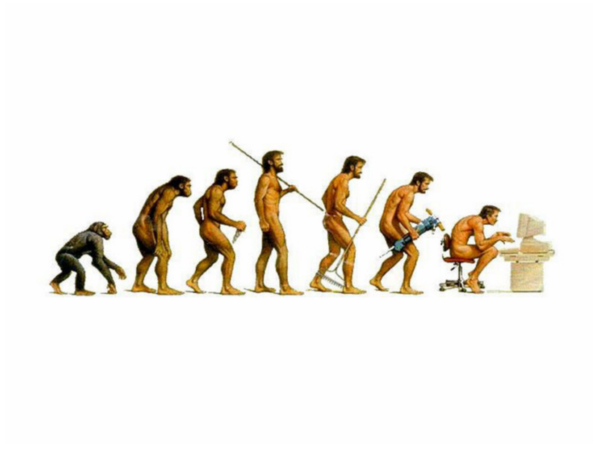


Evaluating Programmer Performance

visualizing the impact of programmers on project goals

Master Thesis

Master Software Engineering, University of Amsterdam

Author:
Hidde Baggelaar

Supervisor:
Prof. Paul Klint

27th August 2008

Universiteit van Amsterdam

Abstract

The evaluation of programmers is at best a hard task. Software projects often have goals like Maintainability and Testability. Regrettably, it is often unclear which developer worked on what file and how their change affected those goals. The amount of data that needs to be analyzed to get a clear picture is too huge to be easily interpreted. This Master Thesis examines the data that is required and explores several visualization techniques to find the best way to display this data. The EvoChart finally uses a decomposition visualization technique, enabling an evaluator to gain both a good overview of how programmers perform as well as detailed information on who worked on what and how this affected the project goals.

Preface

This thesis is the result of my participation in the Meta-Environment visualization project that was created at the CWI (Centrum voor Wiskunde en Informatica). It couldn't have been possible without the guidance of Professor Paul Klint, whom I would like to thank.

I would also like to thank Arnold Lankamp, Bas Basten and Machiel Bruntink who shared their experience and provided me with valuable insight.

Special thanks go out to Jurgen Vinju, whose enthusiasm was a good motivational force during the writing process.

My fellow students Qais Ali, Jeldert Pol and Jeroen van den Bos also deserve a note of thanks. Qais for his part in the start up phase of the project, Jeldert for helping me with various LaTeX problems and Jeroen for his many pointers and paper links.

Hidde Baggelaar August 2008

Contents

Chapter 1

Introduction

This chapter will explain a few of the terminologies that are used in this thesis. The purpose of this chapter is to define the terms as I have interpreted them. This chapter will introduce the visualization that was used in the evaluation. It will also describe my view on software terms of evolution, metrics, programmer evaluation and visualizations.

1.1 EvoChart

The EvoChart visualization is the result of this thesis project. It is a tool that enables an analyst to visualize how programmers worked on a system and how they performed, according to the project's goals and criteria. The tool creates a visualization of the program states, according to various software metrics, and the way they were changed over time. By linking the changing metrics to the software programmers that caused them, the visualization allows the analyst to evaluate the programmers based on the project goals. This is explained in more detail later on.

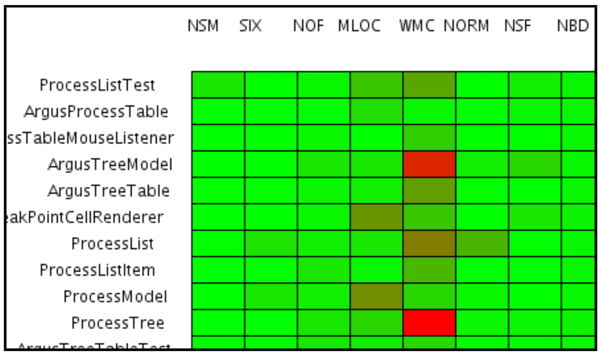


Figure 1.1: The EvoChart Visualization, displaying the current state of the Argus-viewer program

1.2 Software Evolution

"Life can only be understood backwards, but it must be lived forwards."

(Soren Kierkegaard)

Software changes over time. Even when the development process reaches the stage where the software can be shipped to the customers the product doesn't stop changing. Over time, bugs are discovered and fixed, requirements are changed and added and the resulting deterioration of the code can lead to refactorings.

From Program evolution: processes of software change [LB85]

> *"The addition of any function not visualized in the original design will inevitably degenerate structure. Repairs also, will tend to cause deviation from structural regularity since, except under conditions of the strictest control, any repair or patch will be made in the simplest and quickest way. No search will be made for a fix that maintains structural integrity"*
>
> (Lehman and Belady)

Lehman also observed the following laws for software evolution:

- **Law of Continuing Change:** software needs to change in order to stay useful
- **Law of Increasing Complexity:** the structure of a program deteriorates as it evolves

This implies that the longer a software system exists, the more deteriorated it becomes. Because of the deterioration and the increased complexity, the program becomes harder to maintain the longer it exists. This increases the cost to maintain and enhance the program, until it might eventually reach a point where it is better to rebuild a system from scratch because the previous version has become too complex and too deteriorated to maintain it any further.

But rebuilding a system takes a lot of time and resources. There are all sorts of risks involved in rebuilding a system, since during the development of the new system the old one will still be in use. If maintenance is stopped, newly found bugs will not be addressed which could lead to user dissatisfaction. This leads to a situation where it might be necessary to have at least a few maintainers of the old system during the development of the new one.

