3. The group and the process. This is an evaluation of the group process and our cooperation as a team. We will discuss how we divided tasks and roles between the members of the group, what we gained from the project, the time spent and how the possible risks considered at the start of the project played out.

8.2 The Customer and the Task

In this section, we will take a look at how the collaboration with the customer has been.

8.2.1 Working with the Customer

Our customer was IDI, NTNU and Modicum, Ltd., represented by Professor Peter H. Hughes and Ph.D. student Jakob S. Løvstad. We feel the cooperation went fairly well. Professor Hughes was often away due to traveling, so we quickly became very dependent on Mr. Løvstad. When Mr. Løvstad went away for an entire week, we discovered how much we needed his feedback. We were notified that he was going to be absent some time in advance, and we tried to minimize our need for feedback in the period he was away. Nevertheless, this resulted in some stagnation, since we were about to complete the requirements specification phase when he went away. We needed to get the requirements specification approved before we could put our entire effort into the design phase. Many requirements were uncertain, so it would be a waste of time to design a system which satisfied a requirements specification that may had to be drastically changed. Thus, it was a problem for us that he was unavailable during this period, and some work time was unfortunately wasted.

We would like to mention that Mr. Løvstad, right from the start, offered to join our meetings and work sessions to assist us in resolving any uncertainties we had. We took advantage of his offer a couple of times and we are very grateful that he took the time to help us.

8.2.2 The Task

Compared to other groups we were quite fortunate with our task, since the group had most of the knowledge and skills needed. Unlike what was the case in some other tasks, the customer wanted us to create an actual product, and had very clear ideas about what we should do and to a certain extent how we should do it. We received, for instance, a draft for a requirements specification on the first meeting.

As a consequence of the customer having very clear ideas about what we were going to do, we initially felt that the project definition plan and pilot study were phases that did not require as much work as they did. The course staff demanded that all phases were done and demanded a certain amount of documentation as proof. This took a lot of time, and most of the work done in these phases was done to get the documents approved by the Customer Driven Project staff, not to actually create a better solution for the customer.

Consequently, if we could choose one thing to change in Customer Driven Project, it would be to move focus from documentation to the product we deliver. We are not saying that all the documents we produced are not useful —

they are — but it takes considerable effort to complete and perfect them and get approval from the course staff. We have no doubt that for the customer, it would be better if we spent the time we used on perfecting the documents on the product instead.

The greatest problem with this task was that it required more time than we had available. We therefore used some extra time in the requirements specification phase trying to fit the requirements to the available time. In the end we had to push the customer to get it approved, but we feel that it was a fair compromise in the end. Hopefully the customer appreciates a smaller, working program more than a half-finished, bigger program.

The task required us to understand how SP works, since our understanding was essential for getting the program right. During the implementation phase we discovered lacks in our understanding. Since SP is a language undergoing development, the customer also had to make decisions which affected how the SP language would be used. By developing SP Light, the customer was in a sense forced (through our questions) to view the SP language from an implementation point of view. We therefore feel that it is fair to say that developing SP Light also contributed to the development of SP as a language.

We also feel that we have focused on the right things to implement and that the product we delivered to the customer has a lot of value. In conclusion we would say it was a meaningful task which resulted in a product with value.

8.3 Customer Driven Project as a Course

We will now evaluate certain aspects of this project as a course at NTNU.

8.3.1 Supervisors

Our two supervisors were Ph.D. student Rune Molden and Stud.techn. Gunn Olaussen. We would like to first mention that their help and feedback has been of great value to us, and we are very grateful for it.

Mr. Molden evaluated our work. He was very objective in his comments, and often asked us questions about things which we previously had not thought about. He was very good at pointing out things in our documents which were ambiguous, unclear or not easily understood, and suggested different ways of doing it. He also helped us greatly with his knowledge of software engineering and how things should be done and what external sensors would expect.

Ms. Olaussen gave us invaluable feedback that was very detailed and thorough, and she certainly put a greater effort into her work that what could be expected. All our questions were answered amazingly quickly and thoroughly. Since Ms. Olaussen took this course last year, she had a lot of knowledge about how we should do things. This has helped us tremendously. We really feel that she did a truly excellent job.

8.3.2 Other Customer Driven Project Staff

Our contact with the rest of the staff in Customer Driven Project has primarily been through e-mail and the home page. We are not fully satisfied with the home page of the course. It was not easy to find information on the home page, and some times the information posted there was erroneous. This led to some confusion. We recommend that the course staff put out the information in a more structured, clear and concise manner in the future.

The lectures varied in their relevance. The presentation by BearingPoint was relevant and very well adapted to our level of knowledge. The opposite was the case with the "IT-Arkitektur"-lecture by Einar Landre from Statoil ASA. Mr. Landre held a good lecture, but since all concepts have been taught in earlier courses at NTNU, it was a waste of time.

Also, the lectures were according to the Customer Driven Project class compulsory. Our group has been at all these lectures and at the last one (IT-Arkitektur), it was us and 5-7 other students. We ask that the Customer Driven Project staff either enforce that they are compulsory or make them voluntary. The time wasted could have been put to much better use working on the project instead.

8.3.3 Work Process Requirements

We would like to mention that our group decided not to follow the Customer Driven Project staff's strong recommendation to use use cases. We did not do this for the following reasons:

1. Use cases were already described in [1B]. Consequently we felt it was not necessary to repeat this process.
2. Even if use cases were not already described, they are not necessary to use in our type of project. Use cases are used to "[...] capture *who* does *what* with the system, for what *purpose* [...]" [25W]. Our product was a stand-alone application, and who used it, what it was going to do, and for what purpose they were going to use it was known from the beginning. Our primary challenge was implementing a program which was able to model and do calculations on SP diagrams, not finding potential users and uses for the program. This had already been done in [1B].
3. Use cases are also used to communicate with a customer with little technical understanding. Our customer was represented by a Professor and Ph.D. student of Computer Science and certainly did not lack any technical understanding. Consequently, we found more efficient ways of communicating technical information to the customer.

Even though we decided not to do create our own use cases, we could have used use case based cost estimation based on the use cases in [1B]. This was also recommended by the Customer Driven Project's staff, and there was a lecture on this method held by Professor Reidar Conradi on use case based time/cost estimation [31W]. We did not apply this method to the project due to the following reasons.

