Analysis of designers' work

Master's Thesis
Joost Meijles
Thursday, 2005 July 14

1 year Master Software Engineering

Supervisors
Universiteit van Amsterdam
Prof. Dr. P. Klint

Philips Medical Systems
Ir. S.B. Buunen
Dr. R.L. Krikhaar

Universiteit van Amsterdam

Vrije Universiteit

Hogeschool van Amsterdam

Summary

In this report the most important goals that a software designer has during his/her design work are described. These goals are structured and set out by use of the Goal Question Metric approach (GQM). The GQM approach is a widely used method for software measurement. The focus of the software measurement in this project is on the code-base. In other words all metrics will be answered by use of the code-base.

Software designers at Philips Medical Systems MR have several tasks, some of these are; coordination of designs, controlling code-base parts, the responsibility for the software quality and the development of software. According to these tasks a software designer has questions. To structure these questions the following project question is formulated; "Which information does a software designer, with respect to design, need during his daily work to control his team?". It is important to analyze the questions that a software designer has before starting with the implementation or usage of a tool. Without an analysis the usability of the tool will be unclear. Literature study has been done, to research what is done elsewhere in the world on this topic.

The process followed for this project, globally consists of three parts; interview series, goal question and metric definition, and validation. The interview series are input for the goal definition. The goals are prioritized by all software designers, which result into a "top three of goals". For these top three goals, questions and metrics are specified. The last step of the process is validating these questions and metrics. The validation is done by means of interviews and a presentation for all software designers.

As mentioned the top three goals are detailed by questions and metrics. The three goals are;

1. Perform a modification assessment
2. Make building blocks independent
3. Decrease complexity

The number one goal makes it is possible to analyze the impact of a modification in order to predict which other files have to be modified or tested. By having this overview, the software designer will be able to make a proper decision for a modification or redesign.

The number one goal has seven questions. The software designers stated the following questions as most important;

Q2 Which dependencies are there with the file that needs to be modified?
Q4 What is the amount of coupling with other files/classes?
Q5 Which files are logically coupled?

The first two questions relate to file dependencies. At this moment the answer to Q2 is gathered by 'hand', while Q4 is answered using intuition. These questions are a good starting point for a tool that supports the software designer during his work.

Question Q5 seems to be an interesting topic for further research. This question shows the files that were modified together over a significant number of releases, so called "logical coupling". In other words it answers "Programmers who changed this file, also changed…". This question is different then question number two that only shows syntactical coupling. Logical coupling has an overlap with syntactic coupling, but can identify all kinds of coupling. The feasibility and usability of logical coupling was prototyped. Conclusions made from the prototype are;

1. Logical coupling is relevant
2. Logical set coupling is not always relevant
3. Improvements are necessary to provide more reliable results

To conclude, metrics seem to be promising; the GQM significantly contributed to a proper definition of the questions a software designer has. Further investing has to be done, to decide how to implement Q2, Q4 and how to use Q5.

Table of contents

1 Introduction **4**
1.1 Background 4
1.2 Problem 4
1.3 Objective 5
1.4 Scope 5
1.5 Outline 5
1.6 Disclaimer 5

2 Approach **6**
2.1 Deliverables 7

3 GQM Analysis **8**
3.1 Followed process 9
3.2 Interviews & questions (raw) 10
3.3 Goal selection 10

4 Results **12**
4.1 Selected goals 12
4.2 Perform a modification assessment 13

5 Prototype **20**
5.1 Concept 20
5.2 Implementation 21
5.3 Results 23
5.4 Visualization 24
5.5 Analysis 25
5.6 Further research 26
5.7 Conclusions 27

6 Conclusions **28**

7 Recommendations **29**

8 Evaluation **31**
8.1 Project plan 31
8.2 Interviews 31
8.3 Mismatch literature & practice 31
8.4 Follow up 32

9 Terms **33**

10 Literature **35**

Appendix A Interviews **38**
A.1 Entity Relation Diagram – SWAT Workshop 38
A.2 Raw results 39
A.3 Survey results 43

Appendix B Prototype coupling visualization **44**

1 Introduction

1.1 Background

Philips Medical Systems provides equipment and technologies for the healthcare market. PMS Magnetic Resonance (MR) is responsible for the development of the MR Scanner. Within PMS MR there is a hardware and software department. The software department is divided over three sites; Best (NL), Cleveland (US) and Helsinki (FIN). The software development site in Best counts about 150 people.

The software department is divided into seven development groups;

- Scan Platform Software (SPS)
- Scanner Instrumentation Software (SIS)
- Clinical Scanner Packages (CSP)
- Platform Components and Services (PCS)
- Clinical Processing and UI (CPUI)
- Viewing and Storage Platform (VSP)
- Tool and Test Configuration (TTC)

Each group has its own expertise on a specific part of the MR scanner software. The software system contains about six million lines of code. The code-base is written in mainly three programming languages; C, C++ and C#. To control the code-base, customized version of IBM's version control system ClearCase is being used.

1.2 Problem

At Philips Medical Systems (PMS) there are 10 to 15 software designers working. A software designer has different tasks. Some of these tasks are; the technical lead for a small group of developers (about 10 to 15 people), coordination of designs, controlling code-base parts, the responsibility for the software quality and the development of software. The software designer has according to his tasks a number of common questions.
For example;

- When is a file modified?
- How many times a file has been modified?
- How many lines of code this module has?
- Do modifications often occur within this module?
- How many developers have modified this file?

To structure these questions interviews with senior designers were held and research has been done on which current techniques/tools are present to answer these questions.

There are currently no metrics available from the code-base that gives insight into the quality attributes of the developed software. The software designers are currently mainly focused on code-reviews, new requirements, etc. Besides this reason, there is currently no research performed at the software engineering discipline. Because of these two facts, the PMS organization mainly focuses its view inside the company. To trigger them to widen this view, this assignment was defined.

Concluding from the above problem illustration the following research question is addressed:

> *"Which information does a software designer, with respect to design, need during his daily work to control his team?"*.

1.3 Objective

The objective of the project is to answer "Which information does a software designer, with respect to design, need during his daily work to control his team?". This question is answered by means of goals that are detailed by questions and metrics. These goals, questions and metrics were gathered into a Goal/Question/Metric report. Secondly a recommendation to the PMS MR organization is given. In this recommendation the most important goals that a software designer has, and can be extracted from the code-base, are stated. Another component of the recommendation is the rationale for the questions and metrics that are stated for the most important goals.

1.4 Scope

The scope of this project is to propose a set of goals, questions and metrics. Recommendation will be done on the proposed goals, questions and metrics. A prototype is developed, to examine the feasibility and usability from a subset of the metrics. Out of scope is the development or selection of a tool that implements all the metrics, which are needed to answer the stated goals and questions.

1.5 Outline

First the approach used for the project is described in chapter 2. Then the used GQM analysis method will be set out by means of a literature and practice description in chapter 3. The results from the GQM analysis are stated in chapter 4. A selection of the results is examined by means of a prototype; the prototype is set out in chapter 5. The conclusions that can be made from the project are described in chapter 6. A recommendation to the Philips Medical Systems MR organization is given in chapter 7. A reflection on the project will be done in chapter 8. The used terms and literature are named in respectively chapter 9 and 10.

1.6 Disclaimer

Due to proprietary rights the exact figures and data in this document have been slightly modified, however preserving its essence with relation to the case we want to illustrate.

2 Approach

The project is divided into six phases; literature study report, first interview series, Goal Question Metric (GQM) analysis, prototyping, second interview series, GQM analysis and recommendation. (see Figure 1).

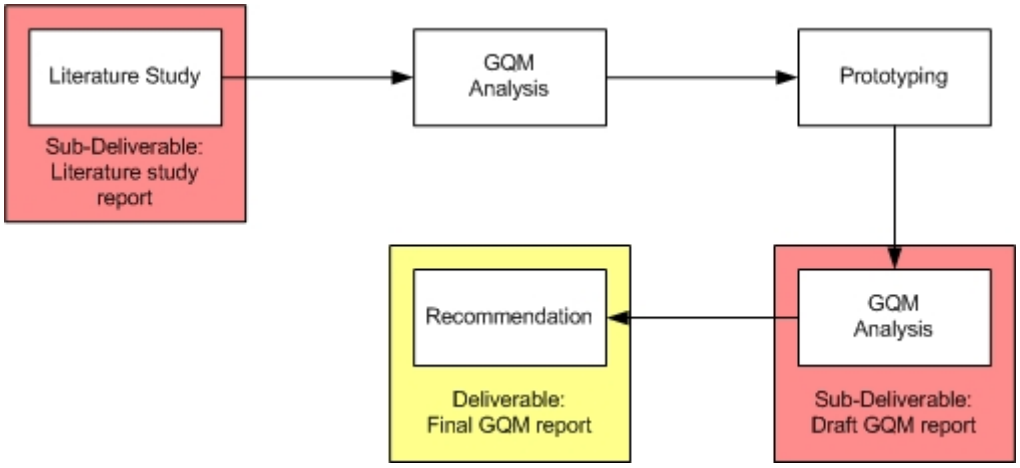


Figure 1 Approach overview

Before conducting the interviews all relevant information is collected in a *literature study* report. The subjects in this report are; the rationale for the GQM approach, and software metrics commonly used with respect to software design.

The second step in the project is conducting interviews with 40% of the software designers and other stakeholders. The target of the interviews is to gain insight in what goals software designers have according to design. This is the first step towards the goal, question, and metric report. As result of the interviews, an unordered list of goals is created. To order the goals that resulted from the interviews, a prioritizing survey is done with all software designers. Resulting from the prioritizing of the goals, it is possible to point out the most important goals and to start with the *GQM analysis*. The set of goals will be further detailed into questions and metrics. All the goals will be stated in a draft GQM report.

Prototyping is based on the specified questions and metrics. The prototype addresses a question, from which the usability and feasibility is not clear yet, but is interesting with respect to a software designer' goal. The prototyping is done parallel at the second interview series. In this way, it is possible to discuss the prototype applicability with the interviewee.

A second interview series is done to validate the proposed questions and metrics according to the selected goals. These interviews are done with a selection of the software designers, which are not interviewed during the first interviews series. In this way, the goals are supported by a larger group of software designers. During the *second GQM analysis*, the final version of the GQM report was written, by use of the results from the second interview series. At this stage, the most important goals of a software designer are clearly stated. These goals are specified by means of selected questions and metrics.

The feasibility and usability of the metrics that are stated in the GQM report are set out in a *recommendation* report. The recommendation report consists of the following items; tools that can be used, results from literature, results from prototyping and a management summary. Finally the GQM report and recommendation can be used as the input for a tool to be developed. This tool will support the software designers, with respect to the goals they have during their design work.

2.1 Deliverables

The project has three deliverables; a literature study report, a draft goal/question/metric (GQM) report and a final GQM report. All these deliverables are described shortly in this paragraph.

Literature study report
The literature study report contains all relevant information, which is read in papers as preparation for the master thesis. This report is used as basis for the first interviews and information from this report is used in the essay.

Draft GQM report
In this report the goals, question and metrics relevant for the software designer are stated. This is done according to the goal-question-metric approach. The report can be used to gain insight in the most relevant goals, questions and metrics that a software designer has.

Final GQM report
This deliverable is an extension of draft GQM report; the recommendations to PMS MR are added in this version. The recommendations describe how, why and which goals, questions and metrics should be implemented. It addresses the following aspects; techniques to use, tools, prototype results and relations with literature.