There is also the risk of repeating mistakes that were made during the development of the first system, especially if those mistakes have not been properly documented or worse, if the documentation is missing completely. The requirements documentation might still exist, but since the software has been changed over time, new requirements might have been added to the system without being recorded in the requirements documentation.

Even though most software has some form of initial documentation, it gets more and more detached from the actual implementation as time passes. In the software life cycle, the documentation is usually written in the beginning, before or in the implementation phase. When the code reaches a certain point, the documentation is usually not kept up to date because it is perceived as a waste of time.

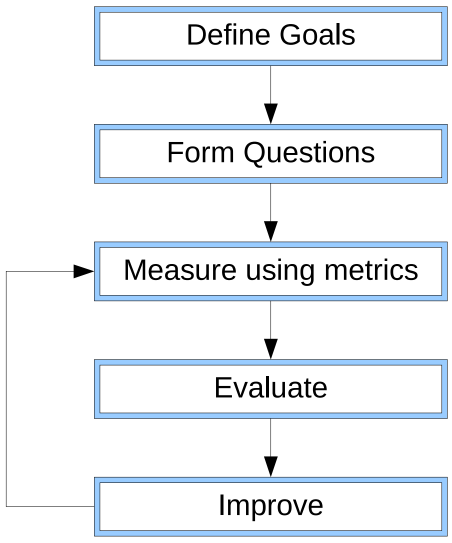


Figure 1.2: Based on the Goals, Questions and Metrics model as described by Fenton and Pfleeger [FP97] a software annalist will first determine the project goals. These goals are the basis for formulating questions about the software. To answer these questions, facts about the software are extracted and metrics are used. Using the gained knowledge, the software can then be improved. Since the software keeps evolving, the extracting, measuring and improving will be repeated during the software life cycle H

1.2.1 Detection of deterioration

Whilst these things eventually happen to most projects, it is unfair to simply blame the entire development team or accept this as a given fact. If there is a way to detect the program deterioration then two things can be done: If a developer has negative impacts on the program's quality and his team leader is made aware of this fact, the developer can then be coached to improve his skills. If a specific component is badly written and this can be detected, this component can then be refactored to increase the quality of the overall product.

To help the software engineer in understanding the state of the system and how it progressed in time, he needs to have information about the current and past states of the system. The amount of data depends on the total amount of states that have to be examined and the size of the system itself. Even for a small software system which was built in a short time frame, this will lead to large amounts of data that needs to be addressed. For instance, the Toolbus Viewer project "Argus Viewer", which took a group of 20 students a total of 4 weeks to make, leads to a large amount of data. Being able to interpret this data becomes a complex task for the software engineer, simply because of the amount of data that needs to be reviewed.

This leads to a situation where it becomes important that the data is filtered. To determine which data is relevant and which is unnecessary, Fenton and Pfleeger [FP97] propose to use the Goals, Questions and Metrics model as a tool for doing so. The software engineer will first have to know and understand the goals of the software project, which will then lead to questions about the system, e.g. if one of the project goals is to reduce maintenance cost, it could lead to questions like "which part of the software is the most complex? which part of the software is the largest? what is the trend of these parts, do they keep increasing in size/complexity or are they decreasing over time?". The goals and questions will help the software engineer in understanding which facts he needs to extract and which metrics he needs to calculate in order to comply to the project goals.

1.3 Metrics

According to Fenton and Pfleeger [FP97] the mathematical definition of a metric is *"a rule used to describe how far apart two point are. More formally, a metric is a function m defined on pairs of objects x and y such that m(x,y) represents the distance between x and y"*

In my paper I view the term metric as a measurement of the internal attributes and characteristics of the system. These attributes give an indication of the quality of a system. For instance, a complexity measurement like 'McCabe's Cyclomatic Complexity' is an indication of how complex a method is. When something is highly complex it has the negative effect of increasing the time it takes to maintain/modify it and it decreases the ease with which the method is understood by another programmer. The effects of complexity on programmer performance has been researched by Curtis [CSM79].

1.4 Visualization

To gain a better insight into the software evolution data, this paper investigates the usage of visualizations.

According to Ware[War00] there are several stages in the process of data visualization:

Collecting the data and storing it

Preprocessing the data into something we can understand

Displaying the data using display hardware and graphics algorithms

Perceiving the data, which is done by the human perceptual and cognitive system

> *"Ware defines visualization as a graphical representation of data or concepts, which is either an internal construct of the mind or an external artifact supporting decision making."*
>
> ([TM04])

Chapter 2

Motivation

Evaluating software developers can be a difficult task. Teachers and managers are faced with a lot of uncertainties. Though it might be clear how the programs meet their project goals, understanding how each developer played his part in their construction is a daunting task.

2.1 Programmer Performance

While it is easy to measure the software quantity, e.g. the amount of lines written by a programmer, measuring the software quality is more complex. Evaluating software developers on the amount of lines of code they write will not establish which programmer is more skilled or even working harder. For instance, writing a complicated algorithm might require a lot of time and result in only a few lines code, making the quantity approach unfair.

