The Software Project Leader’s Cockpit

Master’s thesis

Jonne Kats

Royal Philips Electronics

Friday, 15th of July, 2005

“1 year Master Software Engineering”

Mentor UvA: P. Klint

Mentors Philips: L.Hofland and R.Krikhaar

Version 1.0

Hogeschool van Amsterdam Universiteit van Amsterdam Vrije Universiteit

Abstract

The software project leaders at the Philips Medical Systems - Magnetic Resonance development department collect numerous of measures. However, from one of their most important business assets: the code base, they do not collect much. This thesis describes the results of the definition of a measurement programme, which goal is to define measures that can be extracted from the code base and can help the software project leaders in managing their projects.

Implementing the measurement programme will cost effort and therefore it is important to know why there is being measured. The Goal-Question-Metric approach helps in defining these goals. The requirements on the measurement programme have been gathered by the use of interviews, the results of these interviews have been structured using this Goal-Question-Metric approach. This resulted in measures that have the following goals, improving project control and reducing the number of faults discovered in the field.

During the project research was done to measures that are being used by other companies. The most of these measures focused on bug reports, time sheets, effort estimations, etc. Other research was done to support ideas resulting from the Goal-Question-Metrics analysis. The research was related to the results from the Goal-Question-Metrics analysis in a GAP analysis, most of the goals were supported by the literature except for two goals. However, these goals could be supported by normal logic and did not need additional literature. Prototyping was used to determine the feasibility and usefulness for some of the measures, a metrics called entropy appeared not to be applicable at the MR organization. A merge estimation measurement and an activity parameter appeared to be useful.

Implementing the measures at the MR department will require effort, because it has to be made part of the daily process. However, implementing the measures resulting from this project require minor effort compared to the benefits they deliver. Additional research should be done to ideas that could possibly improve the measures that resulted from this project, but a working tool implementing the measures should not take to much effort.

Index

1 Introduction

3.1 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.

3.2 Background

The software project leaders at Philips Medical Systems – Magnetic Resonance (MR) measure a lot when it comes to project management; hour registration, bug reports, etc, but when it comes to the business object in which all the work takes place: the code base, they do not collect much. Because of this, the software project leaders have no overview of what is taking place in this code base and it is hard to make decisions about the current project state.

1.1 Research question / objectives

During the different phases of development a software project leader (PL) requires different information to monitor the project status and progress. A great deal of this information can be gathered from the configuration management system, which provides historical data of past projects. This information needs to answer certain questions, for example: how many files have changed between releases, how many files have changed last week, how many lines of code have changed last week, etcetera.

Getting answers to these questions is a time-consuming process, therefore it would be ideal to have a tool, which would automatically extract the information that answers these questions from the code base and present them to the project leader.

The objective of this assignment is to define all kind of information a software project leader requires in different phases of a development project with regards to the code base. One can see this as gathering (functional) requirements for a tool that visualizes daily status information of the project. The research question can be formulated as follows:

"What information can be extracted from the code archive that is useful to the project leader for managing his projects?"

Next to the elicitation of the requirements, research will be conducted to techniques for extracting the needed information from the code base and to additional information that can be extracted from the code base that the project leader at MR did not know of and could use for managing his projects.

The main objectives for this project are:

- Define the requirements that the software project leader has on information that can be extracted from the code base

- Describe additional measures and techniques that can be of use to the software project leader

- Determine if the most important measures are feasible by the use of a prototype

1.2 Deliverables

This will be the deliverables for the project:

- **GQM Analysis report**
  The results of the interviews will be presented in a Goal Question Metric (GQM) analysis, this GQM approach will be described in chapter 4.1.1.

- **Prototype**
  This prototype will be developed continuously during the project and has the purpose of giving the stakeholder a glimp of the final tool and to experiment with some of the techniques resulting from the literature study and research. It will give insight in the possibilities and functioning of some of the techniques that were discovered during research.

- **Research report**
  During the literature study, research will be conducted to proven techniques concerning the analysis of a code base. The goal of this research is to find out what activities the software project leader has to perform, what information he needs in order to manage a project and what of this information can be extracted from the configuration management system. These are the initial keywords the research will focus on: Status Accounting, Software Configuration Management (SCM), Development streams, SCM patterns, Merge policies, Stream policies etc.

1.3 Outline

Chapter two covers background information about the *organization* at MR and the development process. Chapter three describes the *approach* that is followed when this project was executed. The fourth chapter, *research,* covers the literature and research that is done during the project. In chapter five, the results of the *GQM Analysis*, are shortly described.

In chapter six, analysis is done to the results of the GQM Analysis in *relation* with the research conducted. A part of the ideas resulting from the research and the GQM analysis are validated for feasibility and usefulness by developing a *prototype*, which is described in chapter seven. The *conclusion* in chapter eight contains the findings of this work. In chapter nine, *recommendations* are done about future implementations regarding the measures. The project is *evaluated* in the final chapter.

1.4 Acknowledgements

I would like to thank Philips Medical Systems for making this project possible, L. Hofland, R. Krikhaar and B. Buunen for the contribution of their thoughts and ideas. I also thank P. Klint for his feedback on this thesis.

2 Organization

The software development department of the magnetic resonance division counts over hundred people. Their main focus is on developing software that drives the Magnetic Resonance scanners. The source code is mainly written in C and C++ and has been around for many years, the systems counts over five million lines of code. New parts of the system, especially the user interface parts are being redesigned and rewritten in C#. In this chapter the development process used to develop the MR software will be described and the configuration management process will be highlighted.

2.1 Development process

The current project life cycle has the following phases:

1) **Project feasibility**
The high-risk requirements are being investigated and the feasibility of those requirements is determined.

2) **Requirements analysis, overall design**
During this phase the project content is determined and the required time and resources are being specified. A software design is created and activity descriptions are written.

3) **Design, Implementation & Integration**
The product is being developed including documentation and validation. The α test plan is also written.

4) **α Test, product verification**
The product is being verified using the α test plan, problems are being corrected and a β test plan is written.

5) **β Test, product accreditation**
The product is being acrreditated using the β test plan and problems get corrected. The product is released and the project gets evaluated.

6) **Product release, maintenance**
The product is released and gets deployed to customers, the product gets into maintenance.

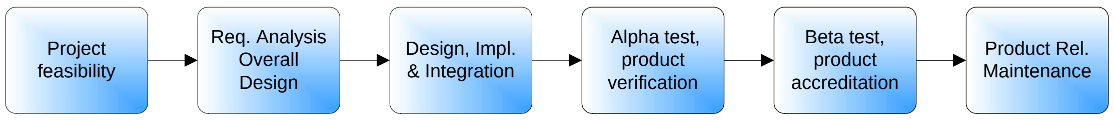

Project feasibility
Req. Analysis Overall Design
Design, Impl. & Integration
Alpha test, product verification
Beta test, product accreditation
Product Rel. Maintenance

Figure 1: Project Life Cycle

2.2 Software Configuration Management

Because the metrics will need to be extracted from the code archive, it is important to know how the Software Configuration Management at MR is arranged. In this chapter the software configuration management system, the process model that is used to build the software and the way in which projects are organized in the software configuration system will be described.

2.2.1 IBM Rational® ClearCase®

ClearCase [IBM ClearCase] is the source control system that is used at MR. A slightly customized version of ClearCase is being used, which better suits the process model (Figure 2).

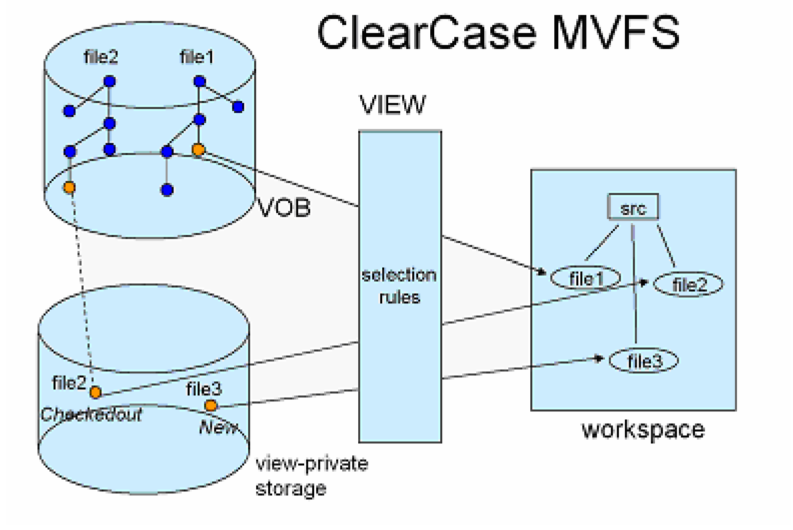


Figure 2: ClearCase Process model

The Versioned Object Base (VOB) is the main archive. When a developer starts developing, a view needs to be created. This view has certain selection rules, which determines the files the developer works with. As soon as the developer modifies a file on this view, a private storage is created in which the modified files are archived.

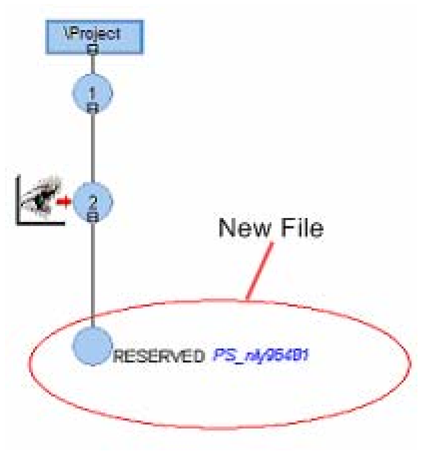


Figure 3: The version tree

In the VOB different files and different versions exist. This figure shows the version tree for a certain file. The file apparently has 2 versions, but the eye is on a version with the number 2. This means that the developer that has opened this version tree is working with that version of the file. There is also a newer file that is checked out by another developer and is not visible to the other developers until the file gets archived.

The files that are modified by the developer do not directly get the state archived, first they have to go through the MR process model.

2.2.2 MR process model

The process model to which the software configuration system has been customized is compliant with the daily build and smoke test [McConnell, 1996], which is common practice at Microsoft [McConnell, 1996]. When a small computer program is developed, it can easily be compiled and linked. When numerous of developers work on the same project, which is the case at MR, the process of building the program gets more complicated, because the program has to be build from thousands of files. This is why the daily build and smoke test is used: every file is compiled and linked every day and the resulting program goes through a smoke test. This smoke test is used to check if the basis functionality of the system is not broken.

How this process is implemented at MR is displayed in Figure 4. First, the developer checks out a file from the VOB, modifies it and then adds it to the post list that is given to the integrators. The integrators are the people responsible for the build process. Next, the file gets the state prepared. When the file is reviewed and everything is ok, the integrators take the file and the file gets the state taken. When the daily build and smoke test succeed, the file gets the status archived and from now on will be visible to other developers.

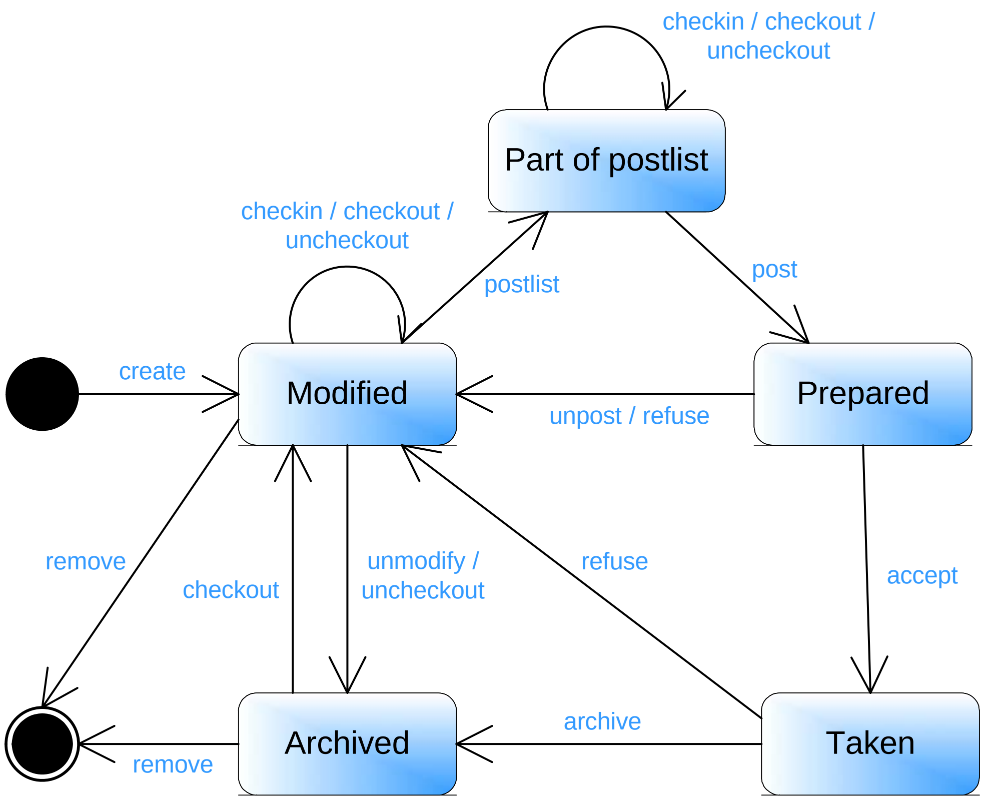
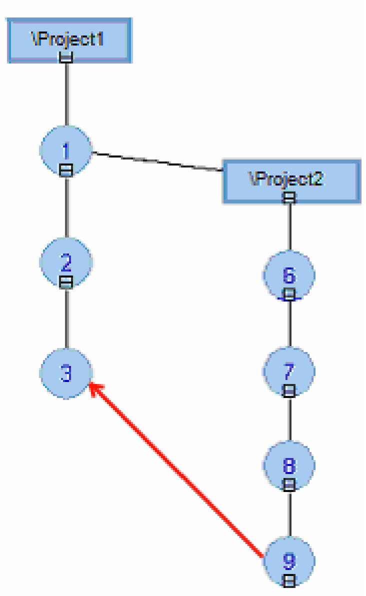


Figure 4: MR process model

The developers at MR do not work alone, they have to collaborate with other developers on the same source code. This parallel development can be managed in different ways.

2.2.3 Projects and development streams

When developing with several developers, much work has to be done in parallel. Multiple releases have to be managed, which require developers to work on the same source code. According to [Perry et all, 1998], parallel development is even inevitable in projects with more than one developer. How should parallel development effort best be conducted? Source control systems often make parallel development possible by providing the ability to create branches. Branches make it possible to create different versions of certain elements (files or folders).



A branch is a tree of versions, which makes it possible to develop different versions of the file simultaneously. When the development is finished on one branch, the files are brought back to the main stream, which is called a merge (indicated with the arrow).

Figure 5: ClearCase Branch

There are several ways in which branches can be used, as stated earlier, MR is not the only organization that has to deal with parallel development. Patterns have been developed, which describe the different ways in which these branches can be used, these patterns are called *streamed lines* [Appleton et all, 1998].

Different forms of branches can be created, like functional branches for developing certain functionality, organizational branches for team activities, etcetera. Streamed lines divide the branching patterns in the following categories: behavioral, creational, and structural.

The branching patterns used at MR will now be described as defined in [Appleton et all, 1998]. Two main development lines are used, the front-end mainline and the backend mainline. This approach is called *multiple mainlines*, these mainlines both represent certain different functionality and modifications should therefore not interfere with each other. Functionality is being developed by using *functional branches* of these mainlines. These *functional branches* allow the developers to work on their branch without interfering with others. The developers working on the *functional branches* merge according to the *merge your own code* policy, which means they are responsible for merging their own code to the main line. However a *code line dictatorship* policy is used, no check-ins, checkouts or branches can be created without the approval of the code line owner.

When the certain functionality is developed and a product needs to be released to customers, a release line is created. This is called *deferred branching* in combination with *overlapping releases*. *Deferred branching* is branching as soon as the work starts to conflict with other work, this is an alternative to branching at the beginning of a project. *Overlapping releases* is the creation of a separate branch before a product needs to be released, this branch can be used to stabilize the project and solve problem reports. New functionality will not be added to this branch, which prevents new problems or bugs to be inserted. On the release lines merges take place according to the *multi-merge early and often* policy, which means that the modifications are batched up into manageable deliveries that are regularly merged in. Code line ownership is used, a code line owner is assigned, which is responsible for the integrity and consistency of the code line.