3 GQM Analysis

An important part of the project is the analysis of the goals that software designers have. To structure the analysis the Goal Question Metric (GQM) approach has been used. The goal question metric analysis (GQM) is a method for software measurement. It has been successfully applied by several software companies [Anacletto2003]. The method is based on the idea that organizational goals can be answered by defining a set of (refining) questions and measurable values (metrics). In this way it is possible to make clear what informational needs the organization has. These measurable values can be analyzed to see if the goals are achieved.

The GQM analysis can be distinguished in a number of steps [Veenendaal2000], these are;

Definition

1. Project characterization
2. Goal definition
3. Developing measurement program (questions and metrics)

Interpretation

4. Execution of the measurement program
5. Analyzing results

During the definition phase first the goals are selected according to project priority, risk or time in which a goal can be achieved. Goals can be directly related to business, project or personal goals. In other words different levels of goals have to be recognized. Second step is to refine these goals by use of questions. To answer these questions, metrics will be defined. In this context it is possible that a question can be answered by a number of metrics, and a metric can be used to answer one or more questions (see Figure 2). This step is a major one when developing a measurement program.

During the interpretation phase the actual measuring will be done, according to the specified metrics. Then the collected measurements will be analyzed and possible improvements to the stated goals, questions and metrics will be identified.

The scope of this project is limited to the first phase, although some prototyping will be done. The implementation of a tool is reserved for future work.

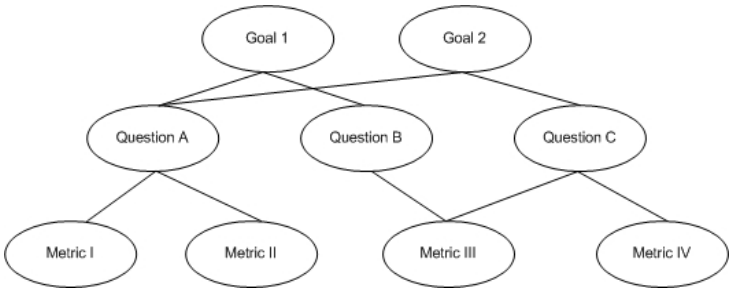


Figure 2 - Goal/Question/Metric relations

The success of a measurement program heavily depends on the factors that are taken into account. Niessink and van Vliet [Niessink2001] observed that it is important for a measurement program to actually generate value for an organization. This does not necessarily mean a higher profit, but can also be employee satisfaction, time-to-market, etc. Niessink and van Vliet identified the following factors as most important;

1. Start small and simple, increase measurement when organization and team has more experience
2. Encourage software designers to use measurement information
3. Use of existing metrics material
4. Make assumptions explicit
5. Monitor the changes implemented
6. Constantly improve the measurement program

These are important characteristics that have to be taken into account. The five industrial measurement programs analyzed in [Niessink2001] failed due to not applying, or succeeded due to applying these characteristics. Concluded can be that when one of these factors is not addressed, the measurement program is likely to fail.

During this project the success factor 1, 3 and 4 will be addressed explicit in the GQM and recommendation report. Success factor 2 will be tried to achieve by involving important stakeholder during interviews, reviews and presentations. The last two success factors are out of scope for the project and should be taken into account during a follow-up assignment.

3.1 Followed process

In this paragraph the followed GQM analysis process is described. The followed process is slightly different than the one that is described in the first paragraph of this chapter. In Figure 3 three phases are distinguished; interview series, goal selection and validation. These phases will be described in the following paragraphs. In Figure 3 two types of actions are specified; own interpretation and gathered results. During an own interpretation action, a step is made without interaction. And during an "gathered results" action there has been interaction with one or more stakeholders.

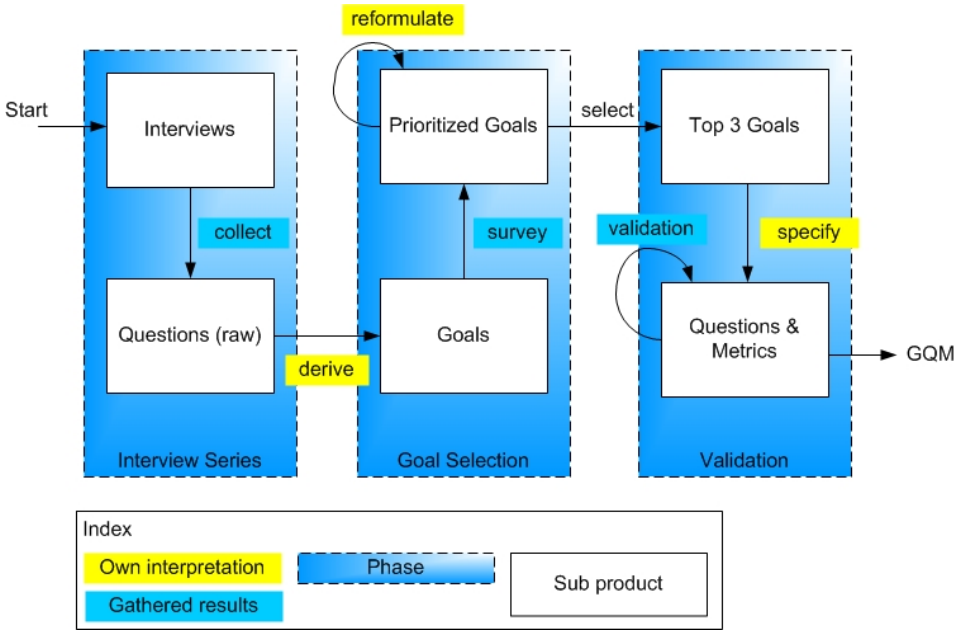


Figure 3 GQM process overview

3.2 Interviews & questions (raw)

The purpose of the interview series is to gain insight in what goals a software designer has during his work, with respect to design. Recently there has been a meeting of the SoftWare Architecture Team (SWAT). During this meeting an Entity Relation Diagram (ERD) has been produced. In this ERD all entities that a software designer uses are stated. At the interview series questions were asked according to analysis of designers' workshop ERD (see Appendix A.1). There are several advantages of this approach; software designers are familiar with the ERD, there is a structure during the interviews, other stakeholders can be interviewed, and the interview can be focused on design work.

The structure of the interview and the focus on design work are important, because a software designer has a lot of other tasks, not only with respect to design. During the interview series also other stakeholders (software architect, software engineer, segmentleader) were interviewed.

The ERD was used as a guideline during the interviews to ask the question "What do you want to know from this entity?". This results in questions like;

- What do you want to know from the code-base?
- What do you want to know from building blocks?
- What do you want to know from the archive?

The results from these questions were gathered in a matrix (see Appendix A.2) and grouped by entity from the ERD. Also the relations between the software designer and their questions are stated. To find the most important questions an addition is made. Following from these questions eleven goals are interpreted;

- Perform a modification assessment
- Make building blocks independent
- Decrease complexity
- Decrease coupling between classes
- Locate unused software
- Decrease code redundancy
- Improve document traceability
- Increase code readability
- Make time estimation for an activity
- Gain insight into the software characteristics
- Gain insight into the software evolution

3.3 Goal selection

The derive step (see Figure 3) results into eleven goals. Because not all goals do have the same importance and the time is limited for the project, a selection has to be made [Niessink2001]. The selection is done by use of a prioritizing survey among all the software designers. The software designers were asked to give a priority from one to eleven to the goals, whereby one has the highest priority. The results from this survey are shown in appendix A.3.

Derived from the survey, the top three of the ranked goals is "decrease risks during modifications", "decrease complexity" and "make building blocks independent". The top three goals are selected with the assumption that it is possible to specify the question and metrics within time for those three. When more goals would have been selected, from the ranked goals, the questions and metrics probably could not have been worked out properly.

While selecting the goals, the realization came that not all the goals in the survey were at the same level. Some goals are more abstract than others, and some goals can be questions for other goals. The right abstraction level of a goal, is the one that addresses the software designer' responsibilities to his superior. Resulting from this, goals are reformulated or removed and thereby grouped into three categories of goals. The grouping by category is done to make explicit the business goals they are trying to accomplish (see Table 1).

Table 1 Goal grouping

Group A Risk management
G1. Perform a modification assessment
Group B Software quality
G2. Decrease complexity
G3. Make building blocks independent
G4. Decrease coupling between classes
G5. Decrease code duplication
G6. Increase code readability
Group C Time-to-market
G7. Make time estimation for an activity
Group D Productivity
G8. Decrease learning curve in unknown software

3.4 Validation

Before the validation process can be initiated, appropriate questions and metrics have to be selected. The questions are divided into three groups; interview questions, literature questions and other questions. The interview questions are raw results from the interviews and therefore some redundancy exists between these questions. Reformulating is necessary to come to clear questions. Thereby some interview questions result in a metric instead of a question. The literature questions did not result from the interviews but are useful to accomplish the goal. Literature questions are useful because the interviewees did not come up with these, but are described in the literature as useful. Finally also other questions are stated. These questions are not explicitly found by interviews or literature, but are created according to ideas that came up while interviewing, reading literature and exploring the code-base.

The second step is specifying metrics for all questions, which is a complicated process; because there are a lot of metrics that can be stated for each question. A list with all kinds of metrics was created to simplify this process. This list was created by use of metrics that commercial tools collect, metrics that PMS tools collect and metrics proposed by literature. Because the usability of a metric is not always clear at first, some indicators are used. An indicator can be a table or a graph explaining the use of a metric or metrics.

During the validation another approach is used then during the interview series of the GQM analysis. During the interview series, open questions were asked to the software designers. During the validation phase mostly closed questions are asked. The questions during the validation focus on "Do you think this is a useful question / metric?" and "Do you miss a question / metric for this goal?". Another important object during the validation is to identify which questions are important and which are less important. The weighting of questions is done by use of two aspects; usage frequency and contribution to goal achievement. For example, when the usage frequency is high and the contribution is high, the question is very important. But when the usage frequency is low and the contribution is low, the question has a low importance.

4 Results

Till now only the followed process have been described. In this chapter the answer to the question "Which information does a software designer, with respect to design, need during his daily work to control his team?" will be given. This will be done by use of the top three goals that were chosen as most important. In the first paragraph, selected goals, the top three selected goals will be described shortly. Following the number one goal will be set out thoroughly.

4.1 Selected goals

The top three selected goals are named in Table 2. They are explained by use of structured statements, which cover the object of interest, purpose and perspective for the goal. These three goals are selected for further specification in the GQM report. For each goal, questions and metrics are given in the GQM report. Hereby a classification of metrics has been self-made;

- Direct
- Indirect
- OO-specific
- In time

Direct metrics are directly extracted from the source; no calculation is necessary. Indirect metrics are derived (calculated) from the direct metrics. OO-specific metrics can only be applied to object oriented programming languages. Metrics in time have a time dimension.

As stated in [Daskal1992], the metrics must be;

1. Simple to understand and precisely defined, facilitate consistency while calculating and analyzing
2. Objective, in order to decrease personal influences
3. Cost effective, return of investment
4. Informative, meaningful interpretation

These metric conditions will be set out in the paragraph, which is created for each question.

Table 2 Selected goals

Goal #1	Perform a modification assessment
Object of interest	Related files to be touched by a modified file.
Purpose	Analyze the impact of a modification in order to predict which other files have to be modified or tested.
Perspective	Examine the files from the viewpoint of the software designer.

Goal #2	Make building blocks independent
Object of interest	Dependencies between building blocks.
Purpose	Analyze the dependencies between building blocks in order to reduce these.
Perspective	Examine the building block dependencies from the viewpoint of the software designer.