Evaluating developers by the amount of code they write would also lead to bad situations. A programmer could be refactoring a lot of the code, making it more compliant with their project goals, which could mean that he is only moving or deleting parts of the program, but he would not produce a lot of new lines of source code. It would be unfair to evaluate him on the amount of source code lines that he writes, since he would get a bad evaluation while he would still be doing a good job. And when developers learn that they are evaluated by the amount of code they write, it suddenly becomes more tempting for them to not write as concise as possible. Loops are suddenly a great way to get a good evaluation, simply by not writing them as for or while blocks but by copy pasting the looping code as many times as needed.

Still, developers need to be evaluated properly. In an educational situation it is often the case that the program gets graded instead of the individual developers, leading to a uniform grade for all the developers. The good developers will get the same grade as bad developers, which hardly stimulates them to do their best.

The grading itself is also a means of helping a programmer to evolve his skills. By detecting the amount of mistakes a programmer makes over time a trend of increasing errors could be detected. This could trigger the programmer's team leader to see if he can help the programmer improve on his skills.

2.2 Code Quality

To establish the quality of the code, the program's project goals are used as the criteria. When a change in the code leads to a better assessment of a project goal, the code makes a positive contribution and vice versa. It is however insufficient to simply look at the program's last state or version and make an evaluation solely on that. Not only would it be hard to determine who wrote what, it would also miss the possible improvements or deterioration in the developer's abilities.

It is much better to look at more states of the program, assessing it's compliance to the project goals for all those states and linking them to the developer that is responsible. The amount of data that this generates can be huge, depending on the number of versions and the amount of criteria.

2.3 Research Question

My research question is about finding a visualization that will be helpful in evaluating software developers. What kind of visualization is best for evaluating software developers and their impact on the project goals?

The reason that a visualization is used for evaluating the software developers is based on two facts:

1. The amount of data that needs to be mined is huge. For every state (e.g. revision) of a software system a number of metrics need to be extracted. My case study, which was coded in 4 weeks, consisted of roughly 1000 revisions. I collected 26 metrics for each state, leading to 26000 facts. Larger systems would generate even more data.
2. The visualization of large amounts of data helps a human being in processing and understanding. This is based on research done by Tory and Muller [TM04]. They claim that a good visualization can:
    - Enable parallel processing (by a person, using the subconscious of the human mind)
    - Offload work to perceptual system (the computer can take over (minor) tasks)
    - Reduce demands on human memory
    - Store large amounts of data in an easily accessible form
    - Group related information, enabling easy search and access.
    - Represent large data in a small space.
    - Reduce task complexity (because of the imposed structure on data and tasks)
    - Allow visual information recognition (which can be easier than recalling it)
    - Allow recognition of higher level patterns (because of selective omission and aggregation)
    - Allow monitoring of a large number of potential events

- Allow interactive exploration through manipulation of parameter values
- Increase pattern recognition (by manipulating the structural organization of the data)

2.3.1 Goal-Question-Metric

The evaluation in my thesis is based on the Goal-Question-Metric approach. Every software project has it's own goals that the project is evaluated on. The goals are therefor contextual and strongly depend on the project customers and the project management. The goals will lead to questions that can be answered by doing measurements on the program. The evaluation of the effect that a programmer has on the project goals leads to questions like:

- How many lines of code does a programmer write?
- How many lines of commentary does a programmer write?
- How does a programmer affect the complexity of the system or its subcomponents?
- How does a programmer affect the cohesion of the system or its subcomponents?
- Does a programmer work alone or does he work with others?
- Does a programmer make many small commits or does he make a few large ones?

These questions, if formulated properly, can be measured using program metrics. Simple questions like the amount of lines of code a programmer writes each day can be measured by extracting facts like the line count of each source code file, while more complicated questions like whether or not a programmer is creating overly complicated code can lead to the calculation of the McCabe Cyclomatic Complexity metric for each source code file.

The assumption made in this process is that a good programmer will cause a good impact on the system's metrics or possibly just a minor negative impact, while a bad programmer will cause a large negative impact on the system's metrics.

Chapter 3

Related Work

This Chapter has an overview of related visualizations and other programs that try to help a software analyst in either understanding the evolution of a software system or grading the developer of a software system. The purposes of each of these related visualizations/programs are explained, as well as why they were not used as a basis to evaluate the performance of software programmers.

3.1 SeeSoft

The SeeSoft software visualization system is essentially a visualization of the source code lines, grouped together by filename. Every line of code is visualized by a thin line within the group. The lines are color coded by a statistic of interest. In Eick's case studies [ESEES92], the color was determined by the modification request that a line belonged to or by the date on which the line was created, but it could be linked to any metric.

The SeeSoft visualization is a good tool for analyzing source code at the line level. It could possibly be adapted for evaluating a software developer, but because of the line abstraction it's not really suited for that purpose. The negative aspects were judged to outweigh the positive aspects, were the SeeSoft tool used for programmer evaluation.

Positive Aspects

Location of good and bad source code: The Seesoft software can locate the source lines that are either judged good or bad, according to the project goals. This makes it easier to get examples of good or bad work.

Overview of the entire code base: The Seesoft software would show which parts a developer worked on. It would also give an indication of how much code of his is still remaining.

Negative Aspects

Wrong level of abstraction: The Seesoft software is intended to evaluate source code lines. It is true that it is possible to grade every source code line written by a certain developer according to the project goals, but it would require the evaluator to either average the scores, pick the best or worst case scenario's or use some other kind of evaluating system.