1. The project time frame was already pre-determined to be about 1500 hours. Thus there was no need to determine the cost of the project. It was going to be approximately 1500 hours in any event.
2. Even if we needed to do cost estimation, there are better methods described in, for instance, [5B]. We feel use cased based cost estimation method has little practical value, at least in our particular project. Professor Conradi's lecture contained an example of a project that was estimated to cost 2000 hours. However, by changing some of the constants used in the method, it turned out that the estimation was uncertain by +/- 1000 hours. We feel it is a waste of time to put a lot of effort into doing an estimation which has such a large degree of uncertainty. Professor Conradi agreed that the method might not suit our particular project. While we understand the staff's interest in use case based modeling from a research point of view, we hope it is understandable that we chose not to use this method.
3. Cost estimation methods would not help us to estimate the cost of the tasks where we needed estimation. With respect to cost, our greatest uncertainty was not how much time software design and implementation would take, but rather how much time we would need to get a final approval of documents, particularly the requirements specification. Since

formal cost estimation methods focus on the cost of design and implementation, we felt that they were unnecessary to use.

8.3.4 Workload and Coordination With Other Subjects

Customer Driven Project demand that each student shall use 24 hours[1] per week, all weeks throughout the project. This is an extremely high requirement. Earlier years this workload has been acceptable, since the students could spend almost all their time on the project and make up for missing study time in the two other subjects *after* project-delivery. "Kvalitetsreformen" [28W] changed this practice. We now have assignments and tests during the whole semester which count towards the final grades, forcing us to spend a lot of time on that as well. The result is a total workload above 50 hours per week.

We would like to particularly mention another project organized in another subject for students in the 4th grade at IDI, NTNU: TDT4230 Visualization. This course had its own project running in parallel with the Customer Driven Project, which two of the group members took. The workload has been extremely high throughout the semester and required a tremendous amount of work. Particularly, the TDT4230 course required delivery and presentation of a project report and nearly 3000 lines of program source code just before Customer Driven Project was about to be finished. It also required large amounts of work throughout the semester. We feel that this subject could be far better coordinated with the activities and workload of Customer Driven Project. The course lecturer in visualization refused to do this.

The other subjects clearly affect the time available to Customer Driven Project. In the interest of the customer, we therefore feel that the workload of other subjects should be taken into consideration when deciding the workload in Customer Driven Project. It is simply impossible to keep Customer Driven Project at the top of the priority lists when compulsory, grade-affecting exercises in other subjects has to be delivered at high frequencies throughout the semester.

In addition it was UKA this autumn. We feel that it is very unnecessary to have both Customer Driven Project and UKA in the autumn. UKA is a national event, with long traditions. We feel NTNU and IDI should support a big student festival as UKA and consider moving Customer Driven Project to the spring semester. Perhaps it could be swapped with the course "Experts In Team"?

8.3.5 Conclusion

Despite these flaws, Customer Driven Project is perhaps the best course taught at IDI. We have learned enormously much and the insight and knowledge gained are highly appreciated. In the next section, we will take a look at what we learned.

[1] 60 minutes per hour, not 45 minutes

8.4 The Process

In this section, we will describe how we worked as a team, work methods, and time spent. We will also try to comment specific episodes which affected the group.

8.4.1 Teamwork

How and where to work was left up to each group member. Prioritizing the tasks and deciding *what* to do however, was done by the Project Manager. This gave the team members a high degree of freedom, while keeping a steady course towards the target. Paper work was usually done separately, while modeling and detailed design was successfully done collectively. Coding was done separately in the start, but after integration of storage and GUI, coding was mainly done by two or more persons working together. This turned out to be a successful approach, because in the beginning, the parts of the code the group members were working on did not significantly depend on other parts, so the group members could work individually. When different parts of code had to be integrated, working in a group was more effective.

Because the group felt that this work style worked well, no other ways to work together was explored. We managed to work in parallel almost all the time. We believe this was possible due to good team working skills.

8.4.2 The group members

Members of the group were randomly selected. The result was a very diverse group. We attended different courses in addition to Customer Driven Project, we preferred different operating systems, some of us liked GUI programming while some did not like it, and so on. Fortunately we managed to turn our different preferences and skills to our advantage. This has been very valuable throughout the project. We can safely state that if all the group members had equal interests, the project would have been less successful.

Tasks were delegated to the person that had the best pre-requisites to solve the problem. An often seen negative effect is that that one person becomes the expert within one field and no one can replace him or continue his work. Working in pairs minimized this potential problem. The overhead with this approach was more than outweighed by the benefits of having lower dependencies between the group members.

8.4.3 Roles

We will now take a look at some of the roles we chose during the project.

Secretary

A secretary was chosen on the very first meeting. The secretary's role was primarily to take notes and write summaries of meetings (internal meetings, customer meetings and supervisor meetings). One group member held this role throughout the project. Because we did not rotate this role among group members, we got better and more consistent meeting summaries and consequently better meetings. We feel this worked very well.

Quality Assurance Manager

Unlike what was the case for the secretary role, we chose to rotate the job as a QA manager. The QA manager was responsible for overseeing the quality of the results (documents or code) of each phase. This worked very well in the start, but midways we chose to exclude some of the groups members from the job. The reason for this was better utilize the different team members' skills.

The QA role worked well. It contributed significantly to the overall project quality to always have someone who was responsible for checking and controlling that results carried a certain quality level.

Document Manager

The last role we want to mention is the Document Manager. The document manager had, together with the QA manager of the phase, responsibility for the overall quality and structure of the document. This involved document formatting, checking for spelling errors and rewriting bad formulations. We were lucky that our document manager had previous experience with the document preparation system we chose to use,LaTeX. This helped ease the learning curve for group members who had no previous experience with LaTeX.

8.4.4 Milestones

We tried to define milestones in the phase documents, A.2. This was not very helpful, since each phase spanned a very short time period. Mainly they provided a way to discuss status with our customer and supervisors. It was more helpful to set small milestones for each person. These milestones usually marked the end of one activity and the start of another. We used these to handle dependencies in the project.

8.4.5 Knowledge and Skills Gained

We all learned a lot about using LaTeXand about using Java 1.5. We all used Eclipse, with mixed success. Some of us were used to other Integreated Development Environments (IDEs) and troubled a lot with Eclipse. Regardless, it was very useful to learn how to use another professional IDE in a large project.

Per Ottar and Erik got quite proficient with XML Schema and XMLBeans. They learned how they together provide functionality for loading and storing to XML, based on the schema. Christian L. wrote a parser and refreshed his memory about algorithms, while Christian B. and Magne got a lot of experience with programming with Swing in Java.