Goal #3	Decrease complexity
Object of interest	Software complexity
Purpose	Analyze the software complexity course in order to reduce it.
Perspective	Examine the software complexity from the viewpoint of the software designer.

4.2 Perform a modification assessment

This goal will help the software designer to make a proper decision for a modification. Making a proper decision will prevent design smells [Martin2003] like rigidity and viscosity to be introduced or increased. Rigidity addresses the fact that it is hard to modify the system, other components have to be modified before the intended modification can be performed. And viscosity implies that it is harder to do things right then wrong. If a modification has to be done, the developer has different choices how to do this. When it is easier to do this in a wrong way the system is not designed well.

To achieve the goal questions and metrics are specified, these are set out in this paragraph. Three types of questions can be recognized; questions from the interviewee (see Appendix A.3), questions resulted from literature and other questions. Questions do have a different amount of contribution to the achievement of the goal; this is made visible by use of the major 'Q' and minor 'q' numbering.

Interview questions
Interview questions are observed from the interviews that were held. For this type of questions no literature references are named. The interview questions are named in the paragraphs 4.2.1 to 4.2.3.

Literature questions
Literature questions did not result from the interviews, but are found by literature study. During the validation interviews, the software designers classified these literature questions as useful to achieve the goal. For each question stated here the literature reference(s) are given. The literature questions are named in paragraphs 4.2.4 to 4.2.6.

Other questions
This question is not explicitly found by interviews or literature, but is created according to ideas that came up while interviewing, reading literature and exploring the code-base. The question, named in paragraph 4.2.7, was also classified as useful during the validation interview.

4.2.1 Which applications are using the file that needs to be modified (Q1)?

Files that depend on the file that needs to be modified are possible risks. These dependencies are mainly important to identify the files that have to be tested. When there is an overview of the applications (obj, exe, dll) that use the modified it is possible to identify the executables that include this file. When there are a large number of applications that use the modified file the risk is higher. Thereby it is important to analyze if the application is within the group or outside the group of the software designer. When the application is outside the software designer' group responsibility the risk increases, and communication is needed with another group. There is a distinction between direct and indirect applications. Direct applications use the file that needs to be modified and indirect applications use the direct application. In this manner the direct applications are the most important to take into account.

Inbound
M1.1 List of direct object (obj), executable (exe), dynamic link library (dll) files that use the file that needs to be modified
M1.2 List of all obj, exe, dll files that use the file that needs to be modified

Summary statistics
M1.3 The number of obj, exe, dll files outside group' responsibility
M1.4 The number of obj, exe, dll files under group' responsibility
M1.5 The total number of direct obj, exe and dll files that use the file that needs to be modified

4.2.2 Which syntactic dependencies are there with the file that needs to be modified (Q2)?

Files that depend on the file that needs to be modified are possible risks. This question identifies syntactical couplings. Syntactic couplings are all the relations that can be extracted from a source-file. The couplings are analyzed at three levels; file, method and attribute level. File level consists of method and attribute couplings. The inbound couplings are the most important, because when a modification is made these are the files that will suffer the most consequences. Therefore three levels of coupling are used for inbound dependencies, in contradiction to only one level for the outbound dependencies. Thereby the modifier of the file mostly knows outbound dependencies, because these can be found in the source-code of the modified file. While inbound dependencies (the customer files) are not to be found in the source-code of the modified file, and are due to this fact less known.

A closer look at the two types of dependencies, inbound and outbound, is given in Figure 4. A change in component D will have more causes for the components A, B and C than for component E, F and G. Because when component D is modified, this can only cause changes in component A, B and C.

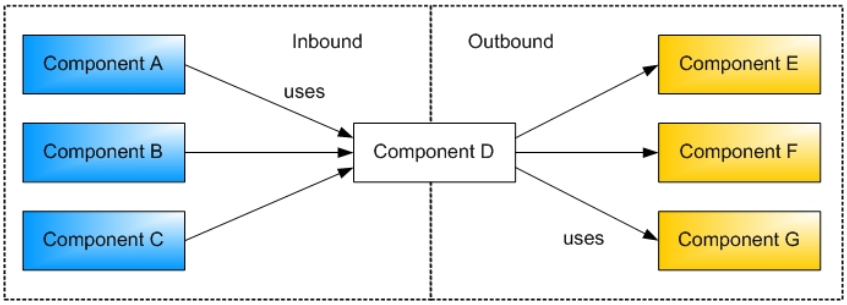


Figure 4 Dependency types

Inbound M2.1 List of files that use the file that needs to be modified outside group' responsibility M2.2 List of files that use the file that needs to be modified under group' responsibility M2.3 List of methods that use the file that needs to be modified M2.4 List of attributes that use the file that needs to be modified *Outbound* M2.5 List of files that are used by the file that needs to be modified outside group' responsibility M2.6 List of files that are used by the file that needs to be modified under group' responsibility

4.2.3 What are the customers of the interfaces that I manage (q3)?

When the modification touches an interface it is important to analyze for what customers this can have consequences. The customers can be categorized in two groups; under group' responsibility and outside group' responsibility. When an interface is outside group' responsibility the impact of the modification of the file has to be analyzed more closely, and communication with another group is necessary. The number of methods that an interface has can identify how complex the interface is. The number of external interfaces in building blocks under group' responsibility will give an identification of the context in which the to be touched interfaces have to be seen.

Direct M3.1 The visibility of the interface (e.g. ext or inc directory)

M3.2 The number of methods that the interface has
M3.3 The number of external interfaces in building blocks under group' responsibility

4.2.4 What is the amount of coupling with other files/classes (Q4)?

When the file that needs to be modified has a high coupling with other files there is a higher risk during modification. The amount of coupling can be used to weight the dependencies that exist. Four of the metrics here are only applicable to object-oriented languages. Files are coupled when methods from one-file use methods or variables defined in the other file. The metrics proposed for this question are mainly based on [Chidamber1994]. In [Chidamber1994] a metrics suite is proposed which is referred often, for example by [Lindroos2004].

The first six metrics proposed here are related to coupling between files or objects. These are important to be taken into account for three reasons according to [Chidamber1994];

1. Excessive coupling between objects is bad for reuse and modular design.
2. Improve modularity. The larger the number of couples, the higher the sensitivity to changes in other parts of the design.
3. Determine how complex the testing of parts of the design is.

In the context of this goal the last two reasons are the most important, because possible changes or to be tested parts have to be identified for this goal. The specified metrics make a distinction between inbound and outbound coupling. The inbound coupling is the most important to address, cause this has the most impact on the coupled files (see 4.2.2).

Inbound
M4.1 The number of files that use the file that needs to be modified outside group' responsibility
M4.2 The number of files that use the file that needs to be modified under group' responsibility
M4.3 The total number of files that use the file that needs to be modified (known as fan-in)

Outbound
M4.4 The number of files that are used by the file that needs to be modified outside group' responsibility
M4.5 The number of files that are used by the file that needs to be modified under group' responsibility
M4.6 The total number of files that are used by the file that needs to be modified (known as fan-out)

The last four metrics look at the Depth of Inheritance Tree (DIT) and the number of children for a class. These metrics can only be used for object-oriented languages (e.g. the C programming language is out of scope). The DIT metric has to be taken into account for three reasons according to [Chidamber1994];

1. The deeper the class is in the hierarchy, the more methods inherited, and the more complex it becomes to predict its behavior.
2. Deeper trees introduce more complexity, cause more methods are involved.
3. The deeper a class, the greater the potential reuse.

The first two reasons are most important reasons to look at the depth of inheritance tree in context of the goal. It is important to identify what consequences the modification might have for other methods; the depth of inheritance tree metric can help identifying this.

The number of children can be used for;

1. Greater the number of children, greater the reuse.
2. Greater the number of children, the greater the probability of improper abstraction.

3. The number of children gives an idea of the potential influence a class has on the design.

The third reason is the most important to take into account for this goal, cause a modification in one file can involve consequences for the children of this file.

OO specific
M4.7 The number of child classes
M4.8 The number of indirect child classes
M4.9 The number of parent classes
M4.10 The number of indirect parent classes

The proposed OO-metrics can be visualized as shown below in Figure 5.

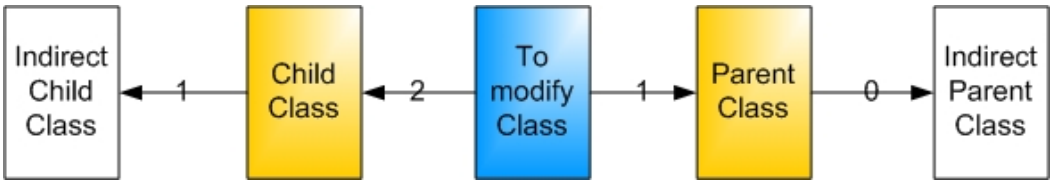


Figure 5 Amount of coupling OO-metrics

4.2.5 Which files are logically coupled (Q5)?

Logical coupling is defined as follows; two entities (e.g. files) are coupled if modifications affect both entities over a significant number of releases [Ratzinger2005]. Logical coupling is being discussed in chapter 5 Prototype. The usability and feasibility of this question are examined by the prototype. Although the metrics that are used that answer the question are named below.

Direct
M5.1 List of files under group' responsibility that are logically coupled to the file that needs to be modified
M5.2 List of files outside group' responsibility that are logically coupled to the file that needs to be modified
M5.3 The number of files under group' responsibility that are logically coupled to the file that needs to be modified
M5.4 The number of files outside group' responsibility that are logically coupled to the file that needs to be modified
Indirect
M5.5 The matching percentage

4.2.6 What is the complexity of the file that needs to be modified (q6)?

When a file has a high complexity there is a greater chance of risks to be introduced during modification. Thereby it will cost more effort to understand the file. Four metrics for identifying the complexity of a file have been selected. Two direct metrics and two indirect metrics, the indirect metrics are calculated from direct metrics.

The average lines of code per method in a file will identify how much effort and time it will cost to modify the file. In [Saboe2001] is observed that the lines of code correlate with several complexity measures, for example with the cyclomatic complexity measure. Cyclomatic complexity is addressed by [McCabe1976] and is part of the Weighted Methods per Class

(WMC) metric, which is discussed later. However lines of code per method are influenced by e.g. code formatting and programming language. The lines of code counting could be done by use of checklists. The metric is quite easy extract from the code-base, and is in this way a good starting point for measurement.

The number of methods is the easiest metric to extract; counting the number method statements in the file can easily do this. The number of methods of the file that needs to be modified, compared to the average number of methods in a file of the building block can give an indication of the complexity of the file. This figure can be made more precise by use of the WMC and Lack of Cohesion in Methods (LCOM) metrics.

The WMC metric is explained by [Chidamber1994]. By WMC the cyclomatic complexity (see [McCabe1976]) of each method in a class is calculated and taken into account for counting the number of methods. The metric is proposed to be applied on object oriented languages, but can also be applied on procedural languages like C. When applying to a procedural language, the methods per file can be taken, instead of the methods per class. The three possible reasons to use the WMC metric are according to [Chidamber1994];

1. Use as indicator for time and effort is required for development and maintenance.
2. The larger the number of methods, the greater the impact on children.
3. The larger the number of methods, the more likely it is that the class/file is application specific; which is bad for reuse.

Reason number one is the most important reason that has to be addressed by this question. When the weighted methods per class/file increase, it will cost more effort and time to perform the modification.