One metric at a time: The Seesoft software only uses a single metric. To use the Seesoft visualization as an evaluation tool, the metric would have to be a computed total of all the goals, possible with weights for the different goals to be able to mark some of the goals as more important then others. This would give a good evaluation, but because of the loss of information, it would be unclear what the grade would be based on. It would also be hard to determine what a programmer would have to improve to get a better evaluation the next time.

No evolutionary information: The Seesoft software doesn't show the evolution of the code. It doesn't show how the programmer has progressed over time. Information about whether his work is getting better or worse will not be shown.

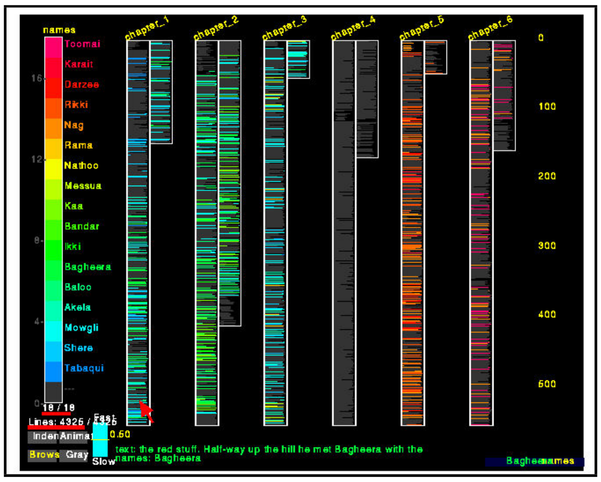


Figure 3.1: The Seesoft Visualization displays the source code based on the source code lines and their statistics. Lines are grouped by file, and a statistic is used to color the appropriate lines of code.

3.2 CVSScan

CVSScan is a program that visualizes a source code file and it's evolution. CVSScan is based on *"a new technique for visualizing the evolution of line-based software structures, semantics and attributes using space-filling displays"*[VTvW05].

The CVSScan program visualizes a single file, namely the evolution of the file and the authors that contributed. It's purpose is to discover how and why a source code file has been changed during the course of its life. It uses several linked views to show not only a global overview of the evolution, but also the detailed source code of a specific moment in time.

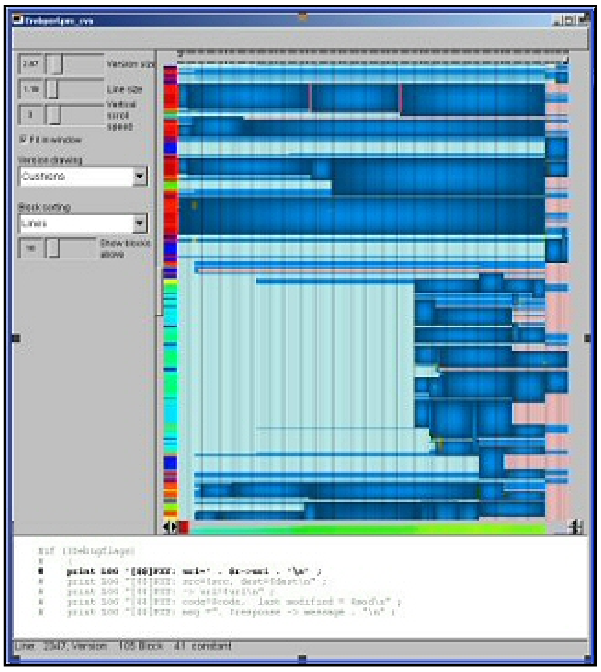
CVSScan is similar to the SeeSoft visualization in that it is a line based visualization. The real difference between SeeSoft and the CVSScan program is the scope of the visualization. The SeeSoft visualization creates a line based graphic of the entire code base, while the CVSScan program focuses on a single source code file.

Positive Aspects

Source code oriented: The CVSScan visualization is based on visualizing the code itself and the way it evolved over time. This would allow the evaluator to see the actual changes that a developer has added to the code.

Negative Aspects

Narrow scope: Because of the source code orientation of the CVSScan visualization, an evaluator can only look at one code file at a time. If a project has a lot of files and a developer worked on a lot of those files, it will be harder to get an accurate impression of his skills.



```
#if ($debugflags)
    {
    print LOG "[$$]FXX: url=" . $r->uri . "\n" ;
    print LOG "[$$]FXX: [illegible]" ;
    print LOG "[$$]FXX: [illegible]" ;
    print LOG "[$$]FXX: code=$code, last modified = [illegible]" ;
    print LOG "[$$]FXX: msg [illegible] message . "\n" ;
```

```
Line 2347; Version 105 Block 41 constant
```

Figure 3.2: The CVSScan visualization displays the evolution of a single source code file. The visualization shows every different version of the code and can even trace where the code originated and how much of it is still left.