3 Approach

The approach for the "Software Project Leader's Cockpit" is listed in the figure below:

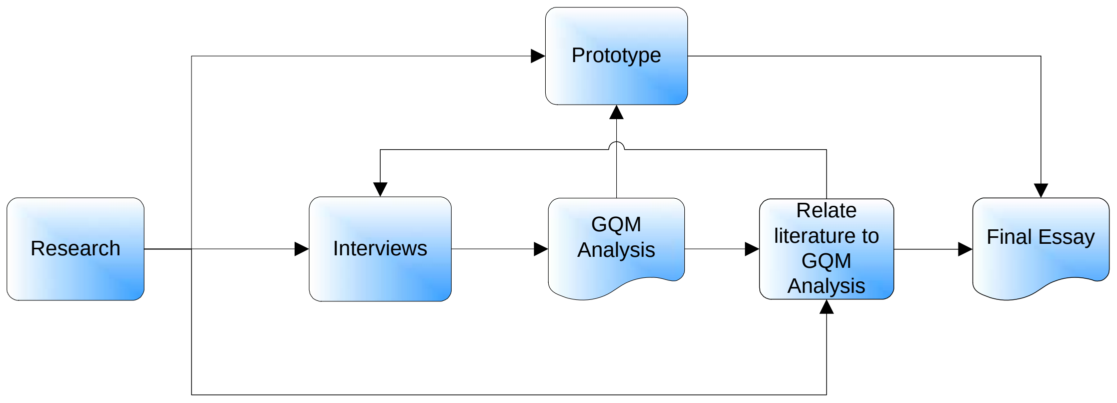


Figure 6: Master project process

First *research* has been done to find out what metrics were currently used elsewhere that could also be useful at MR, this *research was* mainly formed by a literature study. The literature study focused on: metrics that can be extracted from the code base that are useful for project management, branching patterns, Goal-Question-Metric analysis and experiences other organizations had in implementing similar metrics.

Interviews with the stakeholders were held in order to find out what requirements they had on the metrics. These stakeholders were project leaders, team leaders, architects and software configuration managers.

The results from these *interviews* were put in a *Goal-Question-Metric (GQM) analysis*, this analysis is used to gather the goals that drive the requirements. The *GQM analysis* was then validated with the stakeholders by the means of additional interviews and the final report was presented to these stakeholders.

After the GQM analysis had been validated, the results from the literature study were *related* to the GQM analysis to find out what measures resulting from the research could be used to answer the questions defined in the GQM analysis.

Also a *prototype* has been developed using input from both the research and the GQM analysis to study the feasibility of the different metrics and to generate input for the *final essay*. The *prototype* was developed in C#, this language is ideal as a prototype language because it lets the developer create functionality with minor effort. The code that makes up the prototype is not developed to support maintainability and should be thrown away when the actual tool is being developed.

The total results from the project were then described in a *final essay*, which is this document.

4 Research

The period the project took place was only four months, this made it important to make sure that knowledge was already gathered before the start of the project. It was clear that metrics needed to be collected during the project, so the question was which path to follow while elicitating the requirements on these metrics. The Goal-Question-Metric approach is often used to structure measurements, therefore this approach needed to be studied to find out if it was appropriate. Another objective was to find out what metrics are being used by other organizations and what their experiences were. This results in the following list of objectives for the research during this project:

- Find a method for structuring the measurements (Goal-Question-Metric approach)
- Find additional useful metrics for MR
- Find the experiences other companies have with similar metric programmes

4.1 Process for defining software measures

We could just define the metrics that we would want to use and start using them, but in order for the measures to be effective, they must be, among other things, focused on specific goals [Basili et all, 1994] . Therefore a framework is needed for defining the measures in terms of business goals and related statistical data.

4.1.1 GQM Analysis

Metrics are often collected because they seem useful, in the end these metrics are not used, because they don't give a concrete answer to a question or because they do not help in reaching a certain goal. In goal driven measurement the question is not "What metrics should I use?", but "What do I want to know or learn?"[Basili et all, 1994] . The GQM approach is a systematic way to translate the organization's objectives into measurement goals and refine these goals into questions and values. GQM helps setting focus on the collection of data and on how to interpret this data. GQ(I)M is an addition to the original GQM approach that adds an indicator step between the questions and the metrics (Figure 7), the indicators are graphs or other forms in which the data, needed to answer the questions, can be displayed. The indicators make it easier to define the metrics needed to answer the questions.

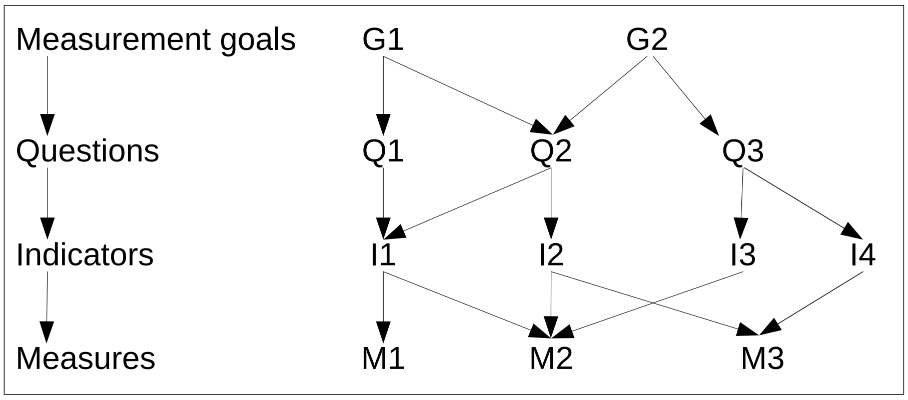


Figure 7: GQ(I)M approach

4.1.1.1 Why measure?

Measurement should take place for the following four reasons [Park et all, 1996]:

- **To characterize -** This helps us understand the element under measurement, which can be used during comparisons in the future.
- **To evaluate -** Evaluating measurements act as a sensor that can help us steer plans when they drift off course.
- **To predict -** Predicting helps us make plans. By looking back in the past we observe historical data that help us in making predictions.

- **To improve -** When improvements need to take place, we need to know how far we are in making those improvements. Improvement measurements help us in communicating those improvement goals and help us in eventually reaching those goals.

4.1.1.2 GQM process

The process definition for measurement using the GQ(I)M approach looks like the following [Park et all, 1996].

1) **Identify your business goals**
   First the business goals have to be identified that drive the organization's effort. This is needed to be able to translate the business goals into measurement goals. "Improve customer satisfaction" is an example of such a goal.
2) **Identify what you want to know or learn**
   What information is needed to improve the understanding of the activities and entities that influence these goals? The questions can be made up by relating a list of aspects like people, computers and subcontractors to the goals.
3) **Identify your sub goals**
   In this step the question list resulting from the previous step is analyzed and similar questions are grouped together into sub goals.
4) **Identify the entities and attributes**
   The sub goals with the questions they are grouped from are analyzed and the entities and attributes that play a part in the questions are listed. The attributes can be for example: size, effort, etc.
5) **Formalize your measurement goals**
   In the previous steps all the information is gathered that is needed to define the measurement goals. These measurement goals explicitly state the goal of the measurement itself. The following issues need to be described: the object of interest, the purpose of the measurement, the viewpoint from which the measurement takes place and the context in which it takes place (the environment).
6) **Identify Quantifiable questions and indicators**
   The measurement goals need to be translated into quantifiable questions and indicators. The indicators are examples of how the measurement should be displayed to answer the formulated questions.
7) **Identify the data elements**
   What information is needed to reproduce the indicators defined in the previous step?
8) **Define your measures**
   The measures that are used to collect the identified data elements are defined. They need to be defined in such a way, that they can only be interpreted in one way. Therefore the definition must not leave any room for interpretation.
9) **Identify the actions needed to implement your measures**
   A plan has to be created, which states how the measures should be implemented. Eventual existing measures should be **analyzed** and the results of these measures should be **diagnosed** to be useful. Next **actions** have to be defined in how to use these existing measures.

4.1.1.3 GQM results

The following elements will result from the GQM analysis:

- **Goal**
  The goal describes the reason for the measurements. The goal is described according to five attributes: the object to be measured, the purpose of the measurement, the measured property of the object, the viewpoint for the measurement and the context of the measurement.
- **Questions**
  These questions refine the goal. There are two types of questions: questions related to the quality model, which give a more detailed definition of the goals quality focus and questions related to variation factors, which describe the attributes that will probably affect the quality focus.

- **Indicators**
  Indicators are examples of how the data must be displayed to answer the formulated questions, this helps in clarifying what exactly to measure. For example, an indicator could be a bar graph or a table with values.
- **Measures**
  The metrics are definitions of data to collect that answer the set of questions.

4.2 Useful metrics

Because the objective is not only to gather the requirements at MR, but also to find out what metrics are being used at other companies, other metrics where researched. The metrics that have been found will be described in this chapter.

4.2.1 Entropy

The entropy [Hassan and Holt, 2003] is a metric that applies techniques from information theory to the history log of a source control system, in particular a technique called the Shannon Entropy. The entropy is a synonym for the chaos or complexity in a data set. The definition according to [Brennen, 2004] is: "*A measure of the amount of randomness of disorder in a system, dataset, or in the data read from a data source or device*". When a device is outputting different symbols and the first four symbols are [A, B, C, D], we are uncertain about what the fifth symbol will be. When the collection is [A, A, A, B], we are less uncertain, because the probability that A will also be the fifth symbol is greater than any of the other symbols; the entropy measures the amount of uncertainty in a distribution.

The authors of the paper apply the entropy formula to the history log of a source control system, this way the chaos is represented by the degree in which the modifications are spread throughout the system. Figure 8 represents two systems, one with minimum entropy, while the other has maximum entropy. The system with the minimum entropy has modifications that have only taken place in one file, the system with the maximum entropy has modifications that all have taken place in different files. When the entropy of a system gets high, the modifications that take place in the system will be harder to oversee for the persons responsible. The system will also get harder to maintain, which influences the delivery of product releases in time and under budget [Hassan and Holt, 2003] .

Software systems need to evolve gracefully to fulfill customers' changing needs and requirements, otherwise they will fail. To ensure such a graceful evolution of software systems, developers need to reduce and control the complexity associated with software systems [Hassan and Holt, 2003] .

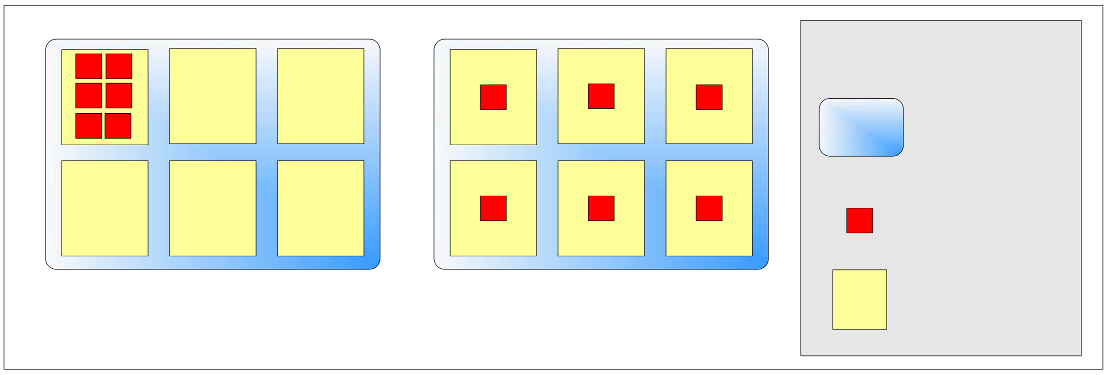


Figure 8: Entropy

According to Lehman's second law [Lehman et all, 1997], the complexity of software systems will increase when new features are added and therefore occasional maintenance is needed to reduce this complexity.

How does the entropy gets calculated? The authors analyze the number of modifications to a file over a certain period and divide this by the total number of modifications for all files. This is what they call the probability distribution, which is calculated for every file and used in the final entropy calculation (Figure 9)

$$H_n(P) = -\sum_{k=1}^{n}\Big(p_k * \log_2 p_k\Big),$$

$$\text{where } p_k \geq 0, \forall k \in \{1, 2, ..., n\} \text{ and } \sum_{k=1}^{n} p_k = 1.$$

Figure 9: Entropy formula

If, for a certain period, a lot of different files are modified, the entropy of the system will be high. This means the system will be hard to maintain, because the modifications are hard to track. There can be a sound explanation for this, the system could be under a large refactoring process. But it is a good strategy to monitor the total entropy of a system and look for large figures that need investigation (Figure 10). When peeks like the one in the figure are visible, they should be investigated.

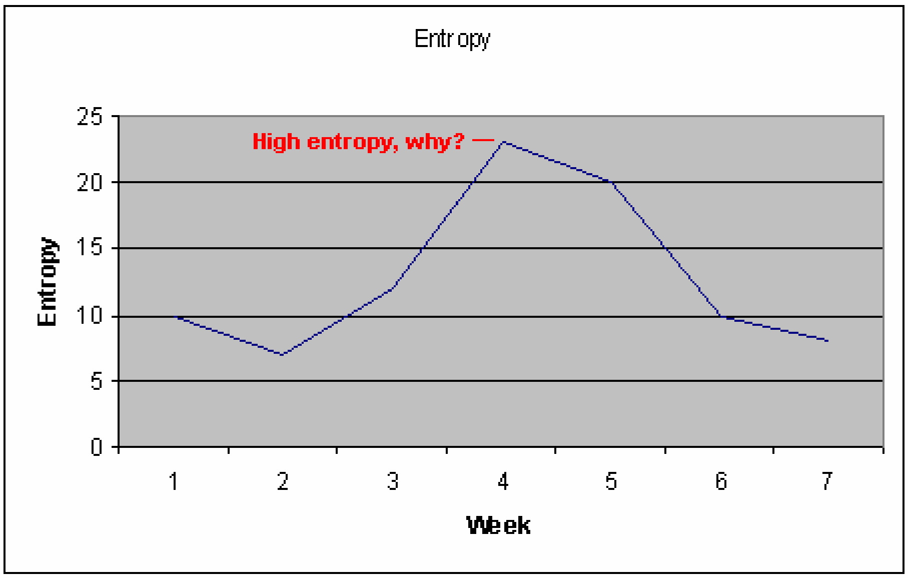


Figure 10: Entropy example

4.2.1.1 Evolution periods

The entropy explained earlier uses normal periods like months and weeks. There are also other period types that can be used to refine the formula when needed [Hassan and Holt, 2003] :

- **Modifications limit based periods**
  In stead of taking normal periods like months or weeks, the periods are determined by the number of modifications. The modification limit for example is set to 3000. This means that if the system has a total number of modifications of 12000, than these modifications are split up into 4 periods. The reason for doing this, is that there will now be four periods that have an equal number of modifications and can be compared better.

- **Moving Window Periods Sampling**
  For this technique, the normal periods that were explained earlier are used, but the window is not moved by the number of weeks or months in a period, but only by one of them. For example: If the normal period size three months, the first period is month 1 to month 3, the second period is month 2 to 4, the third period is month 3 to 5 and

so on. This technique increases the continuity of the graph and permits the data to be smoother.

To make the entropy numbers comparable with previous timescales and other systems, the context of the system needs also to be taken in account.

- **Standardized entropy with system size**
  During the evolution of a software system, files are added and removed. Because the size is changing, so will the number of modifications and the entropy will change. When the entropy is compared between a certain system some years ago and the same system at the current moment, it will make the comparison hard, because of this size. The entropy formula needs to be modified to take the size (number of files) of the system into account, because this influences the entropy value. The standardized entropy extends the original one to take the total number of files in the system into account. The entropy is now comparable between software projects, the only problem is the fact that systems sometimes have a lot of files that are hardly modified. These files will reduce the entropy, which off course is not the intention.

- **Standardized entropy with total number of modifications**
  As mentioned in the previous paragraph, the standardized entropy with system size has implications when a part of the system is hardly modified. This formula is based on the number of total modifications in stead of the total size. To improve the continuity of the graph, not only the total modifications of the current period are used, but the total number of modifications for the previous x periods including the current period. The form of entropy is comparable in time and between software projects.