The LCOM is the count of the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero [Chidamber1994]. The LCOM metric can be used for four reasons;

1. Cohesiveness promotes encapsulation
2. Lack of cohesion implies that the class should be split up into more classes
3. Identify bad design
4. Low cohesion increases complexity

For this question the last reason is the one that should be taken into account. When there is a low cohesion it will be more difficult for the software engineer to understand the code.

Direct
M6.1 Average lines of code per method in a file
M6.2 The number of methods in a file

Indirect
M6.3 Weighted Methods per Class/File (WMC)

OO specific
M6.4 Lack of Cohesion in Methods (LCOM)

These four metrics can be visualized together in a Kiviat graph [Saboe2001], see Figure 6 and Figure 7. By use of this presentation it is possible to analyze the four metrics by use of one representation. In the figures are the average and maximum values calculated for the files within the building block given. The complexity of a file is acceptable when the values smaller than average or just above average, as in Figure 6. When the majority of the values for a file are against the maximum the file is unacceptable, as in Figure 7.

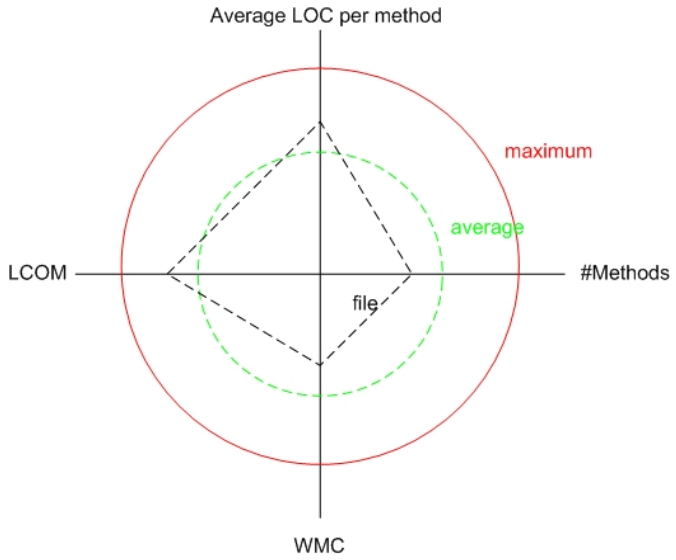


Figure 6 Acceptable complexity

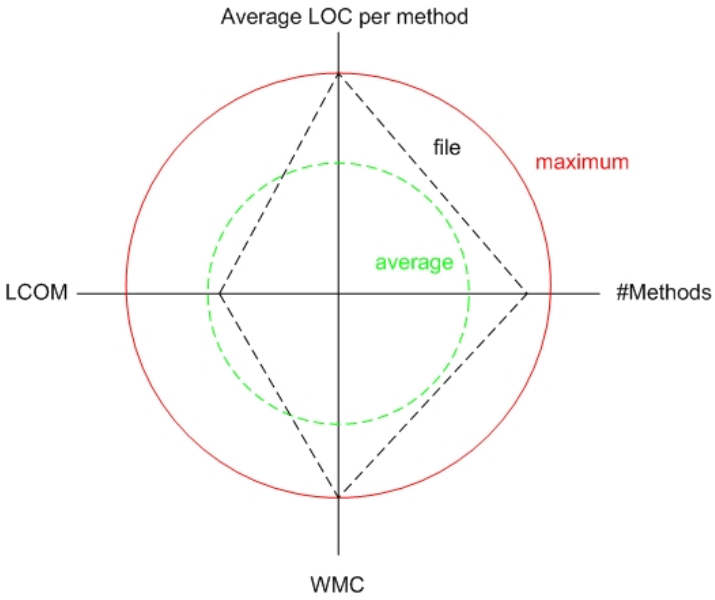


Figure 7 Unacceptable complexity

4.2.7 What are the merge dependencies for the file that needs to be modified (q7)?

A different type of dependencies has to be recognized while modifying a file; merge dependencies. It is important to perform a modification at the right stream in order to minimize the effort of coming merges. When you are working on a project, you know the project stream you are working on. But the file could also have been modified in another project, it is important to monitor this. For example, when you are modifying a file in "stream A" somebody is also working on the file in "stream B". There is no problem with the file you are modifying until you want to merge this file from stream B to stream A again (see Figure 8). When you know this file is being modified at another stream also, you can undertake some action to minimize the merge effort. For example, you can decide to perform the modification together with the other developer.

M7.1	List of stream(s) that contain other version(s) of the file that needs to be modified.

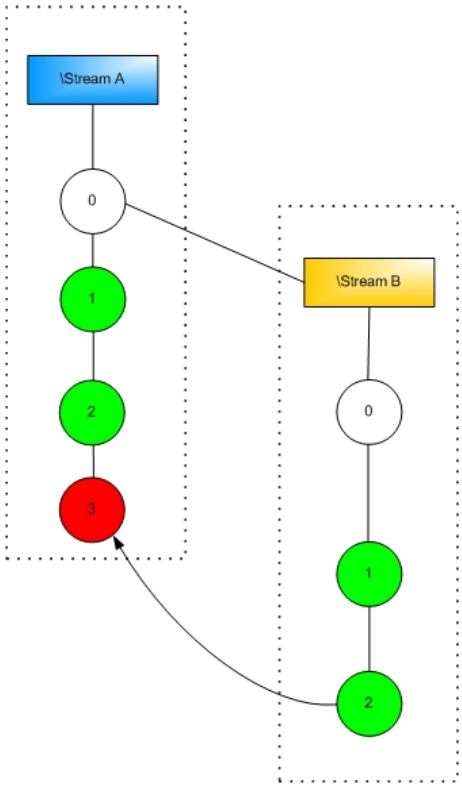


Figure 8 Merge conflict

5 Prototype

It can be concluded from the previous chapter that dependencies are important for the software designer to analyze. Dependencies can be analyzed at different levels and dimensions. Levels that can be analyzed are runtime, linktime and compile time. Dimensions that can be analyzed are inter-files and inter-building block dependencies. Thereby looking at building block dependencies, parent and child building blocks have to be distinguished.

The measurement of code has a strong relation with the program language, for example C++ methods have a different size length than Smalltalk methods [Lindroos2004]. So for each language the use of metrics have to be evaluated. Within PMS MR different programming languages (e.g. C, C++, C#) are being used, this makes it difficult to find syntactic dependencies for the complete system. Syntactic dependencies are the relations that can be extracted by analyzing the source-code. For example, "include" directives or method calls. Another way to analyze dependencies is not to look at the syntax, but at the release history of files. This dependency analysis is known as "logical coupling".

The feasibility and usability of this method for the PMS MR code-base was researched by use of a prototype. The project plan reserved three weeks of prototyping.

5.1 Concept

Logical coupling was for the first time addressed in [Gall1998]. Logical coupling can be defined as follows; two entities (e.g. files) are coupled if modifications affect both entities over a significant number of releases [Ratzinger2005].

Logical coupling can be compared with buying a book at amazon.com. When you buy a book, an overview of related books is given. Readers, of the same book, bought the related books. In other words; "people who bought this book, bought also…". In the context of logical coupling this can be translated in; "programmers who changed this file, also changed…" [Zimmerman2004].

By use of logical coupling changes can predicted, coupling can be revealed and errors due to incomplete changes prevented. When applying the syntax-based approach there are two issues according to [Gall1998];

1. Measures based on source-code are very large; at PMS MR there are approximately 6 million lines of code. Dealing with a large amount lines of code is complicated. When analyzing the release history, less data has to be analyzed.
2. Do not reveal all dependencies. For example, sometimes the software engineer just knows which set of files needs to be modified, to make a change. There is not necessarily a syntactic coupling between these files, but a logical coupling exists.

The basic idea, while analyzing dependencies, is that when more dependencies exist between modules, the less maintainable the system is. This is because a change in a module will cause changes in other modules [Gall1998].

When comparing logical coupling to syntactical coupling there is a major difference; for syntactic coupling all couplings can be retrieved from source-code. While logical coupling is retrieved by analyzing the release history of files. Logical coupling has in this manner an overlap with syntactical coupling, but identifies also non-syntactical coupling. Logical coupling suggests that components are always coupled in a logical manner, however this does

not to have be always the case. In other words some noise can exist in the retrieved logical coupling (see Figure 9).

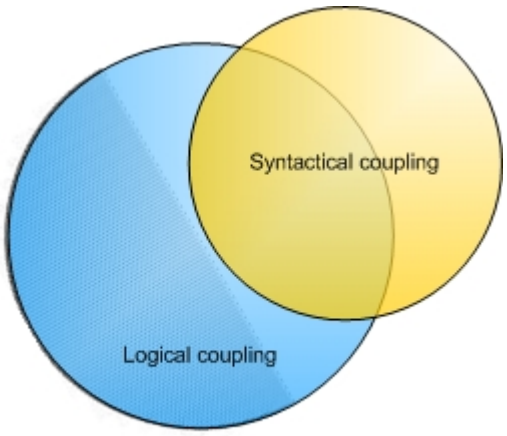


Figure 9 Logical and syntactical coupling

5.2 Implementation

Within PMS MR a file version system (Clearcase) is being used. For each project a separate stream is created. As can be seen in Figure 10 a file can have different versions. For each version a circle is created. For every stream different versions of a file can be created. When the system is built (known as consolidation), a label is created and attached to the file version that was used in the build.
The creation of a label is for example the case in stream B and file version one, by use of the label 'STREAMB_SWID1'. This label is specified on stream B and is independent from consolidation 'STREAMA_SWID1'. The consolidation 'STREAMA_SWID1' on stream A takes file version one. This version one is the result of a merge from stream B to stream A.

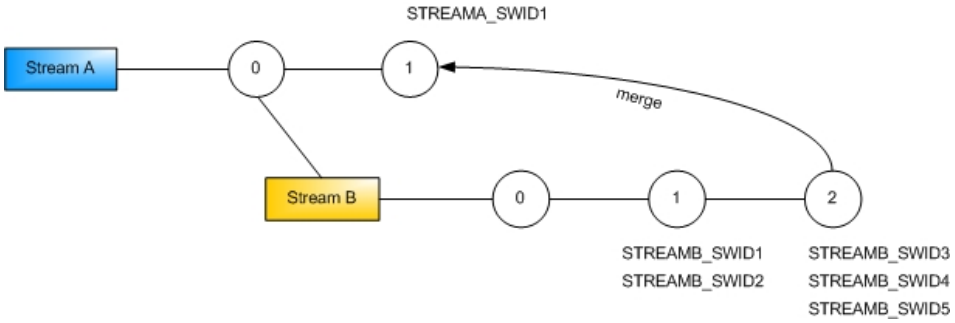


Figure 10 Clearcase process model

To determine logical coupling from Clearcase the following steps are taken;

1. Retrieve file version change
2. Construct a change sequence per file
3. Match change sequences

A file version change can be performed for two reasons; backup (check-in) of the file between posts, or a final post of the file. Only the final posts have to be taken into account by logical coupling. This can be recognized in Clearcase by the fact that a build label is given to the file. All these build labels are extracted for every file from one stream during the first step.