3.3 Chronia

Chronia is a program that creates an ownership map of project files. The overview shows the project files and the commits made by the team's developers. Ownership is determined by the percentage of ownership an author has. When a developer made enough changes to a file, the ownership is transferred to him. More on Chronia can be found in [See06] and [GKSD05]

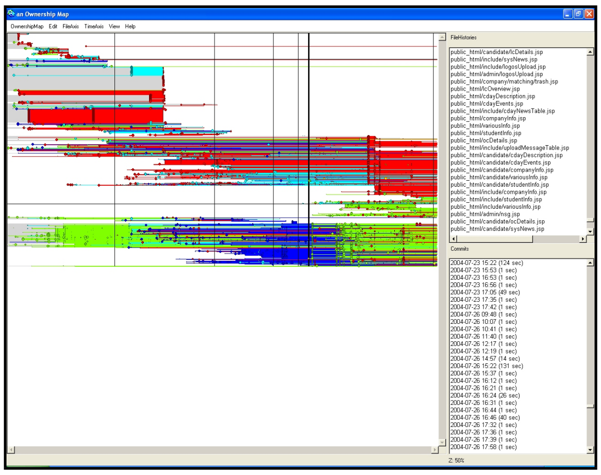
While an ownership map is useful for finding the person to go to when a part of the system is in need of maintenance or needs added functionality, it doesn't really determine the actual state of the software system. It does show who wrote the most code of a certain file during a certain period in time, but that is an unfair measure to grade programmers. It is perfectly viable that a programmer is working on nothing but refactorings and other quality enhancing jobs, where he has to modify lots of files in very small ways (for instance, to decrease coupling). The Chronia map might not even register as an owner of a file even once, while he would still do a good job.

Positive Aspects

Ownership: Chronia makes it easy to see who wrote the most code of a given file. It even shows the way the ownership changed over time. Whenever there is a problem with a part of the system, this makes it easy to locate who to go to

Negative Aspects

Unfair measure of performance: Programmers who refactor and improve coding quality might never obtain file ownership, while still doing a good job.



```
public_html/candidate/lcDetails.jsp
public_html/include/sysNews.jsp
public_html/include/logosUpload.jsp
public_html/admin/logosUpload.jsp
public_html/company/matching/trash.jsp
public_html/lcOverview.jsp
public_html/cdayDescription.jsp
public_html/cdayEvents.jsp
public_html/include/cdayNewsTable.jsp
public_html/companyInfo.jsp
public_html/variousInfo.jsp
public_html/studentInfo.jsp
public_html/ocDetails.jsp
public_html/include/uploadMessageTable.jsp
public_html/candidate/cdayDescription.jsp
public_html/candidate/cdayEvents.jsp
public_html/candidate/companyInfo.jsp
public_html/candidate/variousInfo.jsp
public_html/candidate/studentInfo.jsp
public_html/include/companyInfo.jsp
public_html/include/studentInfo.jsp
public_html/include/variousInfo.jsp
public_html/admin/nsg.jsp
public_html/candidate/ocDetails.jsp
public_html/candidate/sysNews.jsp
```

```
2004-07-23 15:22 (124 sec)
2004-07-23 15:53 (1 sec)
2004-07-23 16:53 (1 sec)
2004-07-23 16:56 (1 sec)
2004-07-23 17:05 (49 sec)
2004-07-23 17:35 (1 sec)
2004-07-23 17:42 (1 sec)
2004-07-26 09:48 (1 sec)
2004-07-26 10:07 (1 sec)
2004-07-26 10:41 (1 sec)
2004-07-26 11:40 (1 sec)
2004-07-26 12:17 (1 sec)
2004-07-26 12:19 (1 sec)
2004-07-26 14:57 (14 sec)
2004-07-26 15:22 (131 sec)
2004-07-26 15:37 (1 sec)
2004-07-26 16:12 (1 sec)
2004-07-26 16:21 (1 sec)
2004-07-26 16:24 (26 sec)
2004-07-26 16:31 (1 sec)
2004-07-26 16:44 (1 sec)
2004-07-26 16:46 (40 sec)
2004-07-26 17:32 (1 sec)
2004-07-26 17:36 (1 sec)
2004-07-26 17:39 (1 sec)
2004-07-26 17:58 (1 sec)
```

Figure 3.3: The Chronia visualization displays an ownership map. The overview displays the files and the authors that worked on them. Every line depicts a file, while every color depicts an author.

3.4 AutoGrader

AutoGrader is a tool that was researched by Helmick [Hel07]. AutoGrader focuses on grading the functionality of the code and was designed for programming class teachers to quickly test their students work. The focus of the tool is just that, functionality. It imposes a set structure that the students must use, namely a series of interfaces that they must implement, for which the teacher can then design unit tests for.

While functionality is indeed a very important aspect of a program, such a test doesn't factor in the quality of the code that was written. For big projects where students have to find the software requirements themselves, it would be very hard if not impossible to define a set of interfaces beforehand to test the functionality on. While AutoGrader is a good tool for testing students homework assignments, it would not be able to be used as an evaluation tool outside of that setting.

Positive Aspects

Testing of functionality AutoGrader can test if the code that a student writes functions as it should, as long as the teacher has imposed a set of interfaces that must be implemented and has devised a set of test cases.

Negative Aspects

Only usable for homework validation Because of the limitations, this tool cannot be used in big projects where students work together as a team, especially when discovering the requirements is part of the project.

No quality check AutoGrader only checks the code's functionality, not how well it was written.

Chapter 4

Background

This chapter explains the background elements that were used to create the EvoChart visualization. To facilitate the validation process, the EvoChart was integrated in the Meta-Environment. The Prefuse Graphics library was used as the visualization framework.

4.1 Case Study

To validate my thesis I used data from the Argus-Viewer project, a Toolbus viewer. The Toolbus viewer was made by a group of 25 software engineering master students, in an attempt to create a debugging tool for Toolbus programs. This case study was selected because of the high availability of both the data of the project itself and the programmers / managers that had created it. The Argus-Viewer project has been made public through the Google Code server, containing the SVN repository that has been used as the main development server.