Working together as a group has caused no problems whatsoever. This is remarkable and must be because we share the same attitude towards teamwork, coding and Customer Driven Project. The nearest thing we have had regarding conflicts has been problems concerning how many work hours each team member should put in. This was not due to laziness or lack of skills, but because other courses at school and other activities demanded attention. We reckon this is a common problem all projects struggle with, and we feel we mastered it well. Although other projects took time from this project, we believe it was good for the group members to experience and having to coordinate between several different projects at the same time.

8.4.6 Time usage

When this report was printed the group had spent about 1450 hours in total. This is *effective* time, meaning we have excluded meals and breaks. The 1550 hours estimated was in other words a good estimate. Over 50% of total time spent has been used on design and implementation[2], which we feel is very good. We do not have statistics for other projects, but feedback from the our supervisors and other groups indicate that this is very good. We would like to point out that the customer estimated total workload to be suitable for 6-7 persons attending Customer Driven Project. Our group consists of 5 persons... (Please refer to section 8.3.4 for a description of what we feel about the workload this semester.)

Our time estimates from the Project Directive can be found in 2.1. Most noteworthy is time spent on the Requirements Specification, where we used lot more time than planned. This was due to the misconception that the customer already had a complete requirements specification. The customer's contribution ([1B]) was helpful, but we did not expect that so many things in it was outdated or had to be changed. Our estimates for the time spent on this phase were therefore inaccurate.

8.4.7 Risks

The risks in the project are shown in Appendix A.4. We will now discuss what risks occurred throughout the project, how they affected us and how they were handled.

[2]includes testing

Unforeseen Risks

First, let us look at the risks which occurred but which we did not foresee. There was one very markable unforeseen risk which did occur: That the workload in other courses the group members participated in would require so much time that prioritizing down Customer Driven Project was unavoidable. We did not expect this to be the case, since we did not believe the workload in the other courses to be as high as it was. If we had known this initially, we might have planned our work differently.

Foreseen Risks

Some, but not all, of the events listed in Appendix A.4 *did* occur.

Risk number 1, the risk of making bad design decisions, to some extent occurred. The most obvious mistake was to use XML as persistent storage. SP is not strictly hierarchic. For instance shall a diagram contain components and some components may contain diagrams. This relationship is not possible to store to XML, so it has to be built on top of it. A relational database could have done the job much easier. We did not foresee this problem. If we had, we had probably chosen other solutions for storage.

Risk 2 and 3, lack of communication within the group and the risk of documents not being managed properly did not occur.

Risk number 4 did (as was known in advance) occur. However, Erik and Per Ottar did have more time than we thought in the beginning, and managed to have time for working on the project even though they also had to spend considerable time on UKA–05.

Risk 5, that group members were absent due to illness, occurred. However, the group members that became ill, fortunately were only ill for a very short period of time, and it did not occur very often. Consequently, this risk did not have a considerable impact on the project work.

Risk 6, problems with configuring and installing software and tools, also occurred, and did somewhat slow down progress, but only in the beginning of the project. We managed to fix our problems quickly, so this did not affect progress significantly.

Risk 7, that the requirements specification was inconsistent, did not occur. We did not have any problems with the requirements specification being inconsistent. This was expected, since we used considerable time on this phase.

Risk 8, pc or hard drive crash did not occour.

Risk 9, serious illness or death in near family did occour. Erik's grandfather passed away, and he was absent for an entire week. The group was not slowed down, but Erik naturally did not work that week.

8.5 Future Work and Conclusion

In this section, we will summarize what future work can be done with SP Light. We will also conclude this report.

8.5.1 Further Work

The customer has planned to introduce SP Light as a modeling tool in the course TDT4220 Performance Evaluation at IDI, NTNU next spring. Their plan is to base further development of SP and SP Light on experiences collected through the use in this course.

Wiki

During the project, we have spent considerable time adding content and structure to the wiki. SP Light *plus* the wiki is a complete solution; providing help with the SP language, as well as help on the functionality in SP Light. SP Light without the wiki has all the functionality, but might prove less user friendly, since only tooltips are provided for guidance. The wiki will be extended further before and during the course Performance Evaluation. No coding is necessary to extend it.

Rules

The customer plans to use SP Light to play with the rules and possibly create new rules. A complete tutorial on how to extend SP Light with new rules can be found in Section 6.3.4. We have also made extensions to allow easy implementation of new rules.

Further Extensions

As described in the implementation chapter we did not have time to implement all requirements. Therefore the customer is likely to want to implement them in the future. Versioning, drag-and-drop, and undo/redo are all missing.

Implementing drag-and-drop of components from the toolbox panel should mostly require changes to `splight.gui.ToolBoxPanel` and changes to `splight.gui.drawing.toolbox.ComponentTool`. Most of the time spent to implement this would probably come from getting a good understanding of the existing code.

Undo/redo is a fairly complicated feature to implement as it has to keep track of all changes done by the user and group them into undoable/redoable steps. One action might include one of more changes to the data models, for instance, changing a component using the component editor might update multiple fields, but the user will expect to be able to undo it using one step. A possible solution would be to keep track of when changes are done to a model

object and take a snapshot of it which can be used to return it to the state it had before it was changed.

Versioning should mostly require changes to splight.storage.DiagramModel where it should be possible to implement fairly transparent to the rest of the program. Some thought should be given to exactly how and where this functionality is useful, since usability is critical for how useful it will be.

It is difficult to give a good estimate of the time required to implement these features. The customer will have to start by getting a fairly good understanding of how the code works before they can actually start implementing additional features.

8.5.2 Conclusion

We believe our product gives all the effects listed in Section 2.2.5. To add weight to this claim we want to point out that the process of creating SP Light already has contributed to the further evolvement of SP, through forcing the customer to concretize the concepts. The product is also the first of its kind to combine a GUI which supports modeling in SP and allows calculations and rule checks. SP Light has a commonly used, recognizable interface, which we believe will be appreciated by users new to SP. It will also gain users familiar with modeling in SP: No longer do they have to use several programs just to draw diagrams and do simple calculations on them. We believe SP Light successfully serves the purpose outlined in the project mandate in Section 2.2.

Our group has worked very well. As we mentioned, there are two primary reasons: the group members worked hard and had the same attitude towards the course and had different interests. Every single group member had something to contribute to the project, and the group dynamics were very positive — we feel fortunate to have experienced absolutely no significant internal arguments.

We have learned a lot from this course, both in terms of working with other people in a project and increased our competence in certain areas of computer science. We are proud that the project work has helped our customer solve a problem, and look forward to watch how SP Light evolves in the future.