4.2.2 Software Evolutions Observations Based on Product Release History

This metric [Gall et all, 1997] analyzes a software system based on product releases. The requirements on software systems keep changing over time, after many modifications have taken place, the complexity of the system will increase. This increment of complexity decreases the evolvability of the system. This is why software systems need periodic refactorings to maintain the evolvability of the system. The goal of the metric described in this paper is to identify modules that are potential candidates for refactoring.

The following three properties of the system are measured:

1. Size (number of modules)
2. Changing rate(number of changes to modules between releases)
3. Growing rate(number of added modules between releases)

The focus of the metric is on the macro-level of the system and only investigates structural information about the different releases. There is no analysis on source code-level. The system is structured in a hierarchical way, it is split up in subsystems, modules and programs.

On system level, the metric provides information of the way in which the system supports evolution, because the system should have a decrease of growing- and changing rate over time [Gall et all, 1997].

By analyzing the size, growing and changing rate at subsystem and module level, candidates for redesign can be identified. When certain subsystems or modules have a large growing and changing rate there is probably something wrong with their design and therefore should be redesigned in order to improve their support for evolution. A system itself could be very stable, but one of its subsystems very unstable, therefore this analysis at subsystem level should always take place and not only as a result of an unstable system. One of the main advantages of this measurement is the fact that the information is gathered relatively easily.

4.2.3 Size estimation

If we want to measure attributes of the code archive, the size is one of the first of these attributes one thinks off. Measuring the size is one of the criteria to reach certain CMM levels and the activity itself is known to be a good practice [Carleton et all, 1992]. According to [Putman, 1992], the size of a system correlates with the amount of effort, which makes it useful as a basis for effort estimations.

> Estimating size is the heart of the software project estimating process. – Lawrence H. Putman [Putman, 1992]

The size of the system can be measured in different ways, often used are Lines of code and Function points. Which one is the most appropriate?

4.2.3.1 Lines of code

Software size is traditionally measured in Lines of Code (LOC) [Boehm, 1981]. According to [Gollapudi, 2005], the lines of code correlate highly with the amount of effort needed to implement them. But there are a number of issues that need dealing with when the size is measured in LOC [Ross, 1999]:

- **The programming language**
  There are a lot of different programming languages and writing the code for a particular function could take 10 lines of code in one language and 30 lines of code in another.
- **Coding style**
  Developers can have different coding styles. One developer develops certain functionality using 10 lines and another develop could do this in 5 lines. This doesn't mean the former is less productive, he probably has another development style.
- **The way the LOC are counted**
  How are the lines of code measured? Are comments and blank lines included? If the LOC are compared with LOC of other systems, then it is important to know how the LOC were measured.

But it is possible to eliminate some of the disadvantages that come with LOC. Measuring the LOC leaves a lot of room for interpretation, which results in the problems mentioned earlier. This is why Park [Park, 1992] has proposed a framework consisting of a number of checklists to describe the definition of LOC that will be measured by the organization. These checklists describe all possible factors that can influence the interpretation, like comments, language, and etcetera.

4.2.3.2 Function point analysis

Because the LOC does not measure functionality [Vickers, 2003], A.Albrecht created the function point analysis. The function points are calculated by taking the factor of the type of functionality, the complexity (number of fields) and environmental factors (experience, staff skills), which results in a normal number, for example 353. The type of functionality, the complexity and the environmental factors are determined by rating them according to predefined tables [Vickers, 2003].

There is a problem with the function point analysis (FPA) method for measuring size, it is a form of rating and not measuring [Bailey, 2004]. Because of this, it is impossible to *measure productivity*, if it takes 10 weeks to produce 250 function points, it doesn't mean that it takes 1 week to produce 25 function points. FPA counts cannot be *compared numerically*, an application with 2000 function points is not twice as functional as an application with 1000 function points. This is because a lot of factors are measured that have no influence on the amount of functionality, like environment and complexity. FPA counts can also not be *compared between different organizations*, the subjectivity in rating the function points and the influences the organization, technology and domain have on the rating makes it meaningless to compare results between different organizations and possibly also in one organization.

FPA MKII [UKSMA, 1998] is a revision by Symons of the original FPA. Instead of counting the size of the components, it counts the number of logical transactions in the system, for example: *ordering a sales item* or *displaying an order list*. But it kept the disadvantages that the original FPA had: it has to be calculated by hand, which takes time and is very subjective. If two different persons calculate it, the results will definitely be different. One attempt in automatically measuring function points is a method called "Backfiring" [Dekkers and Gunter, 2000]. Backfiring counts the lines of code of a system and translates this to function points by calculating the product of the lines of code with a certain number. This number defers per programming language and public tables are available with these numbers [Jones, 1996]. However, backfiring should only by used to get a rough indication about what the function points for a certain system would be and should not be used as a basis for project decisions [Dekkers and Gunter, 2000].

4.2.3.3 Conclusion

Measuring the size of a system can help in making effort estimations and is known to be a good practice. There is no precise formal method to determine the size of a system that can be used to make an accurate estimation of the future effort. The function point analysis method is suited for measuring the business value of a system, but not for measuring the size when this size is to be used in conjunction with effort. The lines of code correlate with the amount of effort en measuring them does not include factors like environment and complexity, making it more suitable when it comes to measuring size. The lines of code are also easier to determine, because they can be extracted automatically. With the minor effort that measuring the size in lines of code takes and the benefits that it offers, every organization that takes effort estimation serious, should measure it and use it to support their project effort estimations.

4.2.4 Software program dependencies

Dependencies in software source code can provide useful information. They can be used to determine the impact of modifications, which can help in making estimations about what components will have to be modified and what components will need additional testing. This chapter will describe what dependencies are and how they can help in making estimations about components that will be modified and components that need testing.

4.2.4.1 What are dependencies

A dependency between components exists when component A requires component B to compile or function correctly [Jungmayr, 2002], in which a component can be a building block, a file, a model, class or other subsystem. There are two sorts of dependencies [Jungmayr, 2002]:

- A dependency between component A and component B is *physical* when component A cannot be compiled (and linked) without component B. This would be for example a direct method call from component A to component B.

- A dependency of component A on component B is *logical* if a change to component B would require a change to component A in order to preserve overall correctness.

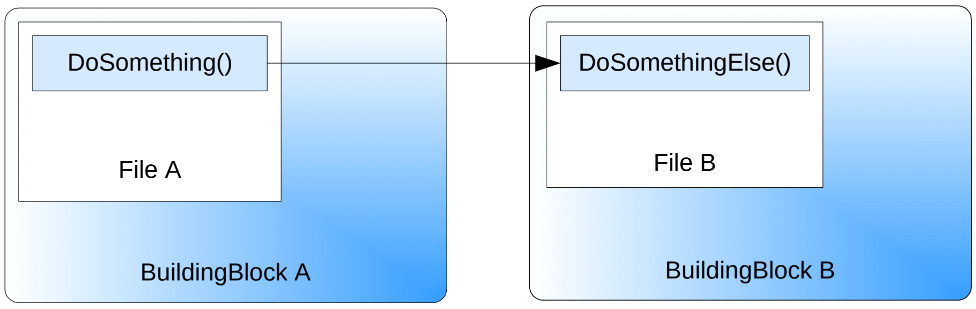


Figure 11: Dependencies between components

As stated earlier, a component can be a class, file or other subsystem. C code does not support the development of object-oriented code, thus eliminating class dependencies. As visible in Figure 11, when File A in Building Block A depends on File B in Building Block B, Building Block A will also depend on Building Block B. Dependencies can be analyzed on different levels throughout the system.

Dependencies between components arise because of the following phenomenons in the source code:

- **One component calls a method of the other component**
  As visible in Figure 11, File A has a physical dependency on File B.

- **They both share a shared/global variable**
  When the shared variable is located in one of the two components, the component using the other components variable has a physical dependency on that component. But when they both share another global variable and depend on the contents of this variable, the dependency is logical.

- **They share an external source**
  When both the components depend on the contents of this external source, than the relation is logical. The external source could be shared memory, a shared database, network, and etcetera. This type of dependency is very hard to detect.

As explained earlier there are physical and logical dependencies. The physical dependencies can be extracted rather easily from the code base by performing static analysis using regular expressions or by performing lexical analysis. Extracting accurate and complete logical dependencies is much harder, logical dependencies are often implicit and these are very hard to automatically extract from the code base. One approach that makes an attempt in doing this is, is described in [Gall et all, 1998]. When the dependencies are known, they can be used to answer certain questions.

4.2.4.2 How dependencies can be used

Dependencies could be, among other things, be used for these two reasons:

- Determine components that will also need to be modified when doing estimation about components that will be modified during the project.

- Determine components that will need to be tested when defining a test strategy at the beginning of the project. When the modifications have already taken place, the components that were actually modified can be used to find dependent components, which also need to be tested.

Dependencies should always be kept to a minimum in order to support testability and other quality attributes, but dependencies will always exist in source code. To use dependencies for the tasks listed above, these dependencies need to be known on component or subsystem level. Most of the dependencies are determined on method level, coupling can be used to relate these to higher levels (e.g. components).

4.2.4.2.1 Coupling

The dependencies itself are analyzed on method level. When making estimations about which components will also be modified or which components need testing, this probably isn't done at method level. This is because it is very hard to make such detailed estimations at the start of a project, therefore this is mostly done at file or building block level. This presents a problem, as visible in Figure 12, File A has a dependency on File B thru the method DoSometingElse(). This means that when DoSomethingElse() is modified, DoSomething() in File A should be retested, but when File B's method AnotherMethod() is modified, File A should not have to be retested. This information is not known because the dependencies are not analyzed on method level but on file level.

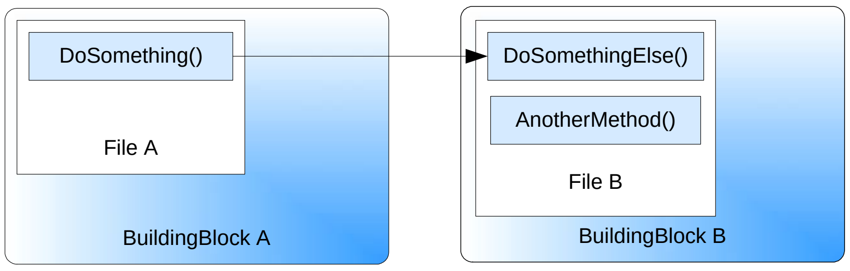


Figure 12: Dependency per method

To solve the above problem, coupling could be used to get a better handle on the probability that a certain component also needs testing. According to [Offut et all, 2000], Coupling between two components measures the dependency relations between the two components by reflecting the interconnections between methods. Therefore by analyzing the number of method dependencies between components, the coupling strength can be calculated. This coupling strength can help in making estimations about the probability that files need to be modified or about the files that need testing.

4.2.4.2.2 Analyzing components that might have to be modified

One might want to estimate the components that will need to be modified during a project in order to make a better impact analysis and in order to improve effort estimation. In practice, modifications always need to be made to other components as well, because of dependencies between these components. When these dependencies are physical the compiler will identify them during compile and link time. Therefore the physical dependencies can be used to make better estimations at the start of the project. However, not only components with physical dependencies will have to be modified, also the components with logical dependencies. How can these be determined? Well, as stated earlier, [Gall et all, 1998] makes an attempt in finding the logical coupling between component by analyzing the version log of the code base. The results of this logical coupling can be used to identify components that also need to be modified both when making estimations and when applying the modifications.

4.2.4.2.3 Analyzing components that might need testing

The components that need testing could be determined by analyzing the modifications that have taken place in the code base and relating these modifications to the dependencies on method level. This way the methods that also need testing can be identified. When the test strategy is defined in the beginning of the project, the modifications are not yet known. As stated earlier the estimation will probably also not take place on method level. Therefore the components need to be analyzed on file or building block level using coupling strength as explained earlier. According to [Offut et all, 2000], faults in one component may affect the coupled component. [Offut et all, 2000] also indicate that during integration testing faults are found exactly where couplings typically occur, therefore the connections (method calls) between program components should be covered during testing.

4.3 Metrics success factors

In order to succeed in the introduction of a metrics program, it is probably good practice to find out what the experiences of other companies were when they introduced similar metrics programs. First the reason for goal driven measurement will be described, followed by success factors concerning the organization and its culture, and finally recommendations concerning the definition of the metrics will be highlighted.

4.3.1 Goal driven measurement

[Dawson and Nolan, 2004][Daskalantonakis, 1992][Wiegers, 1997][Wiegers, 1999] agree that the metrics should be goal-driven. It is not stated in this exact way, but all the papers state that the reasons why the metrics are collected should be determined. One way to determine the metrics in a goal-driven way, is by using the Goals-Question-Metrics approach mentioned earlier. Another useful remark mentioned in [Daskalantonakis, 1992], is the fact that it is the actions taken as a result of analyzing the data that bring the results. So measurement itself must not be the goal of the organization, the goal should be improvement by the use of measurement, analysis and feedback.

4.3.2 Create a measurement culture

The employees that will be responsible for collecting these metrics should be made clear what the exact business value of the metric is and the collection of the metrics should be made part of the every day process [Dawson and Nolan, 2004]. This way the employees will collect complete and accurate data, by making the collection part of the daily process, the collection will also be done consistently.

Management should be committed to software measurement [Wiegers, 1999], establishing a metrics programme is an investment, management should be convinced of the benefits such a programme has to offer. Without management commitment the allocation of resources for the metrics programme will be very hard and employees will not effectively collect the metrics if management does not explicitly requires this of them.

The metrics data should also not be used to reward or punish individuals based on their performance [Dawson and Nolan, 2004]. When someone knows that the numbers they report might be held against them, they will probably stop reporting these numbers, or even temper with them. To succeed with a metrics program, management must make it clear to the personnel that the metrics will be used for project evaluation and not for personal assessment.

4.3.3 Metrics definition

Metrics should be presented in terms that directly answer the question that it was meant to answer [Dawson and Nolan, 2004], for example: a financial manager should view metrics that are presented in terms of money. The metrics should be described in a metrics plan with great accuracy, in order to make sure that these metrics can only be collected and interpreted in one way.

It is important to have a balanced set of metrics. It is wrong to start measuring too much, but it is also wrong to measure too little. Measuring should start with a small set of metrics that represent just the things that need to be known [Dawson and Nolan, 2004]. When these metrics become successful, they can be extended with other information that is desired. It is also important to start measuring on the right moment, the organization should explicitly support the measuring process and the gathering of metrics should happen very consistently. What metrics answer the questions that help you reach your goals? The Goal-Question-Metric approach helps in defining the metrics according the goals that need to be reached.

5 GQM Analysis

5.1 *Process*

Instead of using the GQM process model defined in chapter 4.1.1, a slightly modified version was used (Figure 7).

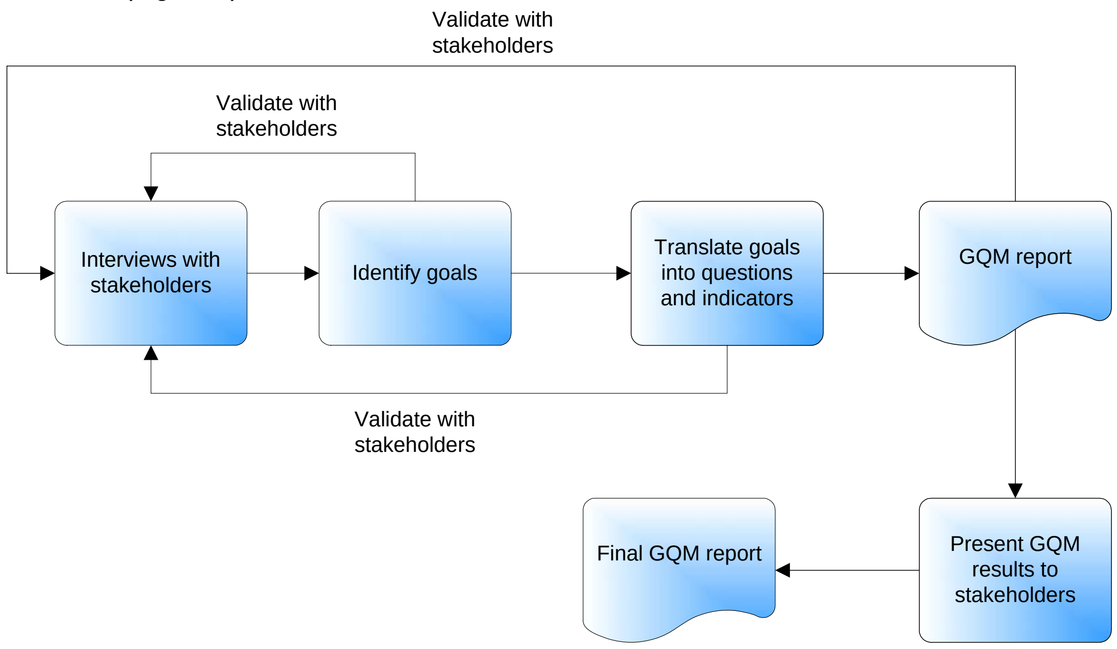

Validate with stakeholders
Validate with stakeholders
Interviews with stakeholders
Identify goals
Translate goals into questions and indicators
GQM report
Validate with stakeholders
Final GQM report
Present GQM results to stakeholders

Figure 13: GQM process model

First interviews with the stakeholders were held in order to find out what the requirements on the metrics programme were. The results from these interviews were then translated into higher business goals, these were validated and prioritized by the stakeholders. When everyone was satisfied with the business goals, the business goals were, by also using the previous results from the interviews, translated into questions and indicators.