4.2 Meta-Environment

As is described by van den Brand et al. [vdBBE$^+$07]: The Meta-Environment is an Integrated Development Environment with the purpose of being a platform for language development, source code analysis and source code transformation. The Meta-Environment gives their users the ability to create their own languages, to better understand their software and to evolve it using code transformations.

The Meta-Environment is an IDE that is used by the Dutch Centrum voor Wiskunde en Informatica (Center for Mathematics and Computer Science) (CWI) to transform and analyze programs. One of the main tasks of the Meta-Environment is the extracting of software facts. These facts are needed in the software evolution analysis process, which makes the Meta-Environment an ideal environment for software evolution research.

The Meta-Environment IDE is based on the Toolbus architecture, which allows them to incorporate modules from a number of different programming languages. One of the modules incorporated is the Prefuse Library. Because of the duration of the research project the decision was made to use the Prefuse Library instead of trying to find a better alternative.

4.3 Prefuse

Prefuse is a visualization toolkit based on Java [HCL05]. The framework creates a visualization component that can be embedded in any Java application or applet. Prefuse creates dynamic 2D visualizations by first collecting data, transforming it to a Prefuse data table, subjecting these tables to visual mappings and then rendering the end result to create a view.

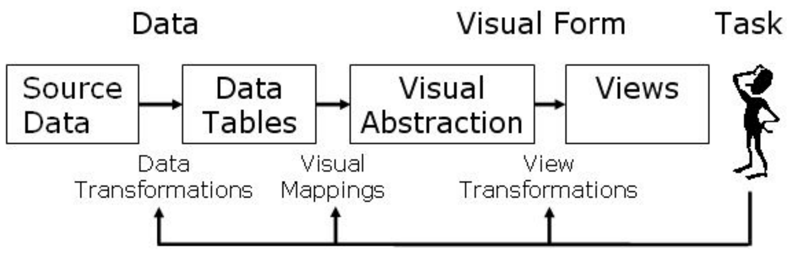


Figure 4.1: The Prefuse Toolkit structure explains the steps required to create a Prefuse visualization

4.4 Subversion

Subversion, or SVN for short, is an open source version control system, built to rival the older CVS (Concurrent Version(s/ing) System). SVN is used to store the different states of a program as it changes through time. Because the changes are linked to the developer that made them, it is a good repository for finding the necessary facts needed for the visualization.

More on subversion can be found in Collins' paper on SVN [CS02].

4.5 PMD

PMD [PMD] is a problem searching tool for Java. PMD allows the user to scan Java source code for possible defects, and it will point out the possible flaws based on rules defined by the user. PMD uses its rules to scan the source code's AST (abstract syntax tree) for the user defined flaws.

The PMD tool has been used in prior research by Helmick [Hel07], who created the Auto-Grader tool as a PMD add on. Helmicks research confirmed that PMD is a useful tool for grading programmers. PMD can even be used in an continuous integration environment, making it an ideal tool for extracting metrics.

This paper treats the PMD rules as metrics. Since a PMD rule detects possible flaws and points out every occurrence, it is numeric in nature. By adding up the detected flaws a natural number is obtained, which will be recorded as the value of the metric.

Chapter 5

Research Data

This chapter contains information about the process of creating the EvoChart Visualization. It begins with a section on the relevant data that need to be visualized. First the requirements of the data are determined, after which the data are extracted, filtered, and stored in several ways.

5.1 Requirements

There are several ways to evaluate programmers. One of these ways is to take a look at the end result of his work and grade the programmer based on how well it is written. This is however a snapshot evaluation, since it doesn't take into account how the software was changed over time.

If the evaluation's purpose was to give a simple grade to a programmer, it would be enough to evaluate the end result. However, in a learning environment where the evaluation is used to assess a programmer and guide him to improve himself, a more detailed evaluation has more merit. There are situations where having information about the evolution of the program will lead to a better insight in someones progress, and this could lead to a better insight into how to help a programmer evolve his skills.

It is therefore important that the data not only records the metrics, but also the moment it was measured. But does it matter what the actual location of the source is? Would it be enough to simply grade the entire program as a whole, instead of looking at the source files themselves? Perhaps, but even then some important data would be lost. By measuring the place of change, patterns of co working might become apparent, which help in detecting how someone operates within a team.

It's also good to know where the possible weaknesses are, instead of simply knowing that the weaknesses exist. This will help in identifying the dangerous areas of a program. Though it doesn't help the evaluation of the programmers per say, it does give a better understanding in the total program as a whole. When grading, it might make a difference whether a program is overall fairly average or if the program is excellent in most areas and only bad in a few. The latter situation would suggest that something went wrong, and that the potential of the software is higher than in the former.

To understand how a programmer affected a given metric, his change will need to be recorded. A bad programmer will tend to worsen the project's evaluation by making mistakes, while a good programmer will have almost no bad effect on them. He might even fix a few of the mistakes that others have made.

The data should therefore consist of five columns, with the following values:

1. The name of the programmer
2. The time of extraction
3. The name of the source code that is being looked at (class, package, file or some other container)
4. The name of the metric/fact extracted
5. The difference between the metric value before and after the programmer worked

5.1.1 Filtering

An N-dimensional image requires at least N+1 data columns for it to be able to draw a meaningful image: 1 column for every axis and 1 column for the represented value. The EvoChart visualization, which creates a 2 dimensional image, would only need 3 columns of data. However, the EvoChart visualization uses a slicing technique to transform a 4 dimensional data object into a 3 dimensional data object. It does this by applying a data filter on the fourth dimension. Every row that contains the correct value in the filtered column will be transformed.