Bibliography

Articles and Books

[1B] Løvstad, Jakob S.
Design requirements for an SP toolset
Software Engineering Group, NTNU, Spring 2004.

[2B] Hawryskiewycz, Igor
Introduction to Systems Analysis and Design, 5.ed.
Prentice Hall, 2001.

[3B] Software Engineering Standards Committee of the IEEE Computer Society
IEEE Std 830–1998: IEEE Recommended Practice for Software Requirements Specifications
Institute of Electrical and Electronics Engineers (IEEE), Inc., 1998.

[4B] Aho, Alfred V., Sethi, Ravi and Ullman, Jeffrey D.
Compilers: Principles, Techniques and Tools, 1.ed.
Prentice Hall, 2003

[5B] van Vliet, Hans
Software Engineering: Principles and Practice, 2.ed.
J. Wiley and Sons Publishers, 2000

[6B] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford
Introduction to Algorithms, 2.ed.
MIT Press, 2001.

Web Pages

[1W] Sun Microsystems, Inc.
Code Conventions for the Java Programming Language.
`http://java.sun.com/docs/codeconv/`.
Last visited 15-09-2005.

[2W] Price, Derek R., Ximibot and Free Software Foundation, Inc.
CVS — Open Source Version Control
`http://www.nongnu.org/cvs`
Last visited 26-09-2005.

[3W] The Eclipse Foundation
Eclipse.org Main Page
`http://www.eclipse.org`
Last visited 26-09-2005.

[4W] Free Software Foundation, Inc.
GNU General Public License — GNU Project — Free Software Foundation (FSF).
`http://www.gnu.org/copyleft/gpl.html`.
Last visited 20-09-2005.

[5W] Apache Software Foundation
Apache License, Version 2.0
`http://apache.org/licenses/LICENSE-2.0`
Last visited 26-09-2005.

[6W] Open Source Initiative
The BSD License
`http://www.opensource.org/licenses/bsd-license.php`
Last visited 26-09-2005.

[7W] Umbrello
Umbrello UML Modeller.
`http://uml.sourceforge.net`.
Last visited 26-09-2005.

[8W] The Mogwai Project
Mogwai ER-Designer
`http://mogwai.sourceforge.net/erdesigner/erdesigner.html`
Last visited 26-09-2005.

[9W] DbModeller
DbModeller
`http://www.horsman.co.nz/story.do?id=69`
Last visited 26-09-2005.

[10W] Team Synergy
Cohesion
`http://cohesion.it.swin.edu.au/`
Last visited 26-09-2005

[11W] Dia
Dia
`http://www.gnome.org/projects/dia/`
Last visited 26-09-2005

[12W] Wikipedia
Open-source license — Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/Open-source_license`
Last visited 26-09-2005

[13W] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung
Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung
`http://www.v-modell-xt.de/`
Last visited 27-09-2005

[14W] Sun Microsystems, Inc.
Javadoc Tool Home Page
`http://java.sun.com/j2se/javadoc/`
Last visited 28-09-2005

[15W] Sun Microsystems, Inc.
J2SE 5.0
`http://java.sun.com/j2se/1.5.0/`
Last visited 05-10-2005

[16W] Wikipedia
Work breakdown structure — Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/WBS`
Last visited 06-10-2005

[17W] Microsoft Corporation
Windows XP — Guidelines for Applications
`http://www.microsoft.com/whdc/hwdev/windowsxp/downloads/default.mspx`
Last visited 06-10-2005

[18W] The XML Schema Working Group
W3C XML Schema
`http://www.w3.org/XML/Schema`
Last visited 23-10-2005

[19W] The Apache XML Project
Apache XMLBeans
`http://xmlbeans.apache.org/`
Last visited 23-10-2005

[20W] eNode, Inc.
Model—View—Controller Pattern
`http://www.enode.com/x/markup/tutorial/mvc.html`
Last visited 26-10-2005

[21W] Gentleware AG
Poseidon for UML — by Gentleware, just model
`http://www.gentleware.com`
Last visited 27-10-2005

[22W] Microsoft Corporation
Microsoft Office Online: Visio 2003 Home Page
`http://office.microsoft.com/en-us/FX010857981033.aspx`
Last visited 27-10-2005

[23W] Sun Microsystems, Inc.
JavaBeans
`http://java.sun.com/products/javabeans/index.jsp`
Last visited 18-11-2005

[24W] IDI, NTNU
TDT4290 Customer Driven Project
`http://www.idi.ntnu.no/emner/tdt4290`
Last visited 19-11-2005

[25W] Bredemeyer Consulting
Use Cases and Functional Requirements
`http://www.bredemeyer.com/use_cases.htm`
Last visited 19-11-2005

[26W] Wikipedia
Extended Backus-Naur form — Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/Extended_Backus-Naur_form`
Last visited 19-11-2005

[27W] Wikipedia
Associativity — Wikipedia, the free encyclopedia
`http://en.wikipedia.org/wiki/Associativity`
Last visited 19-11-2005

[28W] The Bologna Process from a Norwegian Perspective
UFD - The Bologna Process from a Norwegian Perspective
`http://odin.dep.no/filarkiv/238593/041014Fact_Sheet_Bologna-Process.pdf`
Last visited 19-11-2005

[29W] Wikimedia Foundation
Mediawiki
`http://www.mediawiki.org/wiki/MediaWiki`
Last visited 20-11-2005

[30W] TDT4290 Group 7: Bergfjord, Magne, Bøhn, Christian, Drolshammer, Erik, Larsen, Leif Christian and Pahr, Per Ottar
Main Page — Splight
`http://splight.idi.ntnu.no`
Last visited 20-11-2005

[31W] Anda, Bente and Conradi, Reidar
Slides from the lecture *UseCase-based effort estimation of software projects* in course TDT4290 Customer Driven Project at NTNU
`http://www.idi.ntnu.no/emner/tdt4290/Foiler/usecase.ppt`
Last visited 21-11-2005

Abbreviations and Terms

Below, we have listed the abbreviations and terms used in this report.