During four more interviews, the questions and indicators were validated with the project leaders and were modified according to the feedback that was being received, this resulted in the GQM report. This GQM report was then presented in a presentation and the contents was discussed in a discussion at the end of the presentation. The feedback resulting from this discussion was then processed in a final GQM report.

5.2 *Interviews*

Because the outcome and results of the interviews were not clear and because it was undesired to miss other views or ideas, a semi-structure interview list was used. Semi-structured interviews have a predefined structure that can be used as a guideline. The questions are open and additional probing questions can be asked. Eight software project leaders, a team leader, two software architects, a configuration manager and a test coordinator were interviewed. Open questions about the current view the interviewee currently has on the code base and what information they would like to get from the code base were asked. These questions led to a number of interesting ideas, but the interviewees shared the most ideas.

5.3 *Identify goals*

The results from the interviews were then analyzed, the ideas were described and corresponding ideas were grouped. After this, according to the GQM method, the business goals had to be determined. Because the goals that were identified were not goals on the highest level, sub goals were used. The goals were described and send to the stakeholders for validation, they were also asked to give a priority to the different sub goals. The list with business goals and their description can be found in Appendix B. These are the goals with their priority resulting from the validation.

Priority	ID	Description
1	Goal 2.2	Improve estimations about the reliability of released products
2	Goal 1.3	Improve estimation accuracy for components that will be modified
3	Goal 2.1	Improve estimations of components that need testing
4	Goal 1.1	Improve merge efficiency
5	Goal 2.3	Improve the availability of test specifications
6	Goal 1.2	Improve accuracy of merge estimations
7	Goal 1.4	Improve accuracy of project effort estimation & project tracking

5.4 Translate goals into questions

After the goals had been validated, the goals were translated into quantifiable questions and indicators. This resulted in a first draft of the GQM report. Some initial suggestions for questions and indicators were done. Four of the project leaders gave their feedback on the GQM report during interviews. During these interviews, suggestions were also done for a number of new indicators and questions.

5.5 Present GQM report to the stakeholders

When there was satisfaction about the current version of the GQM report, the results were presented to the stakeholders. The goal of the presentation was to stimulate the software project leader to start a discussion about the importance and definition of the different measures. The global GQM report was presented and the measures were shown step by step. The discussion had some trouble starting up, but at the end there was an interesting discussion in which the stakeholders unanimous agreed on the following top three of goals:

Priority	ID	Description
1	Goal 2.2	Improve estimations about the reliability of released products
2	Goal 1.1	Improve merge efficiency
3	Goal 1.3 Goal 2.1	Improve estimation accuracy for components that will be modified Improve estimations of components that need testing

The goals 1.3 and 2.1 are merged because these goals can both be reached by analyzing the dependencies between components.

6 Relate GQM to literature

In this chapter the research that was conducted during the literature study and that was continued during the project, will be related to the literature in order to see how they might fit the requirements. The goals that are not covered by any of the found literature will also be described in the new areas chapter.

Goal 1.1 Improve merge efficiency
Goal 1.2 Improve accuracy of merge estimations
Goal 1.3 Improve estimation accuracy for components that will be modified
Goal 1.4 Improve accuracy of project effort estimation & project tracking
Goal 2.1 Improve estimations of components that need testing
Goal 2.2 Improve estimations about the reliability of released products
Goal 2.3 Improve the availability of test specifications

Goal	Entropy	Software Evolution Observations	Size estimations	Dependencies
Goal 1.1				
Goal 1.2				
Goal 1.3				X
Goal 1.4		X	X	
Goal 2.1				X
Goal 2.2	X	X		
Goal 2.3				

6.1 Entropy

- Goal 2.2 Improve estimations about the reliability of released products

The entropy tells us something about the chaos of the development process. If the modifications in the system are spread throughout the system, the situation of the system could be classified as “unstable”. This is because there is no overview of the modifications taking place in the code base. This is especially interesting during the alpha & beta test phase before a release. The system should stabilize during that project and an increase of chaos in the archive could lead to the decision to delay the release.

Whether or not the entropy metric is useful will be examined using a prototype.

6.2 Software Evolution observations

- Goal 2.2 Improve estimations about the reliability of released products
- Goal 1.4 Improve accuracy of project effort estimation & project tracking

This metric could also help in improving estimations about the reliability of released products and in improving the accuracy of project effort estimations. When a subsystem or module has a lot of changes and additions it will also be prone to errors more easily. Therefore the growing and changing rate not only identifies modules that need redesign but also modules that need testing before release. Because subsystems and modules that don't support evolvability in a satisfactory way are identified, the project leader is able to find out what modules probably take more effort during his effort estimation. The metric however is based on the evolution of the system and therefore more relevant for the software architects.

6.3 Size estimations

- Goal 1.4 Improve accuracy of project effort estimation & project tracking

Size estimation can be used to empirically support effort estimations. Currently size estimations are not used to support effort estimations. Measuring the lines of code of the software is relatively easily done and can be automatically extracted.

6.4 Dependencies

- Goal 1.3 Improve estimation accuracy for components that will be modified
- Goal 2.1 Improve estimations of components that need testing

As described in chapter 4.2.4, logical and physical dependencies can help in finding components that probably also need to be modified and to determine which components are dependent on the components that have been modified and should also be tested.

6.5 New areas

These are the goals for which no literature was found:

Goal 1.1	Improve merge efficiency
Goal 1.2	Improve accuracy of merge estimations
Goal 2.3	Improve the availability of test specifications

Goal 2.3 was not relevant, because during the discussion with the project leader the decision was made that this goal has no direct relation with the code base and falls out of the scope for this project.

Literature was found about how files can be merged in the most efficient way and about patterns that can be used to define merge policies, but this did not relate to the order in which streams should be merged. No literature was found about estimating merge effort. During the interviews measures were already defined, which should help in reaching these goals. Therefore, literature is not really necessary in order to reach these goals.

7 Prototype

Throughout the project, several ideas for measures came to mind. To see if these measures would be useful for the project leaders and to see if these measures are realistic concerning feasibility, they were validated and tested by prototyping them. In this chapter three measures will be described that were validated using this prototype. The global architecture for the prototype can be found in Appendix C. De prototypes will be described according to the concept that highlights the goal of the prototype, the approach that was used to carry out the experiment, the implementation that describes the results and finally the conclusion, which describes the feasibility and usability of the measurement.

7.1 Extracting data from ClearCase

When the version log from the source control system needs to be analyzed, this data needs to be extracted from this system. The source control system in question is called Rational ClearCase from IBM. This system does not support fine-grained analysis on the version log, the version log has to be extracted either thru the command line or thru the use of COM objects, but both take a lot of time to execute. Therefore the version needs to be stored in a local database in a normalized form that better supports analysis, see Appendix C.

First the modifications itself had to be extracted by parsing the version log. As described earlier, the user has to define a view to work with clearcase. A view to one of the projects was created to work with for the metrics programme. Next the command library of ClearCase was consulted to find the command necessary to get the version, this command appeared to be *lshistory*. The directories in the view were traversed and every folder called src was analyzed using the command. The results were parsed and stored into the database, making it easier to apply algorithms and visualizations to the data (Figure 14).

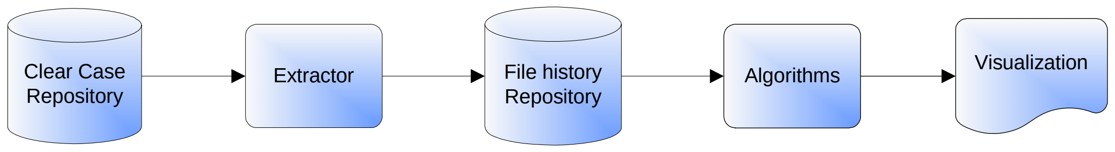


Figure 14: Extracting data from ClearCase

When analyzing the results the history for all the files was there for every stream. Only after a while it was realized that the metrics database did not contain the complete history. The history command was executed on the archer view and files that were added to other views were not there. This was a problem, however further research learned that this data could be obtained by extending the clearcase command.

7.2 Entropy

The Entropy metrics described earlier in the chapter 4.2.1, will be evaluated using the history log at MR. The different types of Entropy formula's will be applied to the historical data of software projects at MR to see if this Entropy parameter can provide the Project Leader with useful information. The Entropy results will be compared to certain events in the past to see if there is a useful relation.

7.2.1 Concept

The creators of the Entropy metric concluded from their research that the entropy metric is a good indicator for the amount of chaos or complexity in the development process. They also conclude that project managers should always try to keep this complexity as low as possible, because this complexity affects the releases of products both in time as under budget. The expectations for this attempt to apply the Entropy metric, is to see certain peeks in the graph that have sound explanations like a refactoring process. If this is the case, the entropy can be used to predict problems due to the complexity in the future, and help in reducing this complexity. At MR, the entropy parameter could be useful during the test phases of the projects. The modifications should decrease during these phases and the focus of t he modifications needs to lie on certain specific parts of the system. When the modifications are spread throughout the system, the project is not stabilizing and the project leader could decide to postpone the release.

7.2.2 Approach

The entropy will be applied to an representive project at MR in order to find peeks that could provide useful information.

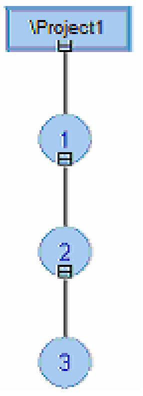


Figure 15: File versions

The entropy formula requires the modifications per file, this was extracted from clearcase by analyzing file versions. These file versions are created when a files get checked-out and is checked back in.

The experiments that are done by the authors of the paper, use the modifications from the version log, but exclude modifications done as bug fixes. They do not give an explanation why this is done, however bug fixes are not labeled in the version log at MR, which makes it impossible to exclude these. This is why the modifications used to calculate the entropy during these experiments include bug fixes.

7.2.3 Implementation

7.2.3.1 Traditional Entropy

First the entropy was analyzed for a small part of the system, but this was too little data for making proper analysis. Project X, which is a closed project at MR, was pointed out by one of the project leaders, because it should have some interesting data. Different periods can be chosen for calculating the entropy, but after experimenting with years, quarters, months, weeks and days, it became apparent that the entropy in periods of months and weeks is the most accurate and provides the most information (See Appendix A, Figures Figure 27 and Figure 28).

As visible in the graphs, the entropy shows different peeks. In order to find out if they would correlate with the releases, the release dates were plotted into the graph (Figure 16).

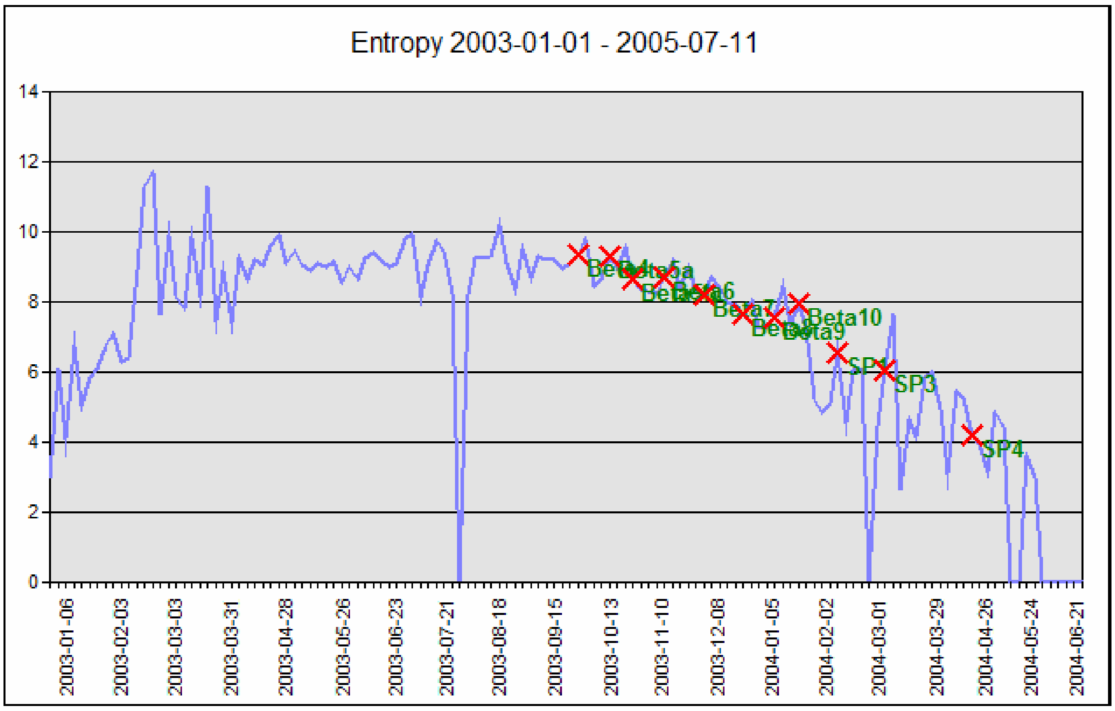


Figure 16: Entropy project X with release dates

As the graph shows, most of the release dates are at a peek. This means that the week before a release the chaos in the archive increases and after the release it settles down. The project leader responsible for project X confirmed this. In the weeks before a release the programmers tend to do as much bug fixing as possible, which causes the rise of entropy in the archive.

Next to this, the only other interesting information resulting from the previous graphs was the fact that the drops to zero represented Christmas holidays.

The normal entropy also seems to correlate with the number of modifications. The projects at MR have a number of phases and in the last two phases the project is stabilized and bugs are fixed. During the bug fixing, the number of modifications will probably be less high, but the entropy should be higher because this takes place throughout the whole project. As visible in the previous entropy charts, the entropy drops, as does the number of modifications.

7.2.3.2 Entropy with moving window period sampling

To see if the entropy in another form could provide interesting information, other entropy formula's, that were mentioned in the paper, were evaluated, to start with the moving window period sampling. According to the paper, this technique is expected to smoothen the graph and perhaps even identify other peeks. To calculate the entropy, periods of three months were used and the moving window was one month (See Appendix A, Figure 29). The graph is indeed smoother than the previous graphs, but does not show any new information. The periods that have high entropy in the previous graphs have now become one smooth peek.

7.2.3.3 Standardized entropy with total number of modifications

The standardized entropy takes the total number of modifications in account when calculating the entropy. This means that the entropy will stop correlating with the number of modifications and the entropy will also be comparable between different software projects and systems. The graph with standardized entropy looks different, the average entropy is the same throughout the project and the peeks in the graph are clearer. (See Appendix A, Figure 30). It also does not correlate with the number of modifications any more, because at the end of the project, the number of modifications is low, but the entropy is not getting lower. The peeks in the graph appear to be merges, the dates the peeks took place correspond with the dates the merges took place.

The standardize entropy shows a graph that better corresponds with the literature. However, for the different projects that have been analyzed, the average entropy does not increase at all over time, which contradicts Lehman's law [Lehman et all, 1997]. The entropy for the total system (the modifications for all the projects) was also analyzed, but showed the same results, the average entropy was not increasing. The graphs did show clear peeks that appeared to correspond to merges.

7.2.4 Conclusion

The normal entropy seemed to correlate with the number of modifications, therefore it was hard to determine if the values represented chaos in the development process or were just periods with many modifications. The weeks before releases the entropy increased due to last minute bug fixing. The project leaders already knew this, because this information can easily be found in the problem report database. In the test phases at the end of the project, the entropy was expected to be high because bug fixes are spread throughout the system, which should lead to a high entropy value. This was not the case, probably because the normal entropy appears to correlate with the number of modifications.