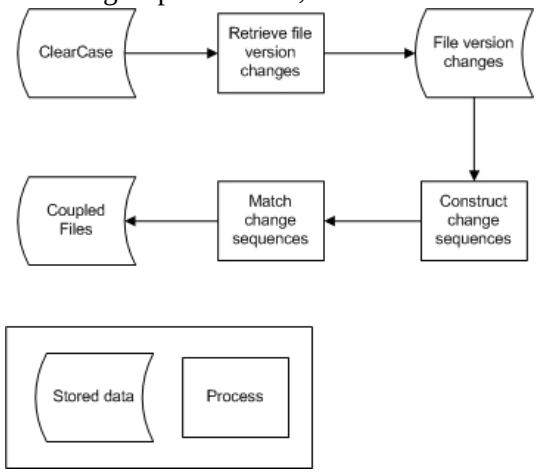


Figure 11 Logical coupling process

The second step is the construction of a change sequence for each file. A change sequence contains all build labels for a file from which the file version was changed. So when the file version number differs from the previous build, the build label is added to the change sequence of the file (see Table 3). In Table 3 "file A" changed during two builds, so two labels are added to the change sequence for "file A". In other words the change sequence becomes "SWID1, SWID3" for "file A".

Table 3 File change sequence

System build	SWID1	SWID2	SWID3	SWID4	SWID5
File A version	1	1	2	2	2
Change sequence	SWID1		SWID3		

When for each file a change sequence is constructed, the next step is to match the change sequences. An example of a match is shown in Table 4. In this example four out of five build labels match, this means a match percentage of 80%. This percentage will later be used to identify the reliability of the matching. Files with a low percentage can in this manner be filtered out.

Table 4 Coupling among files

Change sequence match = <SWID1, SWID3, SWID8, SWID10>					
File A	**SWID1**	**SWID3**	**SWID8**	**SWID10**	SWID13
File B	**SWID1**	**SWID3**	**SWID8**	SWID9	**SWID10**

In addition to the change sequence match two more techniques will be used to refine the matching; the file modifier (user) and the number of lines changed. When the same user performs the change sequence, it is more likely that the files are logically coupled. The influence of this refinement on the results will be discussed later.
For number of lines changed the hypothesis is that when a large number of lines is changed probably this will be the result of a merge. A merge does not have to be taken into account when looking for a logical coupling. And when only a few lines are changed over a large change sequence, there is a logical coupling retrieved. But it is more likely that the coupling is loose. The number of lines changed consists of all lines that where changed in the file, so comment and non-comment lines. The right way to count changed lines is out of scope for

this prototype. To conclude, the number of lines changed is used for two things; determining merges and minor changes.

The method described above has several differences with the method described by [Gall1998]. In [Gall1998] system release and program release history are used to construct the change sequence. A system release is created when a new version of the system is shipped. A program is in terms of [Gall1998] an executable from the code-archive. In the PMS MR situation, for each consolidation of the system the change sequence is constructed. Thereby instead of a program, a file is being used as entity.

5.3 Results

When populating file version changes (for src, inc and ext directories) from the whole archive about 2 million records were inserted into the database in about eight hours. This population time depends heavily on the development that is performed on analyzed the stream. The large amount of records introduces a performance problem with the database. This issue is due to problems that typically appear while prototyping. To overcome this performance problem a selection of the archive is being analyzed for the prototype. It is decided to populate file version changes for only one stream. The population for one stream takes about one hour and inserts eight times less records. To perform step 2 and 3 from the process (see Figure 11) about seven hours is needed to calculate all couplings. These steps again insert too many records into the database. To solve this issue a threshold on the change sequence length is set to three or more lines. A side effect of this decision is that not all coupling is shown, but only the more reliable ones are shown.

Not all files in the archive do have logical couplings. This is due to the fact that not all files change during a project. Therefore the files with a logical coupling are marked with three stars (see Figure 13). While analyzing logical coupling it is important to monitor what building blocks were active during the project. If there has been no or low activity in a building block, it is not useful to examine the logical coupling from that building block. Because the prerequisite for logical coupling is that there has been activity from which the history changes can be analyzed. For example in Figure 12 for the left project there has been a low activity in the first building block. And due to this fact the logical coupling retrieved from this building for this project will be less relevant. While in the right project there has been a lot of activity in the first building block. And therefore the logical coupling this time retrieved for the first building block will be relevant.

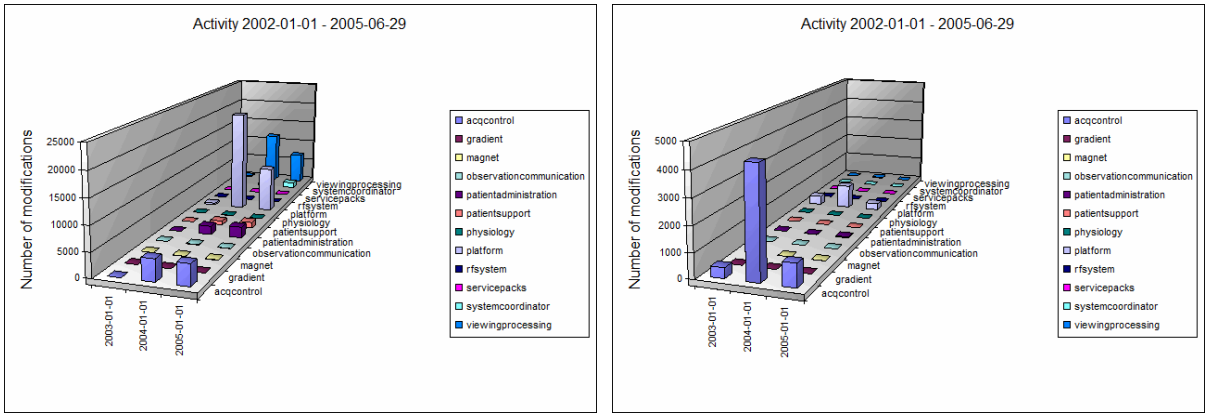


Figure 12 Activity in projects

The implementation of the logical coupling concept has resulted in a prototype that is shown in Figure 13. In this application at the left side the building block tree is listed. By selecting a building block from this tree the source-code files are shown in the right-top list box. When selecting a file from this list box, the right-middle list box will be filled with files logically coupled. For each of the files logically coupled is specified what the matching percentage, how long the change sequence is and how many lines were changed. Thereby in the right-bottom list box the building blocks logically coupled are shown. The building blocks are shown by use of counting and grouping the logically coupled files.

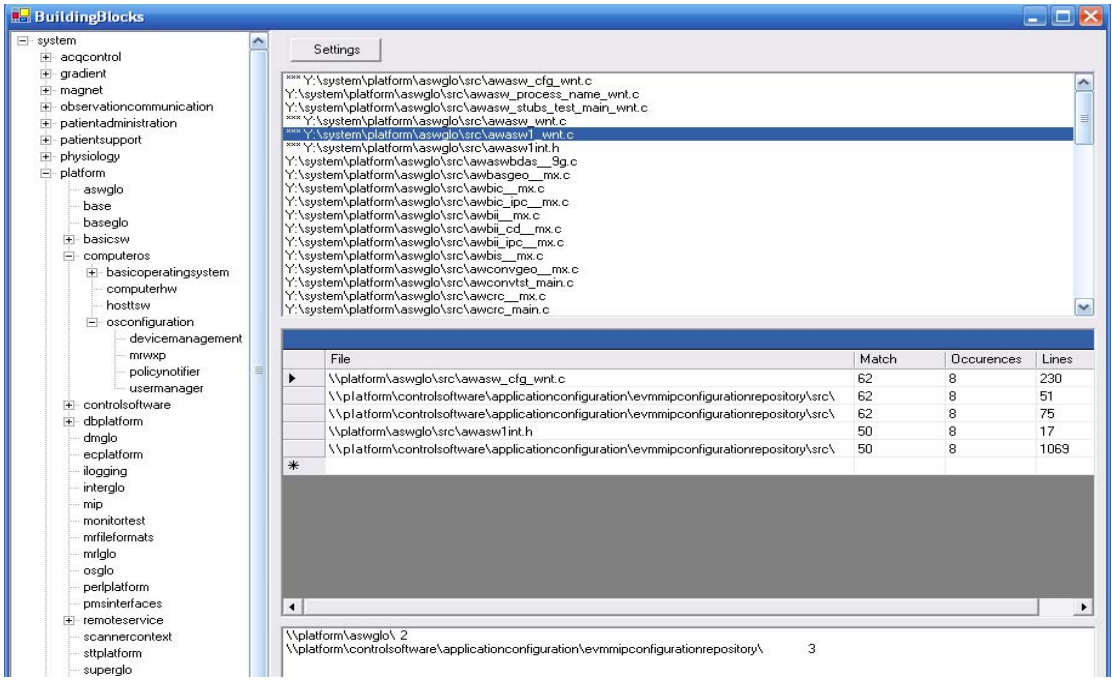


Figure 13 Logical coupling analyzer

5.4 Visualization

So far, only a textual visualization has been used. There are however interesting visualization techniques available for the presentation of coupling. The visualization method that has been implemented in the prototype is shown in Figure 14 (see also Appendix B). The implementation is done by use of [Graphviz]. In this figure all the logical couplings within a building block are visualized. For each coupling is specified how often a logical coupling has been retrieved. The main advantage of this visualization is that an overview of logical coupling per building block is available.

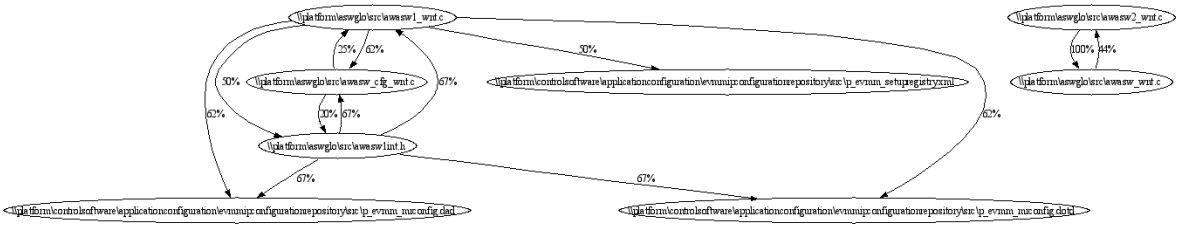


Figure 14 Visualization logical coupling

A similar visualization technique is addressed in [Gall2005]. By use of the technique described in [Gall2005], it is possible to visualize coupling at different levels by use of lens view. Hereby also the amount of coupling is visualized. This will also be a useful extension for the visualization in the prototype.

An interesting addition to the visualization currently used, is coloring syntactic the relations that can be retrieved. By making the distinction between logical and syntactical coupling explicit it will be visible to the user what couplings cannot be found by analyzing source-code. The amount of syntactic couplings retrieved differs. A small test, with two randomly selected files, resulted into four logical retrieved couplings for each file. The logical couplings for the first file did not contain any syntactic couplings. While for the second file 3 out of 4 logical couplings were syntactically.

5.5 Analysis

To assess the interestingness of the results, the gathered information was examined by use of a test case. During this test case a software designer was asked what part of the system he has knowledge of. Following, files from this part of the system with logical coupling are shown to the software designer and asked if these are 'logical' results. In other words, do the retrieved files have a relationship in some manner? And in relation with the correctly coupled files, is the matching percentage relevant. Also if the number of lines changed is useable in addition with the matching percentage was analyzed.

The results of the first test case are that there are logical coupled files shown. But there are also a lot of files shown that have no logical coupling at all with the file. Thereby the matching percentage does not play a role for identifying relevant coupled files. This is the same for the number of lines changed. The reason for these results is probably the fact that when a file is posted many times. It has a higher probability to be taken into account in a change sequence. For example, when file A changes for every build label and file B changes for only one build label; file B will have a 100 percent coupling with file A. A solution to remove these irrelevant couplings might be the use of the poster' username (e.g. the person who posts) while matching the change sequence.