Term	Definition
ADT	Abstract data type
API	Application programming interface
CPU	Central processing unit
CSM	Complexity specification matrix
EBNF	Extended Backus-Naur Form
IDE	Integrated Development Environment
IDT	Implemented data type
IPS	Instructions per second. Note that the term MIPS (millions of IPS) is more commonly used.
IEEE	Institute of Electrical and Electronics Engineers
J2SE	Java 2 Platform Standard Edition
LAN	Local area network
MiB	Mebibyte; 1 MiB $= 2^{20} = 1\,048\,576$ bytes
MS	Microsoft Corporation
MVC	Model—View—Controller pattern
PC	x86 compatible personal computer
SP	Structure and performance
UML	Unified Modeling Language
WBS	Work Breakdown Structure
XML	Extensible Markup Language

Appendix A

Project Definition Plan Appendix

A.1 Development Partners

A.1.1 Stakeholders

Customer Contact Persons

Peter Hughes
Phone: 73 59 87 18 (Norway), 00 44 1 260 276 740 (England)
E-mail: peterh@idi.ntnu.no or phh@modicum.demon.co.uk

Jakob Sverre Løvstad
Phone: 73 88 70 91 (Norway)
E-mail: jakobsve@idi.ntnu.no

Course Supervisors

Gunn Olaussen
E-mail: gunno@stud.ntnu.no

Rune Molden
E-mail: runemol@idi.ntnu.no

A.1.2 Group Member Information

Contact information for each member of the group is shown in Table A.1.

Name	**E-mail**	**Cell #**	**Home #**
Magne Bergfjord	magnekri@stud.ntnu.no	41 60 33 73	N/A
Christian Bøhn	christbo@stud.ntnu.no	97 16 41 82	N/A
Erik Drolshammer	drolsham@stud.ntnu.no	92 08 57 94	73 88 88 57
Leif Christian Larsen	leifchl@stud.ntnu.no	99 77 50 12	73 88 99 18
Per Ottar Pahr	pahr@stud.ntnu.no	41 69 99 08	73 88 97 06

Table A.1: Contact information

A.2 Phase descriptions

A.2.1 *Phase 1.* Planning

The planning phase is the first part of the project. The result of the planning phase is the project definition plan (PDP). The PDP is a dynamic document that will be updated for the entire duration of the project, but the first delivery of the PDP is a milestone for the planning phase.

A.2.2 *Phase 2.* Pilot Study

The overall goal of this phase is to understand and describe the problems to be solved, understand the desired solution and to chose a strategy for attaining the desired solution. Table A.2 shows how we have divided the work among us in the first half of the pilot study:

Activity	**Responsible**
Introduction to SP	Magne
Description of the current situation	Christian B.
Description of the desired solution	Christian L.
Market analysis	Per Ottar
Evaluation criteria	Erik

Table A.2: Delegation of workload in the pilot study phase

After this preliminary work, we will outline some alternative strategies for attaining the desired goal. The following group evaluation will conclude with a preferred strategy, which will be presented for the customer for approval. This marks the end of the pilot study.

A.2.3 *Phase 3.* Requirements Specification

In the requirements specification phase we will define formal requirements for the system based on our knowledge from the pilot study. To be able to define good requirements we need to decide on an overall program architecture. The specific requirements will be divided into non-functional, GUI and functional requirements.

An outline of the test plan, with overall testing requirements, should also be devised during this phase. Everything concerning testing will be put in a separate document.

To gain a common understanding of the GUI with the customer, we use screen shots. These will be sketches to clarify specific requirements.

Milestones

The milestones for the requirements specification phase are listed below.

- Customer approval of the final requirements specification.
- Overall test plan finished.

The work distribution for the requirements specification document is shown in Table A.3.

Activity	Responsible
Introduction	Christian L.
Overall system description	All
Non-functional	Christian L.
Functional	Erik
GUI	Magne
Testing	Magne
Conclusion/summary	Christian B.

Table A.3: Delegation of workload in the requirements specification phase

This phase demands a high degree of collaboration, therefore there will be at least two persons working on each phase. Christian B. will be primarily working on functional and GUI requirements, while Per Ottar will help Magne with the test document.

A.2.4 *Phase 4.* Design Specification

This phase will take us from an overall design description, down to a level where it is possible to see precisely how to implement it. We will use a Work Breakdown Structure to aid us in this task. For more information about WBS see [16W].

Milestones

It is hard to set specific milestones for this phase, so we have settled down to using the obvious:

- Design specification completed.
- Test documentation completed.

This phase is special, because it will start before the requirements specification is 100% done, and implementation will start before the design specification is 100% done. This is unfortunate, but necessary since the customer was unavailable for an entire week. To minimize the damage, we try to work on parts that we are sure the customer agrees on.

We will start this phase by choosing development tools. This choice is based on the assumption/hope that the tools may collaborate in an effective way. Thereafter we will start to define some top-level modules and break them down into

pieces. The breakdown will continue until everything can be explained by work units. We defined work units to be a part of the program which could be coded in one week or less.

The test documentation will be made according to the V-model. See 7 for further information.

The whole group will work on both the WBS and the modeling. UML will be used for class diagrams and sequence diagrams. If we need diagrams that are not included in the UML standard, we will explicitly explain it.

Table A.4 shows the persons *responsible* for the given tasks. Christian B. is responsible for QA and have also the final word when it comes to conflicting design propositions.

Activity	**Responsible**
Introduction	Christian B.
WBS	Christian L.
Modeling	All
Testing	Magne
Wiki	Per Ottar
Conclusion/summary	Erik

Table A.4: Delegation of workload in the design phase

A.2.5 *Phase 5.* Implementation and Testing

This phase is partly overlapping with the design phase. A wiki will be used for documentation, so we will set it up before we start to code. We also want to explore the possibilities of XMLBeans. Modeling of the data models depend on what restrictions XMLBeans enforces, so its implementation will be undertaken at a very early stadium. We hope using an existing solution like XMLBeans will save time and heighten quality.

We plan to iterate over the design phase and the implementation phase. We think this modification of the waterfall model suits our way of thinking best. When we get to the actual coding we divide us into two subgroups; GUI and business logic. The subgroup writing the business logic is also responsible for storage in XML. Magne and Christian B. will mainly work on GUI, while the rest of us will work on the business logic. We focus on early integration between GUI, through business logic and down to storage in XML-files. We hope this approach will give more feedback from the customer.

All tests were not finished in the design phase, so checklists for unit testing will be written at the start of the implementation phase. Magne is responsible for the tests, so he will not be QA responsible for a particular phase. Christian L., Per Ottar and Erik will therefore be responsible for two phases each.

Commenting and documentation through Javadoc will also take place in this phase.

A.2.6 *Phase 6.* Project Documentation and Evaluation

Almost at the end of the implementation we will write an installation manual. We will also evaluate the project, to convince ourselves what decisions were smart and which were not. We will not perform a thorough post-mortem analysis, only summarize what went well and what did not.