The moving window period smoothened the graph, but did not provide new information. The standardized entropy however changed the graph and made it comparable with other projects. The standardized entropy showed clear peeks which appeared to be merges. The peeks in entropy in the graphs are very short, less than a week. Therefore these peeks cannot be used for prediction with eye on project management. The overall entropy did not increase, not for the different projects, nor for the total system, therefore it does not correspond to the Lehman theory mentioned in the paper.

The different entropy graphs did show some interesting information, like when merges took place and that the entropy increases before release dates, but the software project leaders already gather this information from other sources. Therefore the graphs that were generated during this project, cannot be used to predict problems with projects or be used for managing projects at MR.

7.2.5 Further research

The paper that describes the entropy parameter is the first paper to apply the entropy formula to a version log, therefore the application of the parameter still is an experiment. Nevertheless, the concept has potential. Experiments at MR could be done to adjust the parameter to make it useful. At the end of a project, the modifications need to be stabilized and should be concentrated on the functionality that is developed during the project. The goal of the entropy parameter is to measure how the modifications are spread throughout the system, but it does not take into account if certain modifications take place in one building block (Building blocks are the subsystems, which make up the architecture at MR) or throughout several building blocks. If the modifications are spread in one building or throughout several building blocks is more interesting for MR. Therefore, experiments with the entropy parameter should be done to include the building blocks, as visualized in Figure 17.

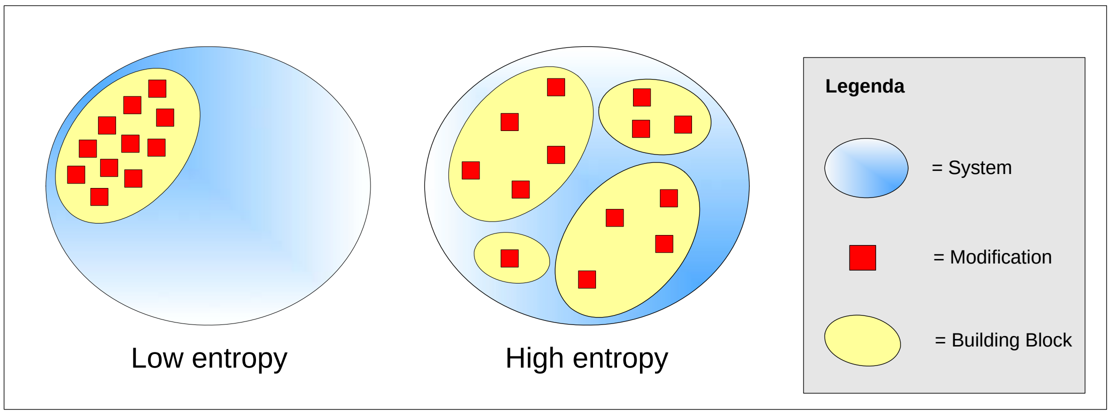


Figure 17: Entropy by building blocks

7.3 Merge estimation

At MR much different functionality is developed into the same software project. When certain functionality is finished and the functionality is released, the software needs to be stable. Therefore, other functionality that is still being developed and is not yet stable, should not be included with that release. This is the reason why there is being developed with different branches or streams. For all the functionality that is being developed, a separate branch is created on which that functionality is being developed without interfering with other projects, according to the *overlapping releases* branching pattern described in chapter 2.2.3. When the functionality is finished and has been tested, the files that have been modified are merged back into the main stream. An example of how this looks is visible in Figure 18. Project A and project B are both branches of the main baseline, when certain functionality is finished, this functionality is merged back to the main project, which is indicated by the blue arrow. A branch itself can also have sub-branches, this could be for several sub projects, these also need to be merged back.

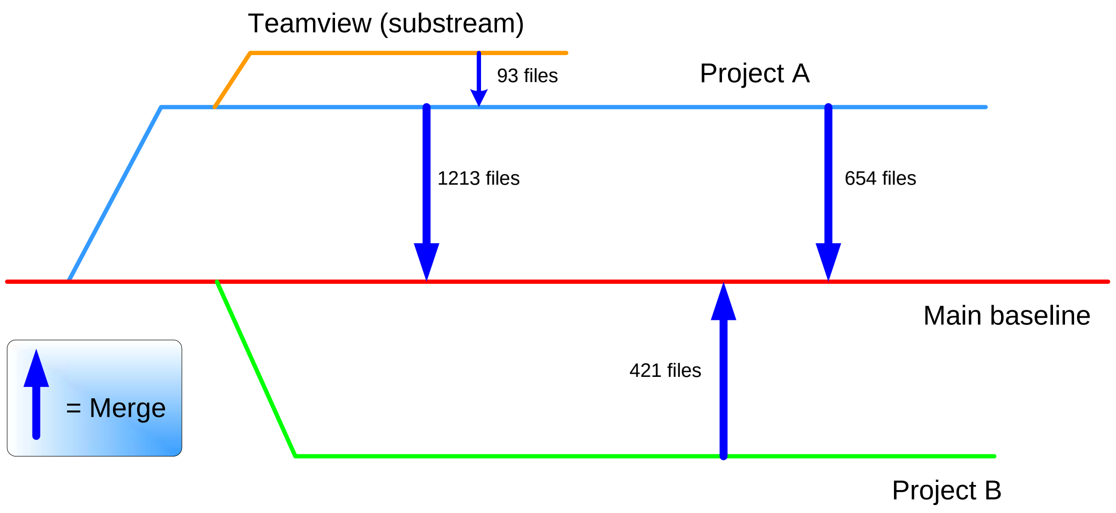


Figure 18: Development streams

When files are modified on only one branch (project A) and not on the other branch (main baseline), the files only can automatically be copied to the main branch. However, usually files get modified on both branches, when the same files are modified on both branches, the developer has to merge the functionality manually, which requires additional programming and therefore additional effort.

At the start of the project when the software project leader makes his estimations for the project, the merges are also scheduled. How much effort a merge will be, is unknown to the software project leader, because he does not know which files will be modified on the different streams. Therefore it is hard to make effort estimations and the effort a merge takes could negatively influence the projects lead-time. Also a project at MR usual contains several sub branches, which also need to be merged back during the project. These branches also have overlapping file and the project leader needs to determine in which order these branches should be merged back to the main branch. Determining the most efficient order is very hard and requires additional knowledge of each of the sub branches.

7.3.1 Concept

The problem with the merging is the fact that when the same files are modified for two projects, those files have to be merged by hand. On the main stream there can be numerous of other projects working parallel and this is where the problem lies. The project leaders don't have a good view on which streams and branches there are on certain moments and about what files are overlapping. Because of this the project leaders aren't capable of making accurate estimations about the amount of work the different merges will be, which influences the planning and the possibility of finishing projects before deadlines.

The number of files that have been modified on different streams that are the same should be visible on a daily basis. A view like the one in Figure 18, with the number of overlapping files between streams, could provide the project leader with more insight on the size of the merges, because currently the number of overlapping files is only known when the merge is about to be done. This information could be used by the project leader to adjust his planning and take corrective action in order to still finish his project in time. Also the number of overlapping files between all the sub branches on the main branch should be visible, this could help the project leader a lot in determining the most efficient merge order of these sub branches.

When a merge is done, the number of overlapping files is listed, there are two types of overlapping files: trivial and non-trivial merges. Trivial merges can be automatically merged by the source control system. Non-trivial merges have to merged by hand, which costs additional development effort. However, files that get marked non-trivial do not always have to be merged by hand. This is because the source control system does not analyze the contents of the files when determining the files that need to be merged. So the project leader needs to know the *actual* non-trivial merges between streams on a daily basis. These non-trivial merges could also be analyzed for the delta in lines of code and the complexity, because these factors probably also influence the effort the merge will take.

7.3.2 Approach

The goals of this prototype are to see if the information needed for the measurement can be extracted from the code base and to see if this information is useful. The first step will be checking the availability of the information, these are the steps for doing this:

- Extract and visualize the development streams (branches) from the code base
- Extract the modified files for all the streams
- Compare the modified files between streams in order to find overlapping files

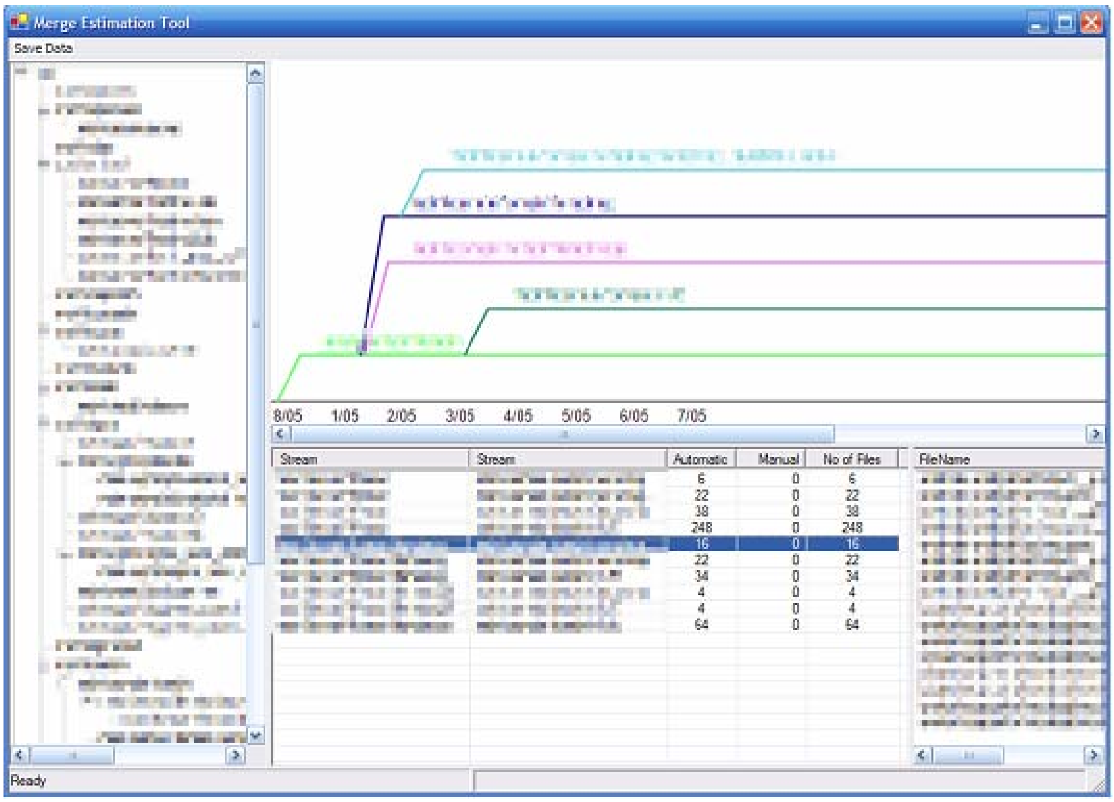
When the information appears to be available, experiments will be done to see if the number of files that have to be merged, correlate with the amount of effort. If there is enough time, the following attributes will also be evaluated to see if they also influence the amount of effort a merge takes:

- The delta in lines of code between overlapping files
- The complexity of the overlapping files according to McCabe's complexity measurement

7.3.3 Implementation

Because the data was already available in the metrics database described earlier, the first idea was trying to visualize the development streams automatically. This was done quite easily and the result was similar to the view displayed in Figure 18. The next step was to identify the files that need to be merged between streams. ClearCase provides a tool to do this, which is called "findmerge". After experimenting with this tool, the conclusion was that the performance was slow and unusable when the overlapping files for all streams need to be determined on a daily basis. Also, because additional analysis had to be done on the overlapping files (complexity, delta in loc), experimenting with manual comparison was the next step.

If the overlapping files for all the streams need to be known, all the modifications for all the streams need to be known. These modifications were already available in the metrics database. An algorithm was used to analyze the modifications for the individual streams and compare them with each other, this way locating the overlapping files between all the streams. This information was saved into a separate table, which supports additional analysis on the contents of the files that need to be merged. The files were visualized in a tool, which makes it possible to select different streams on the left and shows this stream in a different way on the right. It also shows all the combination of overlapping files for all the sub streams on the selected stream (Figure 19).



Stream	Stream	Automatic	Manual	No of Files
[illegible]	[illegible]	6	0	6
[illegible]	[illegible]	22	0	22
[illegible]	[illegible]	38	0	38
[illegible]	[illegible]	248	0	248
[illegible]	[illegible]	16	0	16
[illegible]	[illegible]	22	0	22
[illegible]	[illegible]	34	0	34
[illegible]	[illegible]	4	0	4
[illegible]	[illegible]	4	0	4
[illegible]	[illegible]	64	0	64

Figure 19: Prototype, files that need to be merged

The next step was validation of the correlation between the number of files that need to be merged and the amount of effort. The initial approach was to locate merges in the past with the number of files that were merged and interview the person responsible to ask how much effort they had spend. After a number of interviews it became clear that the effort spend in hours was being registered, but it was almost impossible to trace those hours back to certain merges. To solve this problem, short meetings were held with persons responsible for merges to ask how the number of files would relate to the amount of effort. The result was positive, the number of files has great influence on the amount of effort and they most definitely correlate with each other. However, this concerns the number of *actual* non-trivial files. What also resulted from the meetings was the fact that sometimes changes in design and the complexity also has great influence.

7.3.4 Conclusion

The prototype showed that it was possible and quite easy to extract the files that needed to be merged between the different streams. The prototype shows the number of overlapping files between the different sub streams on a certain stream, which should make it easier for the project leader to determine the most efficient merge order. Resulting from the short interviews that were held with the persons responsible for doing merges, it was confirmed that the number of non-trivial merges between streams correlates with the amount of effort. The number of files and the effort should be registered and used in conjunction with an indicator like the one in Figure 20. The resulting historical data makes it easier to make future estimations with the data that is shown by the prototype.

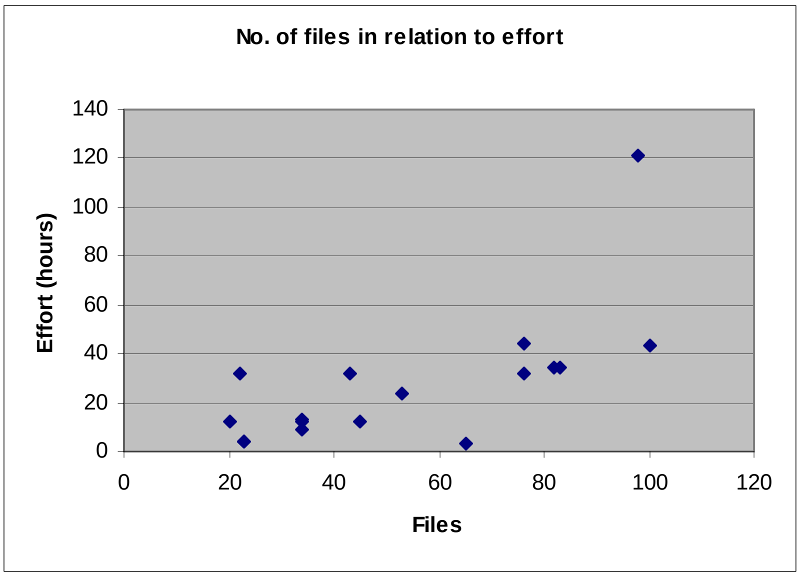


Figure 20: Indicator of historical merge data

7.3.5 Further research

When two files have to be merged manually, the effort it takes varies on different factors. In some cases only one line of code has to be merged, in other cases this can be ten lines or even more. Therefore the delta in lines of code between overlapping files on streams, can probably also provide information that helps in making estimations about the effort the merge will take. The complexity of the files that need to merged can also affect the effort, when this code is very complex, the developer spends more time in understanding this code before the merge can take place. A metric that is often used for measuring the complexity is McCabe's complexity [McCabe, 1976]. Further research should be done to see how these two factors influence the effort merges take.

7.4 Activity

Because the main reason for this project was the fact that the project leaders have no real grip on the work that is going on in the code base, visualizing the activity can be an import factor of success. The activity can help the project leader make decisions based on the stability of the code base and about which parts should need additional testing. The activity provides information about what is going on at a certain moment in de code base. This can help answering the following questions:

- **What parts of the system need more testing?**
  During the alpha test phase, the activity can be used to determine the components that have been modified to create a test plan and increase the test coverage.