When including the poster' username during the change sequence match, the assumption is made that the poster always posts files with the same functionality together. This result in a refining of the change sequence matching performed earlier. An example of this is shown in Table 5; in contradiction with Table 4 the change sequence is now three long.

Table 5 Coupling among files with poster' name

Change sequence = <SWID3, SWID8, SWID10>					
File A	SWID1	**SWID3**	**SWID8**	**SWID10**	SWID13
File B	SWID1	**SWID3**	**SWID8**	SWID9	**SWID10**
File A poster	Engineer 1	**Engineer 1**	**Engineer 2**	**Engineer 1**	Engineer 3
File B poster	Engineer 2	**Engineer 1**	**Engineer 2**	Engineer 2	**Engineer 1**

After this refinement the test case is repeated. This time the test case is showing promising results, all analyzed files are showing relevant couplings. Both syntactical and non-syntactical couplings are shown. These couplings can be named as relevant because the software designer has knowledge of the analyzed part. In contradiction with the promising results for one part of the system, performing the test case on another part of the system is not showing any usable couplings. Although the software designer has knowledge of this other part of the system.

This difference is due to several issues;

- A consolidation can contain more post lists of the same user
- Only change sequences longer than three analyzed
- Consolidations can be delayed
- Software engineers sometimes post files with different aspects onto one post list

Using a different implementation can solve the first two issues. Instead of analyzing the builds the submitted post lists can be used. This will result into a more accurate result. Secondly, solving the performance problems with the database will enable all lengths of change sequences to be analyzed. Consolidations that are delayed will be filtered out when instead of a consolidation; a post list will be used as entity. The fourth issue is process related, and requires a different style of working from the software engineer, which is difficult to implement.

The number of changed lines does not seem to be relevant. However at first sight this seemed to be an interesting figure. During the test case was observed, that sometimes the number of lines changed is very high (over 2000 lines). This is probably due to a file merge from another stream. Filtering the file merges out can help to improve the number of changed lines. Filtering merges will also contribute to retrieving more accurate couplings. It can be concluded that the extension of the change sequence matching with the poster' username is useful, but more refinement is necessary.

Logical coupling is useful for a software designer in addition to syntactical dependencies. The software designer can use the revealed non-syntactical coupled files as reminder of what have to modified or tested. Hereby it is necessary that the software designer has knowledge of the analyzed part of the system.

At the moment the process for consolidating and posting files at PMS MR is changing. In the new situation sub streams are introduced. These substreams are derived from the project stream and are used to develop a feature. During the development on this sub stream no builds are performed, however files are posted. After the completion of a feature the sub stream is consolidated to the project stream. So a consolidation is not done anymore on a regular base, but only after a merge from a sub stream to the project stream. In this new situation the current method cannot be applied anymore. It will be necessary to switch to a post list base instead of a consolidation base.

For analyzing the correct building blocks in a stream the activity within a stream was measured by means the number of lines of code modified. Thereby it is also possible to visualize the number of files modified in a stream per building block. This information cannot only be used for logical coupling, but also helps answering other questions.

An observation made during the prototype is that implementing techniques from the literature is not a simple and straightforward process. But for every specific situation it has to be investigated how they can be implemented into the organization.

5.6 Further research

The usability of logical coupling differs per building block at this moment. Some improvements have been proposed. However these proposed, and possible other, improvements still have to be researched.

At the moment we only looked at source files and header files. It would be interesting to see if the change sequence could also be adapted to document files. For example when this source file is being changed, this document has to be modified as well. This can be used as advice to

the software engineer. It may be useful for educational purposes for new software engineers who are not familiar with the code-base.

Logical coupling can only be found when files have "changed enough" during several builds. In this way it is very useful for the older parts of the code-base (C, C++). But it might be less useful for new parts of the system (C#). This possible issue is not researched yet.

At the moment coupling is only discussed at the file level, but this can also be quite easily abstracted to the building block level. By use of this abstraction it would be possible to analyze coupling between building blocks. The usability of this has to be examined.

5.7 Conclusions

The applied method is showing coupling in addition to the known syntactical coupling. The possibilities of implementing this method at PMS MR have been proven.

The implementation of the prototype resulted in a few performance problems with the database. Therefore not the whole system has been analyzed. However in this context it was sufficient to assess the feasibility and usability of "logical coupling". The prototype allows the software designer to identify the logically coupled files from a certain file. Hereby also syntactic coupled files are taken into account. Resulting from the test case, the logical couplings that are shown are sometimes relevant and sometimes not relevant. It is not possible to distinct relevant and not relevant files without knowledge of the specific system part. To improve the reliability of the results several improvements are proposed.

An important lesson learned, is the fact that the implementation of a tool is specific for every system. This is especially the case for large systems that are custom made, like Clearcase within the PMS MR organization.

Concluding can be stated the logical coupling can be a useful addition to syntactic coupling. Thereby logical coupling can be extracted quite easily compared to syntactic coupling where different programming languages have to be taken into account. The question as stated in the GQM will be feasible and usable, if further research is performed on the proposed improvements. Furthermore, the activity figures from the prototype can already be used to answer several other metrics.

6 Conclusions

In the chapters 3, 4 and 5 the results from the thesis have been presented, here the conclusions from these chapters are named. The Goal Question Metric (GQM) approach was applied to come to these results. The GQM approach contributed significantly to the structuring of the information, and the selection of the measurements. It can be concluded that it was suitable and successful to use the GQM approach.

The first objective of this thesis was to gain insight into the information that a software designer, with respect to design, needs during his work. The insight has been gained by use of literature study, interviews and a survey among all software designers. The gathered information from these interviews and survey is expressed by use of a top of three goals;

1. Perform a modification assessment
2. Make building blocks independent
3. Decrease complexity

It can be concluded that the literature study, interviews and survey were successful for the extraction of the goals that a software designer has with respect to design.

The second objective was to specify questions and metrics, which can be answered from the code-base. This resulted into four major questions to achieve the most important goal;

- Which applications are using the file that needs to be modified (Q1)?
- Which syntactic dependencies are there with the file that needs to be modified (Q2)?
- What is the amount of coupling with other files/classes (Q4)?
- Which files are logically coupled (Q5)?

The software designers agreed that these questions are useful to achieve the goal. All the major questions relate to dependencies. It can be concluded that dependency analysis is the most important topic for the software designer at the moment. The number one goal will help the software designer to make a proper decision for a modification or redesign.

By means of prototyping the usability and feasibility of the question "Which files are logically coupled?" has been examined. The prototype is showing differing results per building block and knowledge of the building block is required to assess the usability of the results. However the results that are reliable look promising and can be used in addition to syntactical dependencies.

7 Recommendations

At the beginning of the project the following question was formulated; "Which information does a software designer, with respect to design, need during his daily work to control his team?". We have answered this question by use of the GQM approach. This resulted into a top three of goals that a software designer has. The top three of goals is;

1. Perform a modification assessment
2. Make building blocks independent
3. Decrease complexity

Implementing all these goals implies that 15 questions and 46 metrics have to be implemented. This is a too large amount of questions and metrics; according to Niessink and van Vliet [Niessink] who identify the success factor "Start small and simple, increase measurement when organization and team has more experience". Therefore at first a selection of these goals has to be made. The goal "Perform a modification assessment" is by far selected as most important. This goal makes it is possible to analyze the impact of a modification in order to predict which other files have to be modified or tested. By having this overview for a file, the software designer will be able to make a proper decision for a modification or redesign. With respect to the fact that this is the most important goal, it is recommended to implement this goal at first. However this goal has seven questions, which is still a large amount to implement at once.

To limit the amount of questions that will be implemented, the contribution of the questions to the goal has been divided into two categories "major" and "minor" (see chapter 4). The software designers stated the following questions as most important;

Q1	Which applications are using the file that needs to be modified?
Q2	Which syntactic dependencies are there with the file that needs to be modified?
Q4	What is the amount of coupling with other files/classes?
Q5	Which files are logically coupled?

Question number one identifies the applications that use the file. Most of the answers to this question can currently be retrieved by use of a present application. The only metrics that cannot be answered by this application are the M1.3 and M1.4, where the software designer' group is taken into account. Due to the fact that most of the metrics are already available to the software designer, it is recommended not to implement the additional metrics for this question at first. Although question number one is rated as important to achieve the goal.

Question number two and four relate to syntactic file dependencies. At this moment the answer to Q2 is gathered by 'hand', by means of opening a source file. While a software designer does Q4 on basis of intuition, in combination with the experience he/she has. So both questions cannot be answered automatically at the moment.

Question number five seems to be an interesting topic for further research. This question shows the files that were modified together over a significant number of releases, so called "logical coupling". In other words it answers; "Programmers who changed this file, also changed…". This question is different then question number two that only shows syntactical coupling. Logical coupling has an overlap with syntactical coupling, but can identify all kinds of coupling. The feasibility and usability of logical coupling was prototyped. The most important conclusions made from the prototype are;

- Coupling is relevant
- Coupling is not always relevant
- Improvements are necessary to provide more reliable results

So at the moment question number five is not recommended to implement, but can become very useful in addition to syntactic coupling. The implementation of question number two and four are a good starting point for a tool that supports the software designer during his work. This implies that in total sixteen metrics have to be implemented. This is still a large amount of metrics, although the ten metrics stated at question number four have an overlap. For example M4.3 "total number…" is a simple addition of the metrics M4.1 "number of files… under…" and M4.2 "number of files… outside…". A tradeoff from the implementation of these two questions also helps answering goal number three "Decrease complexity". This tradeoff can accelerate the implementation process during further steps.

In other words metrics are promising; the GQM approach significantly contributed to a proper definition of the questions a software designer has. Question Q1, Q2 and Q4 have to be implemented and further investing has to be done on Q5. Hereby Q1 is almost completely answered by use of current tooling, and therefore has a lower priority during implementation.

After implementing the questions it is important to monitor the implementation, encourage the software designers to use the tooling and constantly keep improving the measurement program. Only in this manner the measurement program can become successful within the organization.

To summarize, the measurement program has to;

- Start small and simple.
- Encourage software designers to use its information.
- Generate value for the organization.

8 Evaluation

The final remaining question is; "What went following plan, and what could have be done better?". The answer to this question will be given in this chapter by use of four paragraphs.

8.1 Project plan

The first task in the project plan was to perform a literature study. The literature study was finished in time and contributed to the final result. When actual starting in the practice the timeline in the project plan has changed several times. This was mainly due to the fact that it was more efficient to perform tasks like, interviewing and report writing, parallel and not sequential. Thereby has to be mentioned that the interviews took more time than was planned originally. In the first version of the project plan the writing of the GQM and recommendation report were separated. In practice it turned out to be more less the same activity. In the project plan three possible risks were named. None of the possible risks resulted into a problem. So it can be said that the timeline has been changed sometimes, but finally the project has been finished within time.

8.2 Interviews

During the first series of interviews the ERD (see Appendix A.1) has been used as guideline. The use of this diagram resulted in a good structure and overview for the interviewer and interviewees. Although afterwards was observed that the interviewed software designers all had a lot of questions, compared to the other stakeholders. The group of other stakeholders consisted of a software engineer, segmentleader and software architect. The small amount of answers from this group is probably due to the fact, that they have less relation with the used ERD. A conclusion that can be made is that the ERD was very useful to gain information during the interviews with the software designers. The next time it might be more useful to structure the interview with other stakeholders differently. So more information will be gathered during interviews with other stakeholders. Although on the other hand can be said that the other stakeholder do not have more questions than they stated now.