A.2.7 *Phase 7.* Presentation and Demonstration

We plan to run a PowerPoint presentation, mixed with a demonstration of the program. The audience will receive a paper copy of the presentation before it starts. Christian B. will do the different actions in the demonstration, while the PowerPoint presentation will be held by Erik. The rest the group will be available for questions both during the presentation and after the presentation. We have 35 minutes at our disposal for the presentation (maximum 30 minutes) and questions (minimum 5 minutes). We plan to reserve at least the last 10 minutes for questions.

A.3 Templates

The templates discussed in Section 2.5 are shown below.

A.3.1 Meeting Summons Template

A template for meeting summons along with some examples of what information the summons will contain is shown below.

```
** Møteinnkallelse <møtetype; dvs. kunde-,
                   veileder- eller gruppemøte> **
Tid: <Dag, dato, start/stopp-tid>

** Mål for møtet **
a. <mål 1, feks. ‘‘delegere oppgaver til prosjektdirektivet’’>
b. <mål 2>
c. <mål 3>

** Agenda **
1. Godkjenning av dagsorden
2. Godkjenning av møtereferat fra forrige møte
3. Hva har vi gjort siden sist; kort om sentrale valg som er tatt

4. (Feks.) Møteinnkallelser og referater
5. (Feks.) Reponstider
6. (Feks.) Godkjenning av fasedokumenter
7. (Feks.) Interessenter
8. (Feks.) Prioritering av krav

9. Kritikk av møtet
10. Eventuelt
11. Neste møte

** Ekstrabeskjeder **
Husk å ta med laptop
```

A.3.2 Meeting Reports Template

A template for meeting reports is shown below, along with some examples of the information it will contain. Note that each section in the report corresponds to each point on the agenda.

REFERAT FRA GRUPPEMØTE I TDT4290 KUNDESTYRT PROSJEKT

Møtedato: 05.09.2005
Møtested: F304, Elektrobygget, NTNU
Til stede: Magne Bergfjord / Gruppemedlem,
Christian Bøhn / Gruppemedlem,
Erik Drolshammer / Gruppemedlem,
Leif Christian Larsen / Gruppemedlem,
Per Ottar Pahr / Gruppemedlem

1 GODKJENNING AV MØTEINNKALLELSE

Møteinnkallelse godkjent uten innsigelser.

2 RUNDE RUNDT BORDET

Magne Bergfjord har startet på prosjektmandat.

....

3 <agenda punkt 3>

<sammendrag av agenda punkt 3>

4 <agenda punkt 4>

<sammendrag av agenda punkt 4>

5 <agenda punkt 5>

<sammendrag av agenda punkt 5>

<fortsetter nedover helt til alle punktene i agendaen er dekket>

A.3.3 Status Report Template

A template for the Status Report is shown below.

Statusrapport for Gruppe 7, uke XX

1 Oppsummering

```
<Hva har gruppen brukt tid på gjennom uken?>
```

2 Utført arbeid

2.1 Dokumenter

```
<Hvilke dokumenter har gruppen arbeidet med?>
```

2.2 Møter

- Møte1
- Møte2
- Møte3

2.2 Aktiviteter

```
<Hvilke aktiviteter har gruppen utført?>
```

2.3 Andre

```
<Annet?>
```

3 Tid, Risiko, Omfang, Kostnad og Kvalitet (TROKK)

3.1 Tid

```
<Hvordan ligger prosjektet an tidmessig?>
```

3.2 Risiko

```
<Risikomomenter; sannsynlighet, konsekvens og tiltak>
(Gjerne referanse til risikotabellen i prosjektdirektivet.
```

3.3 Omfang

<Har omfanget endret seg?>

3.4 Kostnad

<Hvordan er timesforbruket i forhold til planen?>

3.5 Kvalitet

<Har det skjedd noe som påvirker produktets kvalitet?>

4 Problemer

- Problem1
- Problem2
- Problem3

5 Planlagt arbeid for neste uke

Møter

- Møte1
- Møte2
- Møte3

Aktiviteter

<Hvilke aktiviteter skal gjøres neste uke?>

A.4 Risk analysis

Nr	Activity	Risk Factor	C	P	Strategy	Time Limit	Responsibility
1	Impl.	Bad design decisions	H	M	Avoid, redesign if necessary	October	Magne
2	All	Lack of communication in group	H	L	Avoid	Whole project	Whole group
3	All	Documents are not managed properly	H	L	Mitigate, proactive	Whole project	Document manager
4	All	Erik and Per Ottar are active in UKA-05	M	H	Mitigate, proactive	October	Erik, Per Ottar
5	All	Absence caused by illness	M	H	Mitigate	Whole project	Project manager
6	All	Installation and configuration of software	M	L	Mitigate, proactive	First half of the project	Whole group
7	Design	Inconsistencies in requirements specification	M	L	Mitigate, proactive	Start of implementation	Whole group
8	All	PC or hard drive crash	M	L	Mitigate, proactive	Whole project	Whole group
9	All	Serious illness or death in near family	L	H	Mitigate	Whole project	Whole group

Table A.5: Risk analysis

Terms and measures

We use the following measures in the risk analysis.

Probability (P) There are three levels of probability: high (probability of risk factor occurring is estimated to be over 70%), medium (probability of risk factor occurring is estimated to be between 30 and 70%), and low (probability of risk factor occurring is estimated to be below 30%).

Consequence (C) There are also three levels of consequence: high (risk factor has a near critical or very high impact on the project), medium (risk factor has a non-critical but considerable impact on the project), and low (risk factor has a near negligible impact on the project).

Appendix B

Pilot Study Appendix

B.1 Open Source Licenses

We will now give a brief description of the most common open source licenses. For further information, we refer to [12W].

B.1.1 GNU General Public License — GPL

The GNU General Public License [4W] is the GNU Project's free software license. It is one of the most widely used open source licenses. The GPL requires that derivative works are also licensed under the GPL, and that the source code is made available with any binary distribution of a program. The GPL allows commercial distribution, but the buyer will also have the right to redistribute or sell the software under the terms of the GPL. The GPL forbids redistribution under more restrictive terms.

B.1.2 Apache Software License

The Apache Software License [5W] by the Apache Software Foundation is an open source license made to fit the goals of the Apache Foundation. The license is also usable for projects not affiliated with the Apache Software Foundation. The license requires that the copyright notice must be included in any distribution of derivative works, but it allows distribution in binary form only and use in closed source projects. It is worth noting that the Free Software Foundation considers the Apache Software License to be incompatible with the GPL.