- **Is the system stable enough to release it with a minimal number of faults?**
  At the end of the alpha/beta phases, the project leader has to make a decision about the reliability of the product. When there has been a lot of activity in the system that concern large changes, the project leader could make the decision to postpone the release and add another test phase.

7.4.1 Concept

These are the types of information that express the activity in a way that makes it interesting to the project leader, resulting from the interviews:

- **The number of files that have been modified over a certain period**
  These are the number of files that have been modified over a certain period. This means that if one file is modified twice in that period, it is only counted once. This information gives us information about the coverage of the modifications.

- **The number of checkins / checkouts of all the files over a certain period**
  When a developer makes a modification, the file is checked out and when he is finished, the file is checked back in. So when we count the number of check-ins, we count the number of modifications and the number of new files. This will result in a higher number than the number of files, because when a files is modified twice in a period, it is also counted twice.

- **The delta in Lines of Code between two versions after a checkin**
  This information indicates the size of the modifications. When there are only two lines of code added or remove during modifications, these are small modifications. But when this means 10 lines of code, the modifications become larger and the system becomes less stable.

The goal of this prototype is to see if the data is available and to validate the usefulness of this data. The activity measurement will be validated by identifying peeks in activity in certain parts of the system and relating this to problems during the projects in the past.

7.4.2 Implementation

Because all the modifications were already extract to the metrics database during the implementation of the entropy and the merge estimation, this information could easily be used, which resulted in Figure 21.

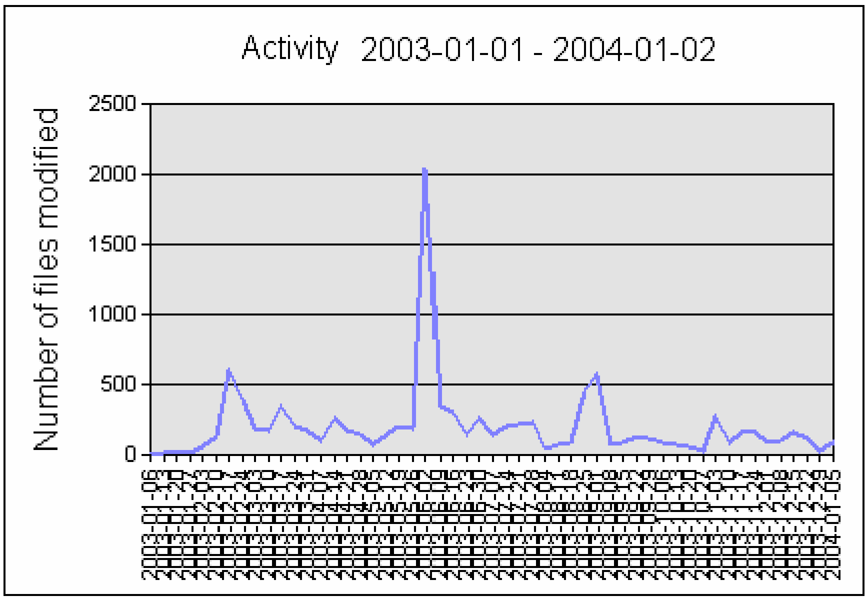


Figure 21: Number of files modified

The peek in the activity appeared to be a merge, the same as the entropy measurement. Next it was interesting to see the activity in individual building blocks, this could be an indicator for building blocks that need additional testing during the alpha test phase (Figure 22).

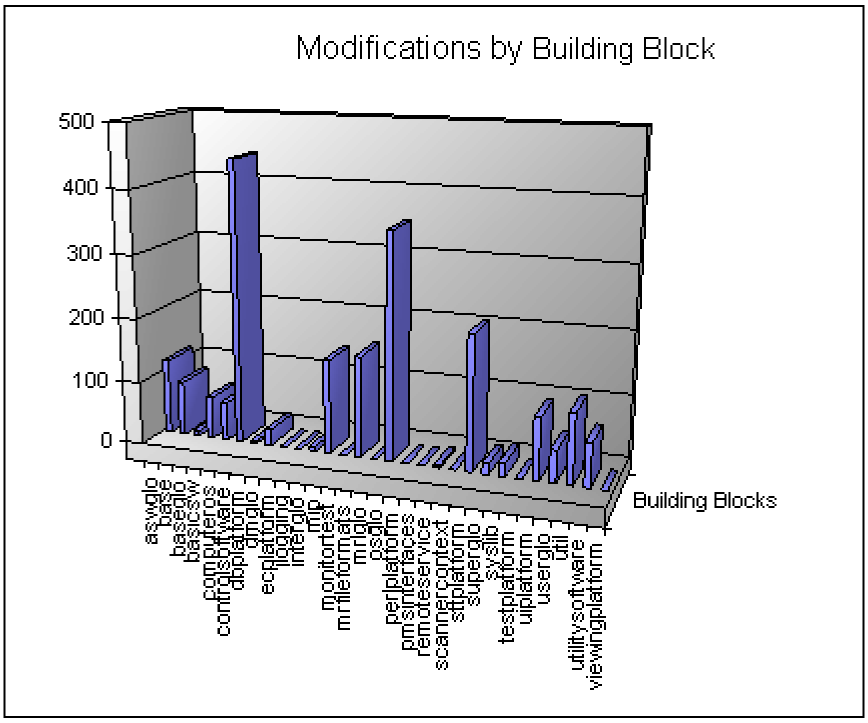


Figure 22: Modifications in building blocks

For the project leaders it was also interesting to see which developer has been making changes in certain building blocks. According to [Eick et all, 2000], the cityscape view diagram is an efficient way of visualizing this sort of information (Figure 23).

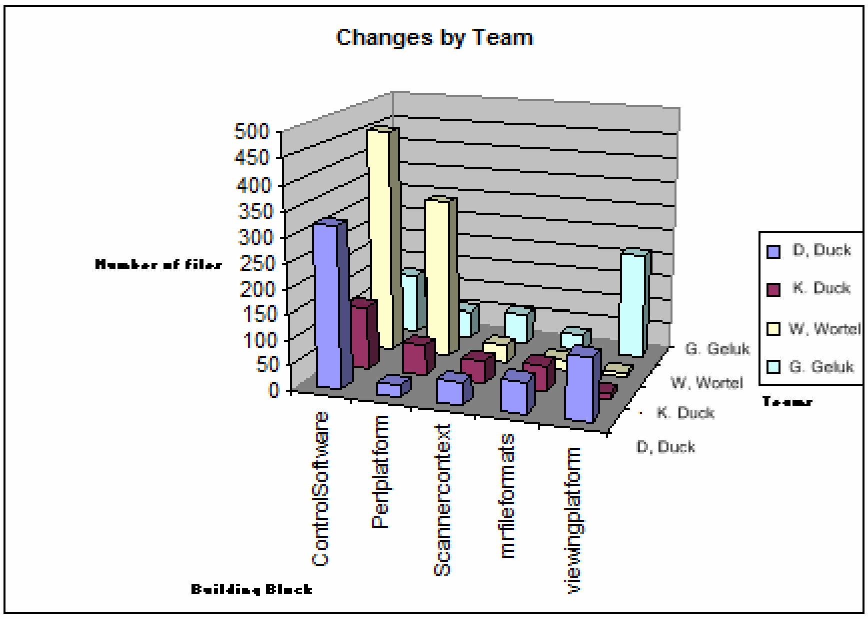


Figure 23: Cityscape view of activity

The activity measurement was validated by asking one of the project leaders to point out problems with certain projects at the past. The aim was to validate that high activity in building blocks could be an indicator for problems. The project leader appointed a certain project in the past together with three building blocks in which there was a lot of activity, almost unmanageable. After analyzing the activity in the project, it appeared that exactly these three building blocks had extremely high values of activity.

7.4.3 Conclusion

The activity can be retrieved from the source control system with minor effort, but it provides the project leader with valuable information. The prototype has shown that high activity in the past could be identified. A project leader provided information about a project in the past that had a lot movement, this project also introduced more problems than usual. The movement was confirmed by analyzing the activity. The prototype showed that the analysis of activity in the code base can really assist the software project leader in making decisions about the stability of the system and in determining what building blocks require additional testing.

7.4.4 Future research

During the prototype the number of modifications was analyzed, but the size of the modifications was not analyzed. During the stabilizing of the project, it is also important to know about the size of the modifications. If only three modifications have taken place, the project should be stable. If these three modifications concerned the addition of hundreds of lines of code, the probability of new problems being introduced is huge. Therefore experiments with the delta in lines of code between file versions should also be done in order to see if this is also useful to the software project leader.

8 Conclusions

The main research question was: "*What information can be extracted from the code archive that is useful to the project leader for managing his projects?".* The scope for this metrics programme included the code base and excluded everything else. Several interviews were held in order to find out what information from the code base could be useful to the software project leaders. The results were structured according to the Goal-Question-Metric approach, this resulted in seven goals. When the software project leaders gave priorities to these goals, the following goals formed the top three:

1. Improve estimations about the reliability of released products
2. Improve merge efficiency
3. Improve estimation accuracy for components that will be modified / Improve estimations of components that need testing

These goals should help the project leader in improving the control he has over his projects and to reduce faults that get discovered in the field. When the goals were defined, they were translated into questions and metrics that answer these questions. This leaded to the definition of eighteen questions and numerous of metrics.

During the project, research was done into other existing measures that could be useful at MR. Numerous metrics exist for project management, but these all focus on bug reports, time sheets, effort estimations, and etcetera. Therefore, only a small number of measures were found that could support the results from the Goal-Questions-Metric analysis that focus on the code. This was mainly due to the fact that most of the metrics that focus on the code-base concern software quality or software architecture. On of the most promising measurements found, was the entropy parameter.

Parts of the research and results from the GQM analysis were validated for usefulness and feasibility by the use of a prototype. First the entropy parameter was validated, which resulted in the conclusion that it is not useful to support decisions or to steer a project at MR. Next the merge estimation measurement was validated, the intention of this prototype was to see if the files that overlapped between streams/branches could be displayed and to see if this data was useful to the project leader. The data was easily extracted and after some short interviews it appeared that this data was useful. At last, the prototype was used to experiment with visualization for the activity in the code base. Carrying out a small experiment positively validated the activity.

To conclude, the result of this project is the definition of three goals that help the software project leader in controlling his projects. These goals were refined in questions, indicators and metrics that help in reaching these goals. Some of the questions that were still ideas and from which it was not certain if they would be useful or feasible, were validated by the use of a prototype.

Next, recommendations will be done about the implementation and continuation of the metrics progamme.

9 Recommendations

In this chapter recommendations will be done on the continuation of the metrics programme. First recommendations will be given on what measurements should be implemented. Next recommendations will be given concerning the success factors that apply on MR.

9.1 Measurements

At the end of the final presentation of the GQM results to the project leaders a discussion followed about what measurement goals were the most important. This resulted in the following list:

- **Improve estimations about the reliability of released products**
  This was chosen to be the most important goal, it is also probably the easiest goal to implement. A working prototype with this functionality has already been developed with minor effort.

- **Improve merge efficiency**
  The functionality needed to realize this goal can also easily be implemented. The functionality developed for the prototype can already provide the project leader with useful information. Next to this, experiments with the complexity and the delta in lines of code between overlapping files should be done in order to see if this correlates with the amount of effort.

- **Improve estimation accuracy for components that will be modified / Improve estimations of components that need testing**
  By analyzing the logical and physical dependencies between building blocks and files these goals can be reached. However determining these dependencies requires additional research and effort. Experiments should be done to validate the usefulness of these dependencies.

These goals should first be implemented, according to [Niessink and van Vliet, 2001], incremental implementation is one of the most important factors to success. Goal 2.2 and Goal 1.1 especially offer a lot of benefits to the project leaders and are relatively easy to implement.

9.2 Considerations

These are some considerations that result from the research done earlier during this project and that apply to MR.

Define a metrics process
For the measurement programme to succeed, effort has to be made in implementing it. Process descriptions have to be made on how the measurements will fit in to the organization. A process owner should be assigned, who will be responsible for the measurement process, this has to be a metrics champion. According to [Niessink and van Vliet, 2001], a measurement program depends on the value it generates for the organization. Therefore, the measurements described during this project should be integrated with the quality improvement process. The measurement data has to be used, but it also has to be seen used to make people aware and involved by the measurement programme.

Make the measurement data easily available
The measurement data also has to be available in an easy way, daily updates thru e-mail or using the intranet, makes it more attractive for people to use this data.

Define roadmap for continuous improvement
The goal of every organization should be continuous improvement of its organization and of what it delivers. Therefore, the metric programme also needs continuous improvement and a roadmap has to be defined that describes the steps concerning the improvement of the metrics programme itself. The contents of the metrics programme should be validated on a

periodical basis and updated as needed. The Deming-circle (Figure 24) is a useful framework for achieving this.

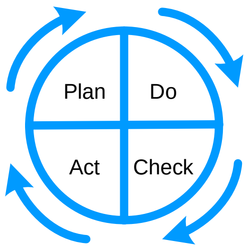


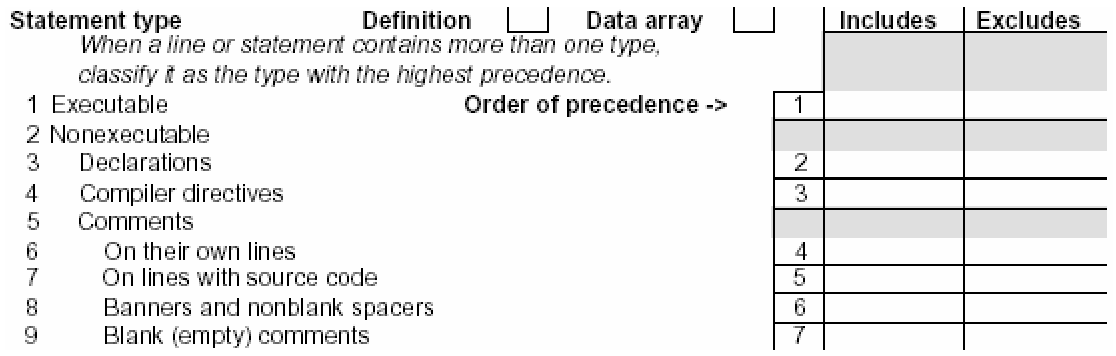
Figure 24: Circle of Deming

Relate problem reports to changes
If the organization wants the metrics programme to expand in the future, preparations should be made regarding data collection and data consistency. When modifications are made in the code archive and the files are checked back in, the reason of the modifications should be labeled to the files. The best way to do this is to create a standard format, with the type of modifications, project, and requirements. What also can provide interesting information is to add the problem report ID to the label. This way the modifications can be related to the problem reports, which can be used to collect additional metrics. XML is the worldwide standard when it comes to data definition and it also makes data extraction easier. The relation between source code and bug reports can be useful for the following reasons [Fischer et all, 2003]:

- Logical coupling between components can be determined by corresponding bug ID
- Error prone classes can be detected; classes that have relations with a log of bug ID's
- Estimations about the maturity of (parts of) the system can be done by measuring the number of bug ID's
- It could also be used to easily determine which bug fix has found place on which branch or stream

The relation between problem reports and changes is currently made because these are listed on the post lists, but this is not a one-on-one relation. When this information is stored in the software configuration system, not only will the data be easier accessible, it will also provide more detailed information, because it is stored on file-level.

Define a metrics plan
In order to make sure that the metrics are collected correctly and that they are interpreted in the same way, they need to be defined in a metrics plan. The specifications of what is being measured do not only need to be defined very accurately, but they should also be communicated to the users of the measurements. According to [Niessink and van Vliet, 2001], an important success factor for the measurement programme is the description on what actions should be taken for what values resulting from the measurements. All the possible results that could result from a measurement should be described with the possible actions that need to be taken when these values appear. [Park, 1992] is an example of a structured framework for defining a metrics plan for measuring the Lines of Code of a system. According to [Park, 1992] metrics should be defined in a similar way. [Park, 1992] also provides a lot of information about how an organization should go about defining similar definitions (See Figure 25). These metrics definitions define exactly was it measured and what is not measured, this way no misassumption can take place about what is being measured. Next to the definitions, also the process for collecting the metrics should be defined. Any change in the way the metrics are collected could affect the values of the metrics or the way they are interpreted.