This filtering technique was also used to create bar charts of the data. The bar charts were used in verifying the effectiveness of the EvoChart visualization. Because a bar chart is also a 2 dimensional image, the same filters applied.

The data is filtered in two distinct ways: by Goal and by Programmer

Filtering by Goal

By using the project goals to group the data, the remaining data reflects the way a programmer affected this goal. This allows the visualization to create an oversight that compares programmers to each other, helping the analyst to grade the programmers. This is the first step of the programmer evaluation, since it helps in detecting the outliers: the really good and the really bad programmers.

Filtering by Programmer

By using the programmers to group the data, the remaining data reflects the way a programmer worked. This helps to detect what the programmer is doing wrong and if his skills are improving or decreasing.

5.2 Determining the Metrics

The metric extraction process is intended as the basis for the validation of the EvoChart visualization. Based on the assumption that the extracted metrics are a good indication of the compliance of the software state to the project goals, they can be used as a means to evaluate how well the individual developers were adhering to the project goals.

The tool used to calculate the metric information is "PMD", which is a metric calculation program for Java based applications. Because the validation is mainly intended to prove the effectiveness of the EvoChart visualization, there is no additional gain to validate whether or not the metric calculation is done correctly by the chosen tool. It is therefor simply assumed that the metric calculator works as it claims.

The Argus-project, which is the case study used in the validation process, was a software engineering project of the university of Amsterdam. The criteria for the evaluation of the project are the basis for the metric extraction process. In the project itself the entire code base was given a single grade, which was then distributed among all the developers that contributed. In a more desirable environment, a better programmer would receive a much better grade than a bad programmer.

The required information to make such a distinction between programmers would require a lot of information. Who changed what and what was the overall impact of those changes? Did the programmer work alone or as a team player? The information required would have to contain all the states previous to the change the programmer made and the new states that the programmer introduced. By comparing each previous state to the new state, the programmer's impact would be discernible.

To discern whether or not a given impact is good, bad, or simply ungraded, the criteria for the Argus-viewer project was checked for each of those states. The Argus-viewer was evaluated on the following criteria:

Design: use of abstractions, testability, cohesion, coupling, use of layers, modularity

Style: layout, proper names, duplication, dead code, commentary, succinctness

Defense: test coverage, use of asserts, error handling, exception handling

Bugs: number of discovered bugs

Some of the criteria were too complex to make them easily measurable. These criteria are mostly on aspects of the design and architecture of the program. Measuring the correct use of these criteria might be possible, but would be too complex to solve in a reasonable amount of time. The unused criteria are:

- Use of abstractions
- Use of layers
- Modularity

The remaining criteria were then translated into the following project goals, with the appropriate criteria used a subgoals for each main goal. For each of the goals an appropriate metric was used to gain an insight into the compliance of the software to the goal. These metrics were obtained from the “PMD” metric calculation tool. The table below only contains the number of metrics that were obtained per criteria, the whole list of used metrics and their explanation can be found in the Appendices.

Table 5.1: Used Metrics

Goal	Criteria	Nr. of Metrics
Maintainability	cohesion	2
	coupling	3
	proper names	18
	duplication	3
	dead code	4
	commentary	2
	succinctness	23
	layout	4
Robustness	error handling	7
	exception handling	15
Testability	testability	2
	test coverage	1
	use of asserts	7
	number of discovered bugs	6

5.3 Extracting the data

The data was gathered from the Google SVN Repository that was used in the Case Study. The states between revision 175 and 1093 were extracted. Revision 175 is the beginning of the actual program, the states before this revision were used for testing the continuous integration server and for writing the numerous wiki pages. Revision 1093 is the head revision of the case study at this time.

The first revision contains 2 java files, an application file and a test file. The metrics for these files were calculated and stored. After that the extraction program used an SVN client to get the next version of the program. For every java file that was changed, new metric values were calculated and recorded. Deleted files were treated as files with metric values of 0.

The written extractor wasn’t perfect however, since it only recorded the filenames. The very few times when a java class existed in two packages at the same time, the metric extractor would only record the changes of one of these files. Because this only occurred in rare occasions, the error percentage was accepted. Adding the changes up led to a total of 1089 errors, calculating the errors using only the latest revision resulted in a total of 1066 errors, making the margin of error 2.2%.

The extracted data was stored in a comma separated values file, which would allow an easy conversion to several different data formats. For the EvoChart visualization, this data was converted into the RStore format. For the validation phase, MS Access was used to convert the CSV into a Microsoft DataBase file.

Program 1: This CSV example shows how the data was recorded after the extraction phase. Developers have been anonimized

```
...
John Doe,Maintainability,MainApplicationTest,180,0
John Doe,Maintainability,DataCommunicatieController,180,0
John Doe,Maintainability,MessageSequenceChartPanel,180,1
John Doe,Maintainability,ProcessListPanel,180,1
John Doe,Robustness,MainApplicationTest,180,0
John Doe,Robustness,DataCommunicatieController,180,0
John Doe,Robustness,MessageSequenceChartPanel,180,0
John Doe,Robustness,ProcessListPanel,180,0
John Doe,Testability,MainApplicationTest,180,0
John Doe,Testability,DataCommunicatieController,180,0
John Doe,Testability,MessageSequenceChartPanel,180,0
John Doe,Testability,ProcessListPanel,180,0
...
```

5.3.1 RStores

The EvoChart requires the data to be in the form of an Rstore Fact. This data format was chosen because it is the main data format that is in use by the Meta-Environment. RStores are relational stores, which contain the data that the Meta-Environment uses, stored in separate Facts. Each RStore Fact is stored in it's own RTuple.