B.1.3 BSD License

The Berkeley Software Distribution License [6W] is an open source license that is less restrictive than the GPL and the Apache License. The license requires that the copyright notice is included when the software is redistributed, both in source and binary form. The license does not require distribution of the

source code of derivative works, and modifications may be used in commercial, closed source products as long as the copyright notice is included. A popular example of this is the use of BSD code in the TCP/IP stack in older versions of windows.

Appendix C

Design Appendix

C.1 XML Schema

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- definition of simple elements -->
<xs:element name="fileName" type="xs:string"/>
<xs:element name="dataLoad" type="xs:decimal"/>
<xs:element name="workLoad" type="xs:decimal"/>
<xs:element name="topConnector" type="xs:string"/>
<xs:element name="lowConnector" type="xs:decimal"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="numberOfComponents" type="xs:integer"/>
<xs:element name="extent" type="xs:decimal"/>
<xs:element name="limit" type="xs:decimal"/>
<xs:element name="aString" type="xs:string"/>
<xs:element name="type" type="xs:integer"/>
<xs:element name="height" type="xs:integer"/>
<xs:element name="width" type="xs:integer"/>
<xs:element name="x" type="xs:integer"/>
<xs:element name="y" type="xs:integer"/>


<xs:element name="upperComponent">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="component" />
</xs:sequence>
</xs:complexType>
```

```
</xs:element>
<xs:element name="lowerComponent">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="component" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="position">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="height" />
<xs:element ref="width" />
<xs:element ref="x" />
<xs:element ref="y" />
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="csm_j">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="aString" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="csm_i">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="csm_j" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="idtMatrix_j">
```

```
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="aString" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="idtMatrix_i">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="idtMatrix_j" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>


<!-- definition of complex elements -->
<xs:element name="connector">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="upperComponent"/>
<xs:element ref="lowerComponent"/>
<xs:element ref="type" />
<xs:element ref="csm_i" />
<xs:element ref="idtMatrix_i" />
</xs:sequence>
</xs:complexType>
</xs:element>


<xs:element name="component">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="topConnector" maxOccurs="unbounded"/>
<xs:element ref="lowConnector" maxOccurs="unbounded"/>
<xs:element ref="name" />
<xs:element ref="numberOfComponents" />
```

```
<xs:element ref="extent" />
<xs:element ref="limit" />
<xs:element ref="position" />
</xs:sequence>
</xs:complexType>
</xs:element>


<xs:element name="diagram">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="component" maxOccurs="unbounded"/>
<xs:element ref="fileName" />
<xs:element ref="dataLoad" />
<xs:element ref="workLoad" />
</xs:sequence>
</xs:complexType>
</xs:element>