Statement type **Definition** ☐ **Data array** ☐

When a line or statement contains more than one type, classify it as the type with the highest precedence.

	Statement type		Order	Includes	Excludes
1	Executable	**Order of precedence ->**	1		
2	Nonexecutable				
3	Declarations		2		
4	Compiler directives		3		
5	Comments				
6	On their own lines		4		
7	On lines with source code		5		
8	Banners and nonblank spacers		6		
9	Blank (empty) comments		7		

Figure 25: Example of a metrics definition [Park, 1992]

Create a metrics database

Source code version control systems contain large historical information that can be used for numerous of useful measurements. However, these systems do not provide sufficient support for fine-grained analysis. The information that is needed for the analysis could not only be coming from the version control system, but also from a bug report system, a planning tool and etcetera. In order for the measurement data to be in a form that is appropriate for detailed analysis a metrics database needs to be created, which contains the normalized data. This metrics database enables an organization to collect metrics with high performance and makes it possible to adjust the parameters of the measurement on the fly. A metrics database also simplifies the addition of other measurements in the future. The architecture of such a metrics database should look something like this:

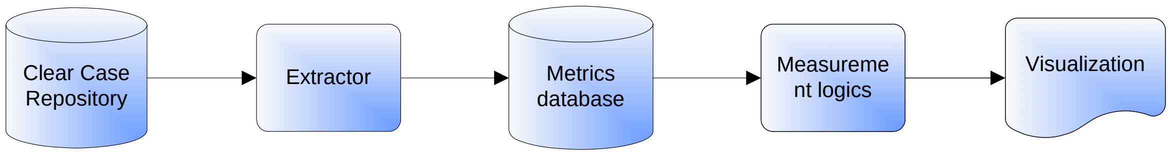


Figure 26: Architecture metrics database

According to [Fischer et all, 2003], [Zimmermann and Weisgerber, 2004], noise should be filtered out during extraction. Noise concerns large transactions, for example a change in coding standards, which requires a certain small modification to all the header files. Merge transactions are also a form of noise, because they contain unrelated changes. However, if merge estimations are noise or not also depends on the measurement performed, therefore the metrics database should allow some form of mechanism to exclude these or not. Currently there already is a system set up that can support this metrics database called SAMS.

In [Fischer et all, 2003] a description is given about how a metrics database should be set up and what information can be interesting and relevant. Also, a data model is given for an example metrics database.

9.3 Summary

This summarizes the list of recommendations:

- Start small, first implement the top three measurement goals
- Define a metrics process and a metrics champion
- Make the measurement data easily available
- Define a roadmap for continuous improvement
- Relate problem report to changes
- Define a metrics plan
- Create a metrics database

10 Evaluation

In this chapter the results of the project will be evaluated. First, the positive results of this project will be described, followed by the negative results. Next, the usability of the measurements will be stated and the approach followed during this project will be evaluated.

10.1 Positive results research

Not all the research that was done appeared to be useful, the research that is described in this document is only a small part of the information that was gathered during the project. Nevertheless, this information was still interesting and much knowledge was gained during this research. Also project management experience was gained, because of the analysis of the daily tasks of a software project leader that was done during the interviews and the research. Several presentations were given, which contribute to the communicative skills. The development of the prototype went well, because of the development experience gained earlier.

10.2 Negative results research

One of the important deliveries of this project was finding additional metrics used outside MR that could be useful to the software project leaders. During the research to these metrics, two metrics were found that had this potential. Other metrics were found that support the results from the Goal-Question-Metrics analysis, but this was it. This is mainly due to the fact that most metrics with regards to the code base focus on software quality and software architecture. These are also important areas, but do not directly help the software project leader in managing his projects. There also exist numerous of metrics that could help the software project leader, but these use sources like problem report databases, time sheets, effort estimations, and etcetera. However, the focus of this project was the code base, so these were out of scope. Also, the experiments that were done using the prototype require some further research, the need for this further research became apparent during the development of the prototype, but unfortunately there was not enough time to include it in this project.

10.3 Usability results

During the project it became clear that new techniques, methods or procedures are not easily carried out in the MR organization. However, the results from this project are very usable and can really support the software project leader in managing his projects. The measurements defined in the Goal-Question-Metrics approach were presented to the software project leaders, who were positive about the usefulness of these measures. The measurements are defined in a Goal-Question-Metrics report, which has been reviewed and approved by selected software project leaders. The measurement that applies to the main goal: *Improve estimations about the reliability of released products*, which concerns analysis of the activity, has been validated for usefulness and feasibility and the results were positive. A measurement for *improving effort estimations and efficiency for merges* has also been validated for usefulness and feasibility, which results were also positive. The time spent on validating the entropy measurement is not wasted, because now at least it is known that it is not applicable in the MR organization. Therefore the results of this project are definitely useful and a follow up assignment should be executed in order to implement the chosen measurements.

10.4 Reflection on research approach

The Goal-Question-Metric analysis itself went very well. Because semi-structured interviews were held, the chance existed that everyone that was interviewed would have different requirements. Luckily the requirements that resulted from the interviews were similar in most cases and the Goal-Question-Metric approach was very useful for structuring these requirements. Also, several validation cycles were held during the project. During these validation cycles the requirements were improved and the stakeholders got more involved in defining these requirements, the feedback that was given during validation was also positive.

11 Bibliography

In this chapter the references used in this thesis are stated. References that were used intensively are accompanied by a short summary. References that were only used for a simple quote or fact are not accompanied by a summary.

[Appleton et all, 1998] Appleton, B., Berczuk, S.P., Cabrera, R., and Orenstein, R., 1998, Streamed Lines: Branching Patterns for Parallel Software Development.

[Bailey, 2004] Bailey, R., 2004, Function Points – Numerology for Software Developers, http://www.hacknot.info/hacknot/action/showEntry?eid=59 (July 3, 2005).

[Basili et all, 1994] Basili, R., Caldiera, G., and Rombach, H.D., 1994, The Goal Question Metric Paradigm, *Encyclopedia of Software Engineering - 2 Volume* Set, John Wiley & Sons, p. 528-532.

The Goal Question Metric approach is an approach to define structured measurements, which need to be defined in terms of higher business goals in order to be successful.

[Boehm, 1981] Boehm, B.W., 1981, *Software Engineering Economics,* Englewood Cliffs, NJ : Prentice-Hall, 767 p.

[Brennen, 2004] Brennen, v.a., 2004, Cryptography Dictionary, http://www.cryptnet.net/fdp/crypto/crypto-dict.html (August 12, 2005).

[Carleton et all, 1992] Carleton, A.D., Park, R.E., Goethert, W.B., Florac, W.A., Bailey, E.K., and Pfleeger, S.L., 1992, Software Measurements for DoD Systems: Recommendations for Initial Code Measures, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

[Daskalantonakis, 1992] Daskalantonakis, M.K., 1992, A Practical View of Software Measurement and Implementation Experiences Within Motorola, IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992.

This paper describes the experiences of the author during the establishing of a company wide software metrics initiative at Motorola. A multi dimensional view of software measurement has formed the basis of this metrics programme. These views on the metrics range from categorizing the metrics to the describing the level in the organization at which the metrics are used. The paper also shows some example of metrics with their description.

[Dawson and Nolan, 2004] Dawson, R., and Nolan, A.J., 2004, Towards a Successful Metrics Programme, Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice, 2004.

The authors of this paper describe a number of ways that ensure a metrics programme is considered successful. They have come up with these ways during the 43 years of experience. The paper explains why a metrics programme should be established and what issues one should be aware of when establishing such a programme.

[Dekkers and Gunter, 2000] Dekkers, C., and Gunter, I., 2000, Using "Backfiring" to Accurately Size Software: More Wishful Thinking Than Science?, IT Metrics Strategies - November 2000 vol VI, no. 11.

[Eick et all, 2000] Eick, S.G., Schuster, P., Mockus, A., Graves, T.L., and Karr, A.F., 2000, Visualizing Software Changes, Technical Report Number 113, National Institute of Statistical Sciences.

[Fischer et all, 2003] Fischer, M., Pinzger, M., and Gall, H., 2003, Populating a Release History Database from Version Control and Bug Tracking Systems.

[Gall et all, 1997] Gall, H., Jazayeri, M., Klosch, R.R., and Trausmuth, G., 1997, Software Evolution Observations Based on Product Release History, Proceedings of the International Conference on Software Maintenance

The authors of this paper analyze the modification history of a system in order to uncover potential shortcomings in the design of the system. The system is analyzed on different levels: system level, subsystem level and module level. The following attributes are analyzed: The size in number of programs, the changing rate in percentage of changes between releases and the growing rate in percentage of added programs between releases. If for instance a certain module of the system has a great changing and growing rate, it gets interesting for redesign.

[Gall et all, 1998] Gall, H., Hajek, K., and Jazayeri, M., 1998, Detection of Logical Coupling Based on Product Release History, Proceedings of the International Conference on Software Maintenance.

The authors of this paper introduce a new method for detecting logical coupling between components. They analyze the release history of a software system to find logical relations and change patterns among modules. Components that are often modified together and have the same labels are grouped together. The technique can also be used to detected dependencies that do cannot be found in the source code and identifies modules that need redesign. When two components have more identical labels they will get a higher logical coupling rating. The technique is applied to a software project for telecommunication many discovered dependencies were actual dependencies and the results were promising.

[Gollapudi, 2005] Gollapudi, K.V.V.G.B., Function Points or Lines of Code? – An Insight.

[Hassan and Holt, 2003] Hassan, A.E., and Holt, R.C., 2003, The Chaos of Software Development, University of Waterloo.

In this paper the authors apply a technique from the information theory called entropy on the source log of a number of open source project to see if the entropy correlates with the chaos of the development process. The entropy formula measures the uncertainty of information in this case the uncertainty of what file will be modified. For software systems to evolve gracefully the complexity needs to be controlled and reduced. The authors conclude that the complexity normally has a good explanation it could be a refactoring for example.

[IBM ClearCase] IBM Rational ClearCase, http://www-306.ibm.com/software/awdtools/clearcase/index.html (July 13, 2005).

[Jones, 1996] Jones, C., 1996, Programming Languages Table, http://www.theadvisors.com/langcomparison.htm (July 3, 2005).

[Jungmayr, 2002] Jungmayr, S., 2002, Testability Measurement and Software Dependencies, FernUniversität Hagen, Germany.

Most papers have focused on testability metrics for system components, this paper however, describes the analysis of test critical dependencies to measure the testability of a software system. The dependency measures are also used to track down these test critical dependencies in order to redesign the parts where they occur. This way the number of test critical dependencies will decrease which will increase the overall testability of the system. Four case studies are explained, which indicate that a small number of dependencies have a large impact on testability.

[Lehman et all, 1997] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., and Turski, W.M., 1997, Metrics and Laws of Software Evolution - The Nineties View, Proceedings of the Software Metrics Symposium, 1997, Fourth International (1997), pp. 20-32.

[McCabe, 1976] McCabe, T.J., 1976, A Complexity Measure, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

[McConnell, 1996] McConnell, S., 1996, Daily Build and Smoke Test, IEEE Software: Best Practises, Vol.13, No.4, July 1996, http://www.stevemcconnell.com/bp04.htm (July 13, 2005).

[Niessink and van Vliet, 2001] Niessink, F., and van Vliet, H., 2001, Measurement Program Success Factors Revisited, Vrije Universiteit, Amsterdam, The Netherlands.

[Offut et all, 2000] Offut, A.J., Abdurazik, A., and Alexander, R.T., 2000, An Analysis Tool for Coupling-based Integration Testing, The Sixth IEEE International Conference on Engineering of Complex Computer Systems, p. 172-178, Tokyo Japan, September 2000.

This paper described a tool that can assist in integration testing. The tool takes a collection of test data and measures the amount of test coverage according to coupling criteria. It is based on coupling based testing, which concerns the testing of the path that the parameters, which are used in method calls, can possibly follow.

[Park, 1992] Park, R.E., 1992, Software Size Measurement: A Framework for Counting Source Statements, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

This paper provides a framework for defining counting and reporting two frequently used measures of software size: physical source lines and logical source statements. The paper provides templates that should decrease the possibility of misinterpretation of the measured size.

[Park et all, 1996] Park, R.E., Goethert, W.B., and Florac, W.A., 1996, Goal-Driven Software Measurement – A Guidebook, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

With the normal Goal Question Metric approach, the measurement questions have to be directly mapped to the business goals, which appears to be quite difficult. This paper proposes a new process including measurement goals, which is an extra step between the original business goals, and measurement goals. Another step is also included between the questions and the metrics, called indicators. The indicators are graphs and display the information in such a way that it easily answers the question, which makes it easier to determine the data that answers the question.

[Perry et all, 1998] Perry, D.E., Siy, H.P., and Votta, L.G., 1998, Parallel Changes in Large Scale Software Development: an Observational Case Study, in Proceedings of the 20th International Conference of Software Engineering (ICSE 20), ACM Press, Kyoto, Japan, April 1998.

[Putman, 1992] Putman, L.H., 1992, Measures for Excellence: Reliable Software On Time, Within Budget. Englewood Cliffs, NL: Youtdon Press, 400 p.

[Ross, 1999] Ross, M., 1999, Size Does Matter: Continuous Size Estimating and Tracking.

[UKSMA, 1998] UKSMA Metrics Practises Committee, 1998, MKII Function Point Analysis Counting Practises Manual.

[Vickers, 2003] Vickers, P., 2003, An Introduction to Function Point Analysis, Northumbria University.

[Wiegers, 1997] Wiegers, K.E., 1997, Software Metrics: Ten Traps to Avoid, http://www.processimpact.com/articles/mtraps.html (July 1, 2005).

This paper identifies ten traps that often sabotage metrics initiatives. When an organization initiates its own metrics programme it should be aware of these common risks. The traps range from lack of management commitment to measuring the wrong data.

[Wiegers, 1999] Wiegers, K.E., 1999, Software Metrics: A Software Metrics Primer http://www.processimpact.com/articles/metrics_primer.html (July 1, 2005).

This paper provided information about how an organization should start with a measurement programme. The author explains why an organization should start measuring what it should measure and provides the reader with tips.

[Zimmermann and Weisgerber, 2004] Zimmermann, T., and Weisgerber, P., 2004, Preprocessing CVS Data for Fine-grained Analysis.