During the second interview series "validation", the interview approach was to ask only closed questions to the interviewees. For example the question "do you think this is a useful metric?". After this question a follow-up question like "Why do you think this metric is useful?" was asked. This resulted into a good overview of the opinions that the interviewees had. It can be said that this approach was successful and can be used again.

8.3 Mismatch literature & practice

When reading the literature, the GQM process is described as a pretty straightforward process. In this process the right goals are defined at once, and the right questions and metrics are specified after this. At the start of the project this was also the intake for the GQM analysis. After the first interview series, goals were formulated. These goals were prioritized by use of a survey. But while performing this survey we realized that not all the goals were at the same level. Some of the goals were more abstract then others and some were not goals but more questions. Due to these facts a reformulation of the goals was necessary. This unexpected observation occurred because of applying the GQM approach for the first time. It can be said that this was an important lesson learned during the project.
Thereby no GQM reports were found on the area of software measurement with respect to the code-base. This resulted in the fact that the right abstraction, for the questions and metrics had to be found by experimenting. To see if the right abstraction is achieved and questions, metrics were clearly stated a validation interview series has been done. Relating to the fact

that all software designers agreed upon the stated questions and metrics, this has been successfully.

8.4 Follow up

The interpretation phase of the GQM analysis has been completed, to complete the GQM analysis process the measurement program needs to be implemented. We created some support by the software designers by involving the software designers as much as possible in the project. Although the management has to decide if the GQM analysis will be completed. At the moment, a part of the management stated that the selected goals look promising. Concluding from this statement a follow up seems to be initiated, although no assurances are given yet.

9 Terms

Term	Description
Building Block	A building block corresponds one-to-one with the directories in the code-base. Building blocks have different levels. The highest-level building blocks (subsystem) are identified by not having parent directories, and the lowest level building blocks are identified by not having any child directories.
Child class	A class that is derived from a particular class, perhaps with one or more classes in between. Also known as sub or derived class (see Figure 15).
Clearcase	File version control system used within Philips Medical Systems MR.
Component	A component can be a building block, file, class, or method entity. The meaning of the component has to be retrieved from the context.
Consolidation	A consolidation is performed when the code-archive for a certain project is build or tested successfully. A SoftWare Identifier (SWID) specifies a label for a build.
Cyclomatic Complexity	Measures the number of linearly independent paths through a program module [McCabe1976].
Depth of Inheritance Tree	The maximum length from the class node to the root of the tree, measured by the number of parent classes. If multiple inheritance is allowed, the longest path is taken.
Fan-in	Fan-in counts the number of references that use the attribute declarations, formal parameters, return types, throws declarations and local variables, and types from which attribute and method selections are made. Primitive types are not counted.
Fan-out	Fan-out counts the number of references that are used in attribute declarations, formal parameters, return types, throws declarations and local variables, and types from which attribute and method selections are made. Primitive types are not counted.
GQM	Goal Question Metric, a widely used method for software measurement.
Inbound	All connections that are made by components, which are using a selected component (see Figure 4).

Indirect child class

Class C is an indirect child class of class A if class C is a child class of class B, and class C is not identical to class B. So there must be at least one other class B that is a parent class of class C and a child class of class A (See Figure 15).

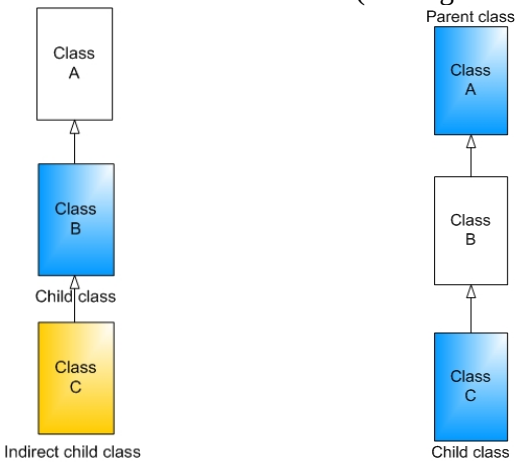


Figure 15 Parent-, child- and indirect child class definition

Interface

An interface provides the functionality, but does not contain the implementation itself. An interface is a file that is located in the 'inc' or 'ext' directory, or is a header file located in another directory in the code-base.

Lack of Cohesion in Methods

This is the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero [Chidamber1994].

Outbound

All components that are used by the selected component (see Figure 4).

Weighted Methods per Class

The sum of the complexity of methods of a class (WMC). For each method the cyclomatic complexity is calculated. Following WMC is the sum of the cyclomatic complexities per method. This can also be done per file.

10 Literature

[Anacletto2003]
Alessandra Anacletto, Teade Punter, Christiane Gresse von Wangenheim, GQM-Handbook and Overview of GQM-plans, A comparison of three GQM-methods and an Overview of GQM-plans, 2003, 55p.
http://www.iese.fraunhofer.de/pdf_files/iese-008_03.pdf
In this report the history and rationale of the Goal Question Metric (GQM) approach are described. Then a comparison of the three GQM approaches is made by the use of mentioning the similarities and differences. The main part consists of how to successfully apply the GQM approach.

[Chidamber1994]
Shyam R. Chidamber, Chris F. Kemerer, A Metrics Suite for Object Oriented Design, 1994, 18p.
http://www.pitt.edu/~ckemerer/CK%20research%20papers/MetricForOOD_ChidamberKemerer94.pdf
The research in this paper is concentrated on the development of a new suite of metrics for OO design. Six metrics are developed and evaluated against a proposed set of measurement principles. A tool was developed and implemented to demonstrate the feasibility and to demonstrate how the metrics can be used. The proposed metrics proofed to be platform independed, the results gathered from the metrics give an indication of the integrity of the design and can be used to identify areas that need extra attention.

[Daskal1992] Michael K. Daskalantonakis, A Practical View of Software Measurement and Implementation Experiences Within Motorola, 1992, 13p.
In this paper the implementation of a software measurement program within Motorala is described. This is implementation is done by the use of the Goal Question Metrics approach. In this metric program 10 metrics are used to support the software manager and engineer. The important lesson learnt from this implementation is that it is better to start with metrics and let these evolve overtime than discuss about these forever. Thereby it is important to remember that metrics can only give an indication of what action can be done. The actions taken are the real results.

[Gall1998] Harald Gall, Karin Hajek, and Mehdi Jazayeri, Detection of Logical Coupling Based on Product Release History, 1998, 10p.
http://www.infosys.tuwien.ac.at/Staff/mj/Papers/icsm98.pdf
In this paper a proposal is made for a method to find logical dependencies between subsystems. To find these dependencies different versions from the system and it's containing systems are evaluated. The logical coupling resulting from this evaluation is refined by use of textual pattern matches on change reports. This method shows promising results for identifying possible couplings among modules. In this way normally hidden dependencies between modules are revealed.

[Gall2005] Harald Gall, Jacek Ratzinger, and Michael Fischer, EvoLens: Lens-View Visualizations of Evolution Data, 2005, 10p.
http://www.infosys.tuwien.ac.at/Teaching/Courses/SWE/papers/ECoVis.pdf
Making software evolutionary aspects explicit by use of visualization can help the software engineer by identifying design smells. Only very few tools allow software engineers to view a system through a kind of lens view. In this paper a visualization method is described that allows software engineers to look through a lens to the software evolution. The software evolution is made visible in means of logical (or change) coupling and size measures. This information is automatically extracted from CVS.

[Graphviz] http://www.graphviz.org/
Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. Graphviz is open source graph visualization software.

[Lindroos2004] Janaa Lindroos, Code and Design Metrics for Object-Oriented Systems, 2004, 10p. http://www.cs.helsinki.fi/u/paakki/Lindroos.pdf
There are quite a few metric sets that are specially set up for OO software. In this paper a set of 6 metrics is presented and the usability of these metrics is validated by a few real-life projects. The conclusion consists of the fact that the metrics are suitable to use in all OO software projects. But the addition of other metrics is possible.

[Martin2003] Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, 2003, 529p.
In this book Martin describes his view on the agile development process. For this research only the chapter 'design smells' is used. In this chapter the rationale and description for design smells are given.

[McCabe1976] Thomas J. McCabe, A Complexity Measure, 1976, 13p.
http://www.literateprogramming.com/mccabe.pdf
The well-known McCabe complexity is set out in this paper. It is illustrated how this can be used to manage and control program complexity. But no figures are given on what is an acceptable or emerging complexity.

[Niessink2001] Frank Niessink, and Hans van Vliet, Measurement Program Success Factors Revisited, 2001, 20p.
http://www.cs.vu.nl/~hans/publications/y2001/revisited.pdf
Niessink and van Vliet state that not only internal success factors for measurement programs have to be taken into account. But besides these factors, external success factors have to be recognized. For a measurement program to be successful it has to generate value for the organization. In the paper five case studies are evaluated by use of 15 internal success factors and 4 external success factors.

[Ratzinger2005]
Jacek Ratzinger, Michael Fischer, Harald Gall, Improving Evolvability through Refactoring, 2005, 5p.
http://msr.uwaterloo.ca/msr2005/papers/21.pdf
To identify "bad smells" in the software design they propose to use logical coupling. Before and after refactoring software parts are analyzed. The advantage of the use of logical coupling is that it also reveals "hidden" dependencies. These hidden dependencies, not evident from the source code, can identify structural weaknesses. They conclude that, after observing a system for 15 months, the logical coupling approach helped improving the maintainability and evolvability of the system.

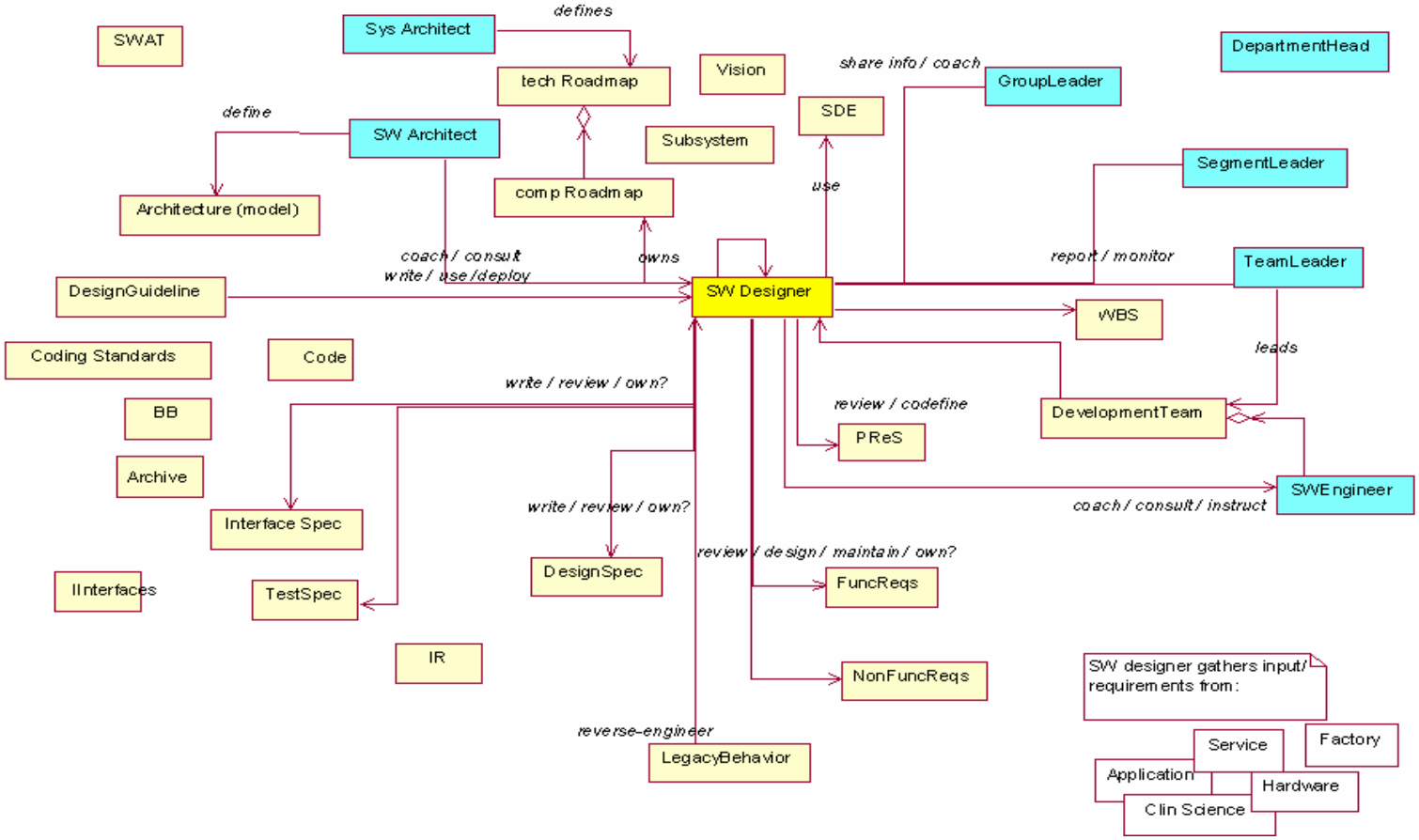
[Saboe2001] Michael Saboe, The Use of Software Quality Metrics in the Materiel Release Process Experience Report, 2001, 6p.
In this paper metrics are proposed to measure the software maintainability. The proposed metrics are; cyclomatic complexity, halstead measures and maintainability index. The metrics maintainability index, total lines, cyclomatic complexity and halstead effort are combined into a graph to identify which modules are more likely to contain defects. Also a graph presentation, to identify unacceptable modules by use of 4 metrics is proposed.