</xs:schema>
```

C.2 Work Breakdown Structure

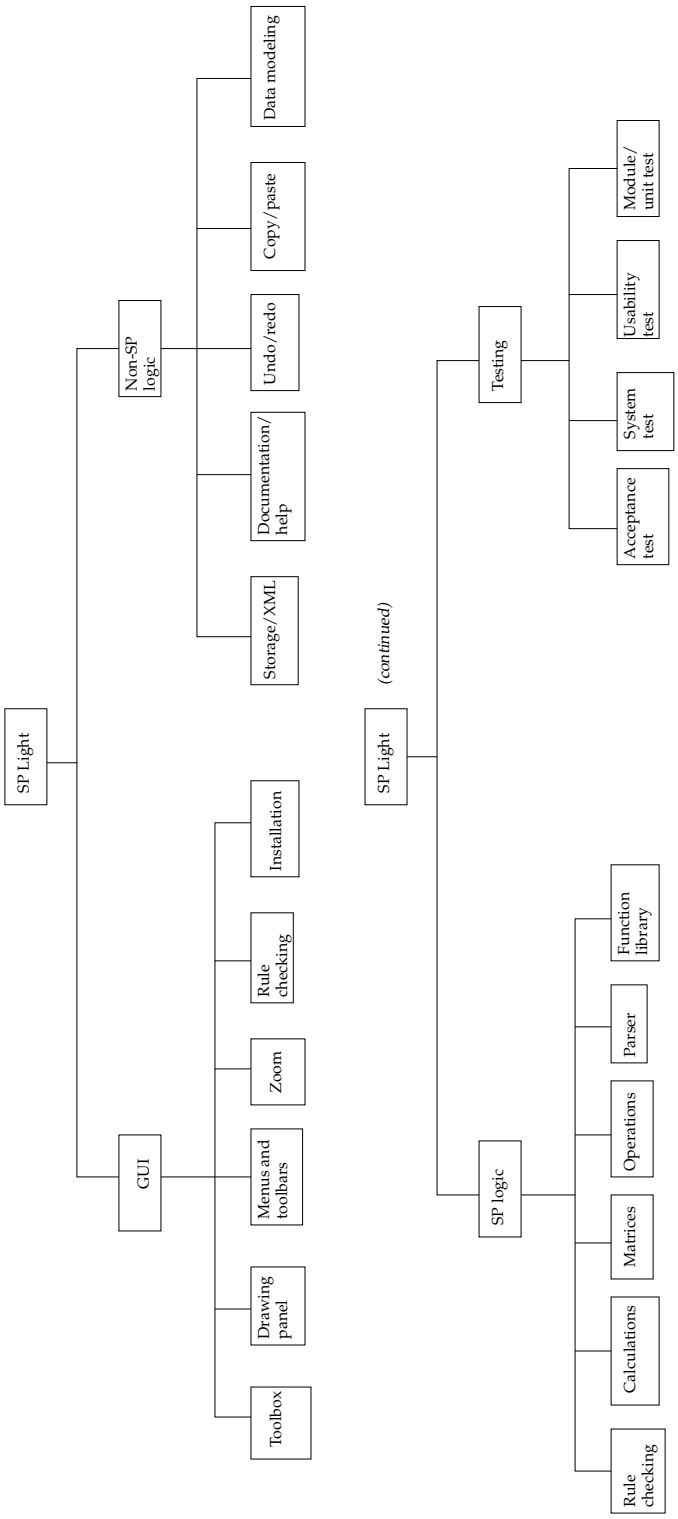


Figure C.1: Work Breakdown Structure

Appendix D

Implementation Appendix

D.1 XML Schema

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- http://xmlbeans.apache.org/docs/2.0.0/guide/
conXMLBeansSupportBuiltInSchemaTypes.html -->
<xs:schema elementFormDefault="qualified"
attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Definition of simple elements -->
<xs:element name="fileName" type="xs:string"/>
<xs:element name="name" type="xs:string"/>
<xs:element name="aString" type="xs:string"/>
<xs:element name="subDiagramName" type="xs:string"/>
<xs:element name="versionName" type="xs:string"/>
<!-- Internal type.
        Created to implement a 2-dimensional array.-->
<xs:element name="service" type="xs:string"/>
<xs:element name="logicalStorage" type="xs:string"/>
<xs:element name="logicalStorageUnit" type="xs:string"/>
<xs:element name="topNode" type="xs:string"/>
<xs:element name="limit" type="xs:double"/>
<xs:element name="primitive" type="xs:boolean"/>
<xs:element name="speed" type="xs:double"/>
<xs:element name="activeDiagram" type="xs:int"/>
<xs:element name="upperConnectionPoint" type="xs:int"/>
<xs:element name="lowerConnectionPoint" type="xs:int"/>
<xs:element name="function" type="xs:string"/>
<xs:element name="description" type="xs:string"/>
<xs:element name="type" type="xs:int"/>
<xs:element name="numberOfComponents" type="xs:int"/>
<xs:element name="height" type="xs:int"/>
<!-- This is a Java Rectangel attribute. -->
<xs:element name="width" type="xs:int"/>
```

```
<!-- This is a Java Rectangel attribute. -->
<xs:element name="x" type="xs:int"/>
<!-- This is a Java Rectangel attribute. -->
<xs:element name="y" type="xs:int"/>
<!-- This is a Java Rectangel attribute. -->
<!-- Definition of complex elements -->
<xs:element name="upperComponent">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="aString"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="lowerComponent">
<xs:annotation>
<xs:documentation>Comment here! </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="aString"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="position">
<xs:annotation>
<xs:documentation>
This is a Java Rectangle attribute.
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="height"/>
<xs:element ref="width"/>
<xs:element ref="x"/>
<xs:element ref="y"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="workLoad">
<xs:annotation>
<xs:documentation> </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="matrix"/>
```

```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="dataLoad">
<xs:annotation>
<xs:documentation> </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="matrix"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="idtMatrix">
<xs:annotation>
<xs:documentation> </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="matrix"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="csm">
<xs:annotation>
<xs:documentation> </xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="matrix"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="list">
<xs:annotation>
<xs:documentation>Internal element</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="aString" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="matrix">
<xs:annotation>
<xs:documentation/>
</xs:annotation>
<xs:complexType>
```

```
<xs:sequence>
<xs:element ref="list" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="connector">
<xs:annotation>
<xs:documentation>
This is the data that shall be stored about a connector.
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="upperComponent" minOccurs="0"/>
<xs:element ref="upperConnectionPoint" minOccurs="0"/>
<xs:element ref="lowerComponent" minOccurs="0"/>
<xs:element ref="lowerConnectionPoint" minOccurs="0"/>
<xs:element ref="type"/>
<xs:element ref="csm" minOccurs="0"/>
<xs:element ref="idtMatrix" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="component">
<xs:annotation>
<xs:documentation>
This is the data that shall be stored about a component.
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="type"/>
<!-- Is used as a reference to a diagram,
when type (below) is subSystem og subDiagram)-->
<xs:element ref="name"/>
<xs:element ref="numberOfComponents"/>
<xs:element ref="position"/>
<xs:element ref="service" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="limit" minOccurs="0"/>
<xs:element ref="logicalStorage" minOccurs="0"
maxOccurs="unbounded"/>
<!-- extent is the summation of logicalStorages
in a Component -->
<xs:element ref="logicalStorageUnit" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="primitive"/>
<xs:element ref="speed" minOccurs="0"/>
```

```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="diagram">
<xs:annotation>
<xs:documentation>
This is the element describing a SP diagram.
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="type"/>
<xs:element ref="name"/>
<xs:element ref="component" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="connector" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="topNode" minOccurs="0"/>
<xs:element ref="workLoad" minOccurs="0"/>
<xs:element ref="dataLoad" minOccurs="0"/>
<xs:element ref="activeDiagram"/>
<xs:element ref="versionName" minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<!-- All functions are stored
        as strings and parsed later.-->
<xs:element name="libraryFunction">
<xs:annotation>
<xs:documentation>
Functions in the library.
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element ref="function"/>
<xs:element ref="description" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="project">
<xs:annotation>
<xs:documentation>
This is the main element.
It shall contain information about the project,
as well as a list of diagrams and a list of library functions.
```

```
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="name"/>
<xs:element ref="fileName"/>
<xs:element ref="diagram" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="libraryFunction" minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Appendix E

Testing Appendix

E.1 Usability Test Handout

E.1.1 Introduksjon

SP (Structure and Performance) er et modelleringsspråk for å modellere IT-arkitektur og ytelse. SP Light er et program vi holder på å utvikle for å lage SP-diagrammer. Språket består av forskjellige bokser og piler mellom dem, og i testen vil vi teste hvordan tegning av disse bør gjøres, samt litt generelt om lagring og hjelp-funksjon. Flere andre skal gjøre samme testen senere, så det er viktig at du følger instruksjonene under, slik at vi kan se etter om flere personer støter på de samme problemene. Snakk gjerne høyt under testen. Forklar hva du tror kommer til å skje når du trykker på ting, og fortell hvis ting ikke skjer som du tror.

For at vi skal kunne avsløre faktiske mangler kan ikke testleder gi hjelp underveis. Etter testen kan du spørre om ting du lurte på underveis.

E.1.2 Oppgaver

Testperson kan starte på oppgavene under, så snart testleder har startet programmet. Rekkefølgen kan fravikes om ønskelig.

- lag et nytt prosjekt
- lag to komponenter
- lagre prosjektet
- lukk programmet
- start programmet og hent frem tidligere lagret prosjekt
- bind komponentene sammen med en konnektor
- legg til en ny komponent
- sett på konnektorer slik at den nye komponenten er koblet til de to andre.
- bytt type på en konnektor

- slett en komponent
- finn to måter å bruke hjelp-funksjonalitet på
- finn hjelpedokumentasjon for “rules”.
- finn programmets versjonsnummer

E.1.3 Etter testen

Var det noe spesielt du strevde med? Hvorfor?
Var det navn på eller plassering av funksjonalitet du syntes var misvisende?
Hva synes du om vår variant av dra-and-drop? (Les: Merk og sett)
Synes du bruk av “tabs” virker som en god ide?
Skjønner du hvordan trestrukturen skal brukes til navigering?
Forslag til forbedringer?