Appendix A. Entropy Diagrams

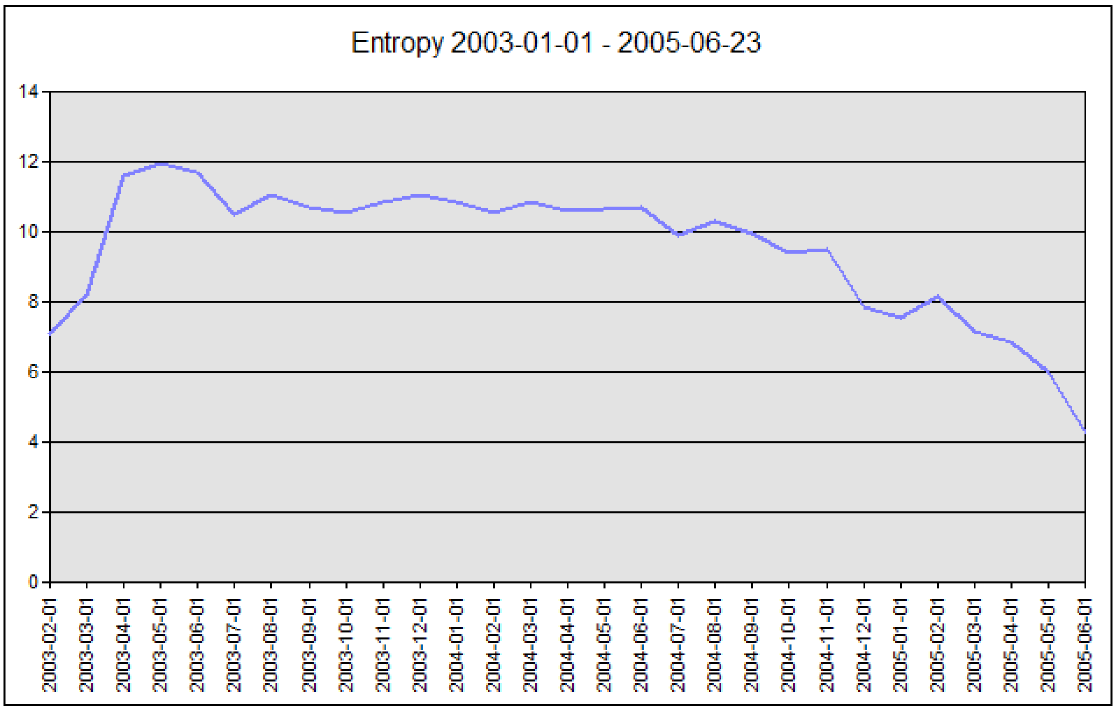


Figure 27: Entropy in months

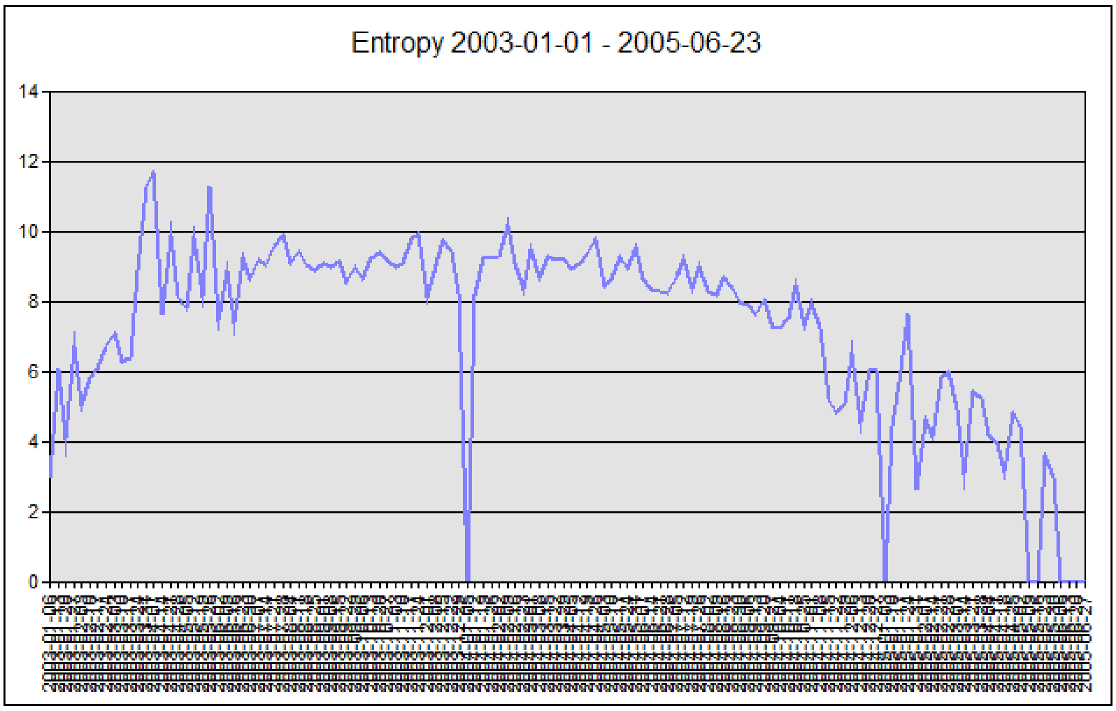


Figure 28: Entropy in weeks

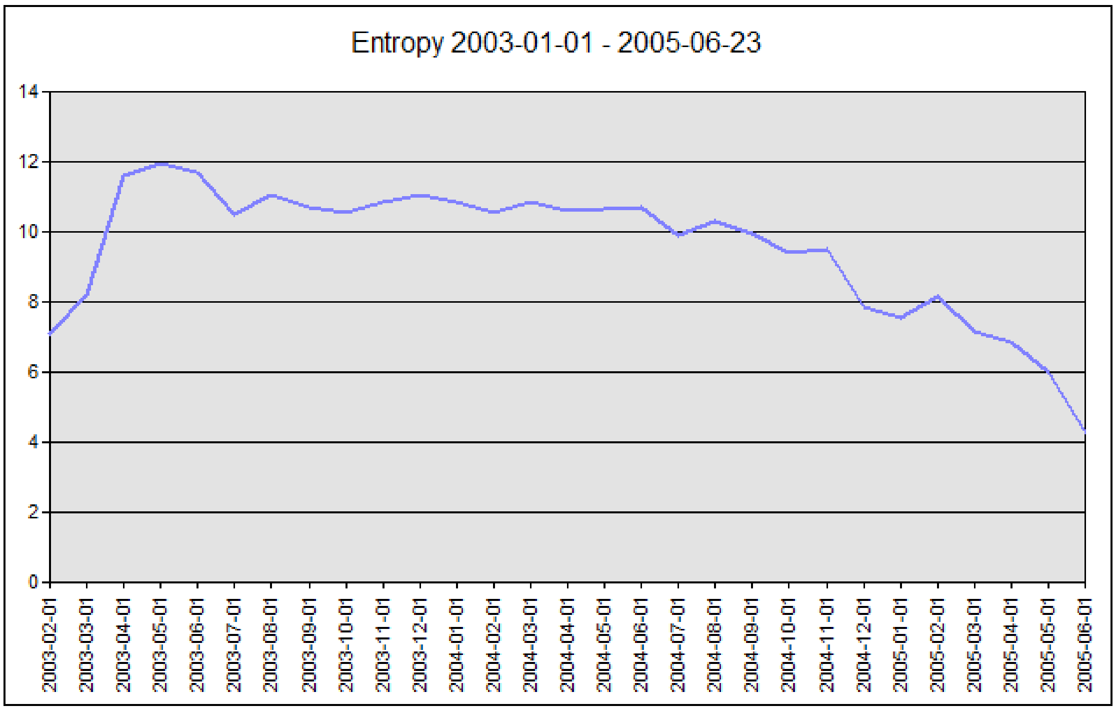


Figure 29: Entropy with moving window period sampling

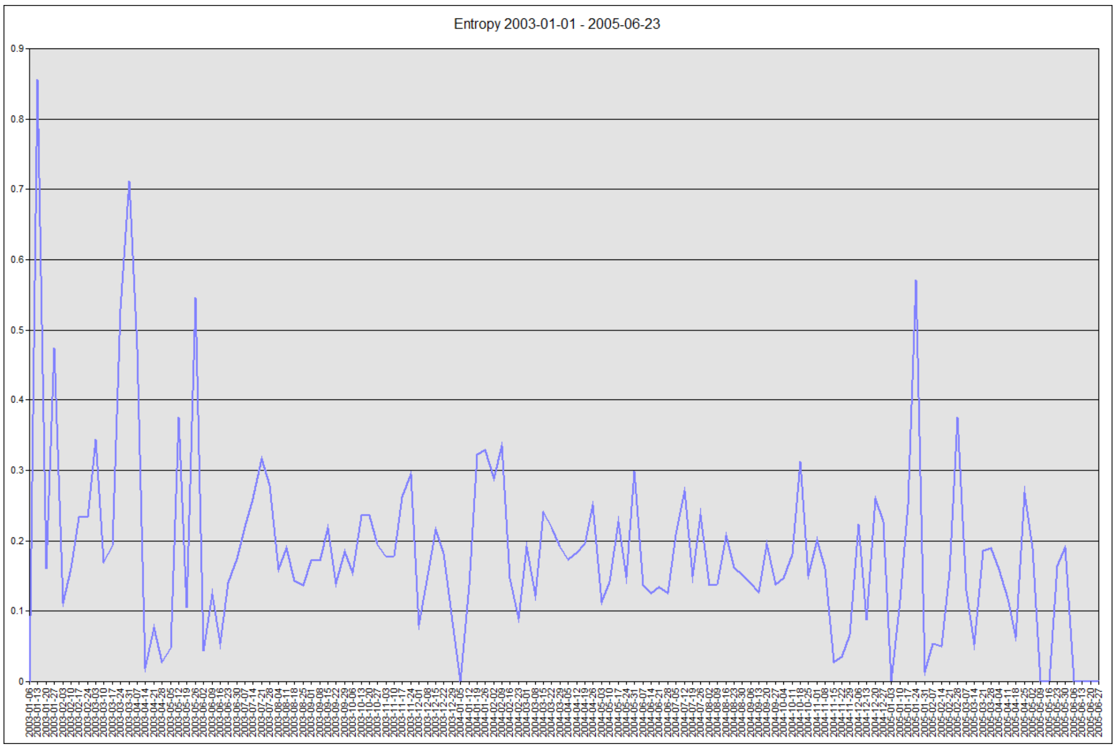


Figure 30: Standardized entropy in weeks

Appendix B. GQM Business goals

These are the goals that were identified during the first session of interviews:

- **Goal 1. Improve time to market**
  Projects need to finish before deadlines. To make sure this happens, realistic deadlines are needed, by making accurate estimations of time, costs and effort.

  - **Goal 1.1 Improve merge efficiency**
    For a project, several branches are created on which features are being developed. These branches occasionally have to be merged back to the project base line, which is called a delivery. The delivery of different branches influence each other, the order in which the branches are delivered correlate with the amount of work this will take. Therefore the branches need to be merged in the most efficient order.

  - **Goal 1.2 Improve accuracy of merge estimations**
    Different projects need to be merged with each other, when a lot of files between these projects overlap, the merge can take a considerable amount of work. If the project manager has no clue about the effort the merge will take, he cannot make an accurate planning and cannot ensure deadlines will be made. Therefore the accuracy of the estimations on merges effort need improvement.

  - **Goal 1.3 Improve estimation accuracy for components that will be modified**
    When a Work Break Down structure is made, estimations need to be done about the components (Files & Building Blocks) that will be modified. In reality more components are modified than the ones that were estimated, this is because there are numerous dependencies between the components, which cause the need for additional modifications. If the dependencies between the components are known, more accurate estimations can be made by calculating the components that also need modifications.

  - **Goal 1.4 Improve accuracy of project effort estimation & project tracking**
    Project effort estimation & project track should be improved by using methods and techniques that analyze the activity in the codebase.

- **Goal 2. Reduce the number of faults in the field**
  The number of faults that are discovered when a product is already in use by customers can be devastating to business, especially in medical systems. This is why these faults should be kept to a minimum.

  - **Goal 2.1 Improve estimations of components that need testing**
    Currently only code that is modified is tested. Often there are other files that depend on the functionality that was modified, which off course should also be tested. By analyzing dependencies and looking at the activity in the different parts of the code base, better estimations of components that need testing can be made, which will hopefully lead to a decrease in the number of faults.

  - **Goal 2.2 Improve estimations about the reliability of released products**
    When a project leaves the alpha test phase, all tests have been performed and the number of faults in the sourcecode should be minimal. But what about the modifications that were needed to fix the bugs in the alphatestphase, what size have they been? What part of the source code has been touched and how many times? When large modifications have taken

place right before the project left the alpha test phase, this could mean trouble. The project leader needs a better view on these modifications to make better decisions about the reliability of the product.

- **Goal 2.3 Improve the availability of test specifications**
  The test specifications that specify which part of the system needs to be tested and in what way are often missing or incomplete. Because of this, when the system needs to be tested, the people who are responsible for this, don't really know what to test and how to do this. This leads to uncertainties about the test coverage. Indicators about the completion of test specifications could help in improving the availability.

Appendix C. Prototype figures

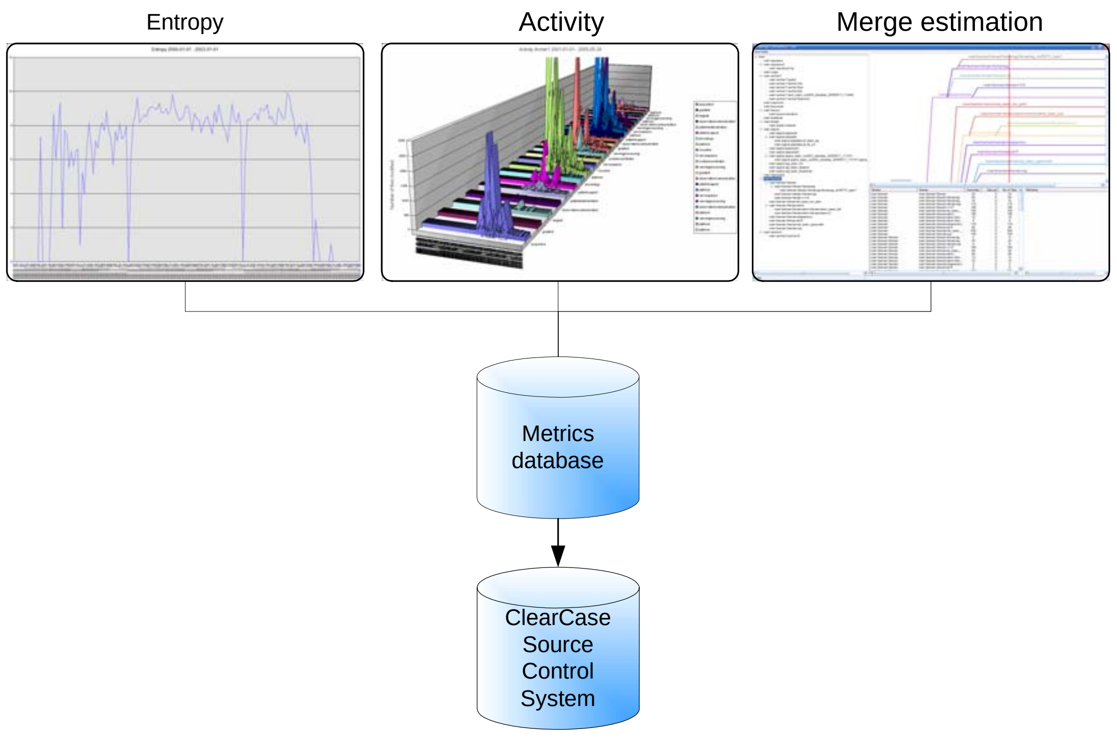


Figure 31: Prototype architecture

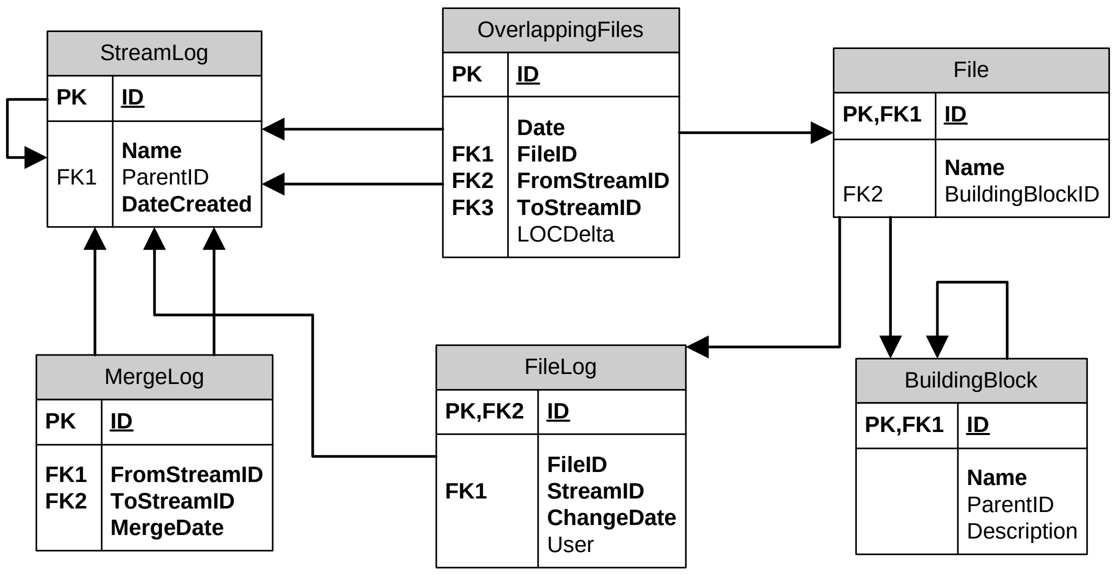

StreamLog
PK ID
FK1 Name ParentID DateCreated
OverlappingFiles
PK ID
FK1 FK2 FK3 Date FileID FromStreamID ToStreamID LOCDelta
File
PK,FK1 ID
FK2 Name BuildingBlockID
MergeLog
PK ID
FK1 FK2 FromStreamID ToStreamID MergeDate
FileLog
PK,FK2 ID
FK1 FileID StreamID ChangeDate User
BuildingBlock
PK,FK1 ID
Name ParentID Description

Figure 32: Datamodel metrics database prototype