[Veenendaal2000]
Erik van Veenendaal, Mark van der Zwan, GQM based Inspection, 2000, 6p.
http://www.improveqs.nl/pdf/GqmInspecties.pdf
In this paper the importance of software inspections is described. Software can play an important role in the quality improvement of software, this is widely accepted. However software inspections are not a standard practice in projects. Introducing inspections is a difficult task, for this task the GQM approach is used. The GQM approach will help to focus on the data gathering process, and support the interpretation process.

[Zimmerman2004]
Thomas Zimmerman, Peter Weissgerber, Stephan Diehl, Andreas Zeller, Mining Version Histories to Guide Software Changes, 2004, 10p.
http://www.st.cs.uni-sb.de/papers/icse2004/icse.pdf
In this paper data mining of version history is proposed to predict which files will change during the modification of a file. They compare the principle of data mining of version history with buying a book at amazon.com; "Customers who bought this book, bought also…". They propose to apply the same concept to programming. While you are editing a method within a file, a suggestion is given of methods that also have to be changed. For the top three suggestions a likelihood of 64% is achieved.

Appendix A Interviews

A.1 Entity Relation Diagram – SWAT Workshop



A.2 Raw results

The raw results from the first interview series are stated in the table below. The questions are grouped by entity from the SWAT workshop Entity Relation Diagram (ERD), see appendix A.1. For example during the interview software designer "DH" stated that he has by the entity "Building block" from the ERD, the question "What functionality does a building block offer?". The software designers "BB", "JWD" and "GW" also stated the same question. This results in a cumulative of 4 for this question.

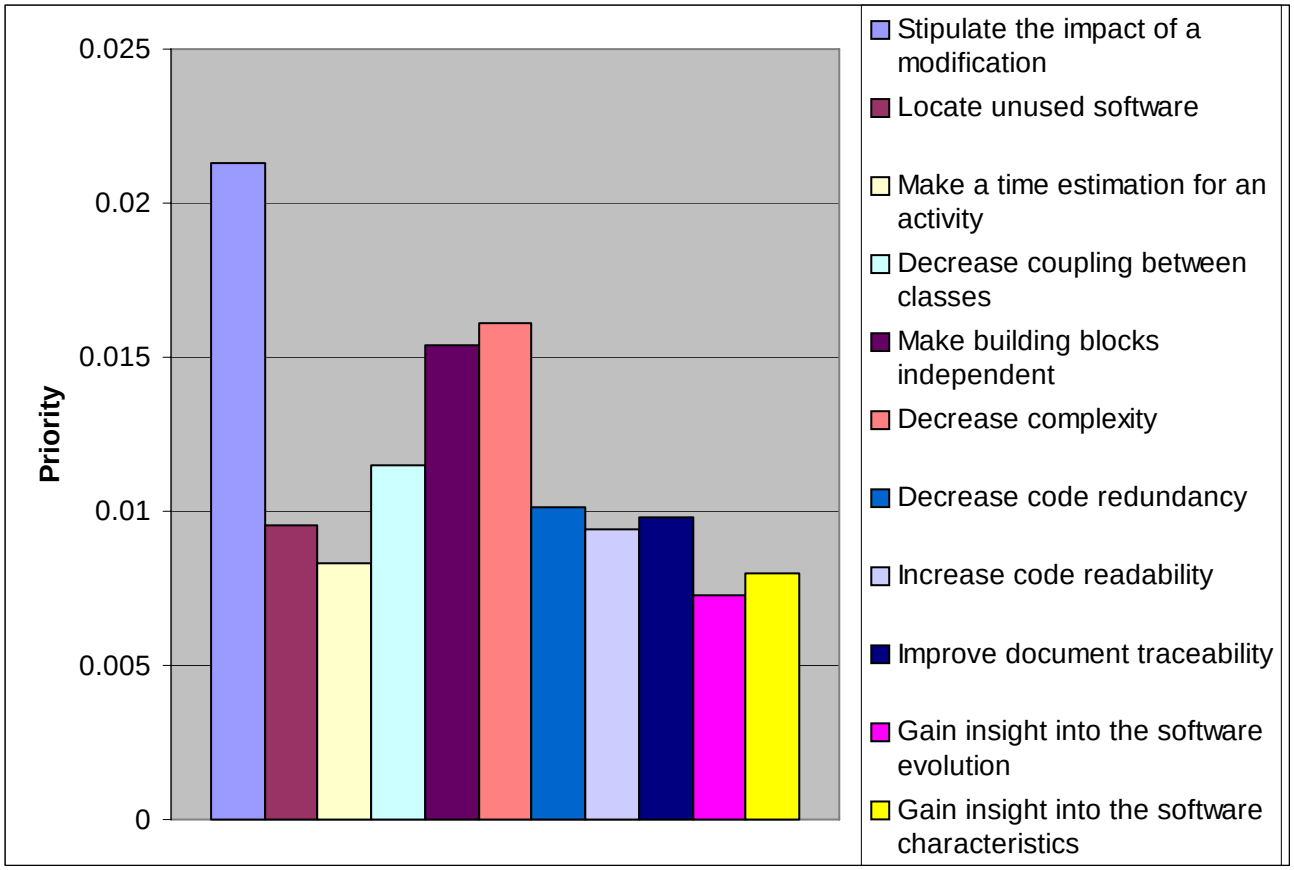
#	Questions	DH	BB	JWD	IO	WM	RG	BV	GW	JB	Cumulative
	Building block										
1	What functionality does a building block offer?	x	x	x					x		4
3	Where to find the functionality in a building block?	x	x								2
4	Where is which building block to find?	x									1
5	What applications are using the building block?	x			x	x	x		x		5
6	Where is a building block located in the hierarchy? (in relation to design, req specs documentation)	x		x							2
7	How often is a building block used?	x									1
8	How often is a building block modified?	x				x	x				3
9	What dependencies are there between building blocks?	x	x	x	x	x	x	x	x	x	9
10	Where is a building block located in the architecture?	x		x							2
11	What is the relation with the number of PR's and modifications in building blocks?	x					x				2
12	What are the metrics of a building block? (e.g. LOC, classes, comment)	x	x			x					3
13	What building blocks have which group under account?		x		x	x			x	x	5
14	Where is the gravity point of a building block?		x			x					2
15	What parts of a building block are not documented?		x	x							2
16	Who is using building blocks under my responsibility?		x		x	x	x				4
	Code										
18	What is the quality of a source file?	x				x					2
19	What is the testcoverage of a file?	x			x	x					3
20	What is the runcoverage of a file?	x	x			x					3
21	Where is duplicate code to be found?	x	x		x	x	x	x	x		7

#	Questions	DH	BB	JWD	IO	WM	RG	BV	GW	JB	Cumulative
24	How often is a file used?		x								1
25	What are the relations that a file/class has?		x	x		x	x				4
	Archive										
26	On what stream do I have to perform a modification to make it visible for everyone?	x									1
27	On what stream are modifications performed for a certain building block?		x								1
28	How much code is added/deleted/modified?		x						x		2
29	What is the size of a modification?		x			x					2
	Interfaces										
30	Which interfaces do I manage?	x	x		x	x	x		x		6
31	What are the potential users of my interfaces?	x									1
32	What are the current users of my interfaces?	x									1
33	How often does an interface change?		x								1
34	What is the width of an interface?		x	x		x	x				4
35	What exceptions can occur?			x							1
	Design guidelines										
36	What design guidelines are there?	x			x						2
37	What design guidelines are applicable on my task?	x	x		x						3
38	Where is which guideline used?		x		x						2
	Coding standards										
39	Does the number of code violations increase?	x	x		x		x				4
40	What is the type of increase code violations?	x			x						2
41	How many violations are there within a building block?		x		x						2
42	What building blocks don't have to meet coding standards?		x				x				2

#	Questions	DH	BB	JWD	IO	WM	RG	BV	GW	JB	Cumulative
	Design specs										
43	What is the relation between design and code?	x	x	x	x	x		x			6
44	Does the code meet the design?	x		x	x						3
45	On which streams is the most recent spec located?		x								1
46	How often is a spec changed?		x								1
47	What is the state of a document?		x								1
48	What is the size of a document?		x								1
49	What documents does the group manage?		x								1
50	Is the design scalable?			x							1
51	What design patterns are being used?			x							1
	Legacy behavior										
52	What is the design of the code?	x									1
	Requirements specs										
53	How many requirements are implemented in the design?	x				x	x				3
54	What is the relation between requirements and tests?		x		x	x					3
	Test specs										
55	How long does it take to execute a test?		x								1
	Pres/WBS										
56	What projects can I expect?	x									1
	Teamleader										
57	Which designer is the most suitable to implement a certain part of a project?	x									1
58	What is functionality is implemented on the wrong place?	x									1
59	Where is functionality already implemented?	x			x						2
60	How much work it is to implement a certain functionality?	x			x	x					3

#	Questions	DH	BB	JWD	IO	WM	RG	BV	GW	JB	Cumulative
	Software engineer										
61	What I have to make? (from the viewpoint of the software engineer?)	x									1
62	What is the vision of the software engineer?		x								1
63	What is the spent time of a software engineer?		x								1
64	What is the expertise of the software engineer?		x								1
	Software architect										
65	What are the directions to go? (from both viewpoints)		x								1
66	What is the roadmap of a software designer?		x				x				2
	Component roadmap										
67	What is the configuration of a component?	x									1

A.3 Survey results



Calculation

$$\text{Priority} = 1 / \left(\sum^{\#persons} rank \right)$$

Appendix B Prototype coupling visualization