The Meta-Environment uses these RStores to supply the visualizations with the data they need. The Meta-Environment is currently built in such a way that a visualization only receives one Fact at a time. The data, after being filtered, was stored in such a way that each portion of filtered data was stored in a single fact.

Program 2: RStore example. Two Facts can be seen, the DEV:John Doe Fact and the GOAL:Maintainability Fact.

```
rstore([
  rtuple("DEV:John Doe",
    relation([str,str,str,str]),
    set([
      ...
      tuple([ str("Maintainability"), str("MainApplicationTest"),
              str("180"), str("0") ]),
      tuple([ str("Maintainability"), str("DataCommunicatieController"),
              str("180"), str("0") ]),
      tuple([ str("Maintainability"), str("MessageSequenceChartPanel"),
              str("180"), str("1") ]),
      tuple([ str("Maintainability"), str("ProcessListPanel"),
              str("180"), str("1") ]),
      ...
    ])
  ),
  ...
  rtuple("GOAL:Maintainability",
    relation([str,str,str,str]),
    set([
      ...
      tuple([ str("John Doe"), str("MainApplicationTest"),
              str("180"), str("0") ]),
      tuple([ str("John Doe"), str("DataCommunicatieController"),
              str("180"), str("0") ]),
      tuple([ str("John Doe"), str("MessageSequenceChartPanel"),
              str("180"), str("1") ]),
      tuple([ str("John Doe"), str("ProcessListPanel"),
              str("180"), str("1") ]),
      ...
    ])
  )
])
```

Chapter 6

Visualization of the data

This chapter explains the attributes that visualizations have and why they are useful as a tool to understand large amounts of data. It also shows what the EvoChart requirements are and how the visualization was created.

6.1 Visualization attributes

Depending on what the user wants to review, the data is grouped either by project goal or by programmer. Data that is grouped by project goals will have 4 data columns, containing information about the programmer, the location, the revision number and the value of the goal. When the user wants to review the performance of a single programmer, the data is grouped likewise. This leads to 4 data columns containing the goal, the location, the revision number and the value of the goal.

The data suggests that a 3-dimensional image is required: The first three columns contain the axis values where the last column contains the actual measurement. If the data would be represented as a cube filled with colored dots, the first three columns would determine the placement of the dot where the last column would determine the actual color of the dot. Using a 3-dimensional image would however introduce new problems: Occlusion and Navigation.

In a 3-dimensional world it is easy to place separate objects in such a way that one prevents the other from being seen. Even if the camera is able to move around the objects and has the ability to rotate, it is still possible to hide an object by placing large enough objects around it, enclosing it completely. Because the data that drives the creation of the objects and their placement is generated it can not be guaranteed that such a situation cannot occur. Adding techniques like transparency would prevent this, but it would make the visualization more complex to understand by the analyst because of the fuzziness such a technique would add to the objects.

While in a 2-dimensional world the camera is locked in a top-down view, in a 3-dimensional world this camera should be able to move and rotate along all axis. The moving and rotating along the axis is necessary in order to be able to 'look behind' objects, to minimize the occlusion aspect. Because of the added movement complexity, it becomes easier to get lost in the scene.

The addition of a 3-dimensional visualization to the meta environment would also pose certain technical difficulties. It would require the selecting and incorporation of a open source 3-dimensional toolkit able to display it's graphics on a Java Swing Component. Because of the time constraint imposed on the research and the added occlusion and navigational complexities, the notion of using 3-dimensional graphics was discarded. The EvoChart visualization would instead be based on a 2-dimensional image, implemented with the already existing Prefuse framework.

6.2 Cognitive Science

In order to make a good decision on which visualization to implement, the attributes and aspects of these visualizations need to be evaluated. Cognitive Science has long studied these attributes and aspects of visualizations. One of the cognitive science theories is the Gestalt Psychology.

The Gestalt School of Psychology examines the way the human mind perceives different shapes and how they affect the spectator in processing them. According to Colin Ware [War00], *"the work of the gestalt psychologists is still valued today because they provide a clear description of many basic perceptual phenomena."*

6.3 Gestalt Perception: Law of Prägnänz

The gestalt law of Prägnänz is the fundamental principle of gestalt perception. The law says that our mind tends to order our experiences in a regular, orderly, symmetric and simple manner. The law gives an indication how a visual image will be perceived by a human spectator, making them a useful tool for selecting useful visualizations. The law of Prägnänz can be broken down into seven sub laws:

Law of Proximity: Elements that are close can be perceived as a group

Law of Similarity: Similar elements are grouped into collective entities

Law of Continuity: The mind fills in gaps in visual patterns

Law of Symmetry: Symmetrical elements are perceived collectively, despite the distance

Law of Closure: Lines or shapes that are not connected can still be perceived as a single shape

Law of Relative Size: Smaller elements of a pattern are perceived as object while larger elements are seen as the background

Law of Figure and Ground: The figure is something in the foreground, while the ground makes up the background. The Figure and Ground law is that those two can switch, according to your own personal focus.

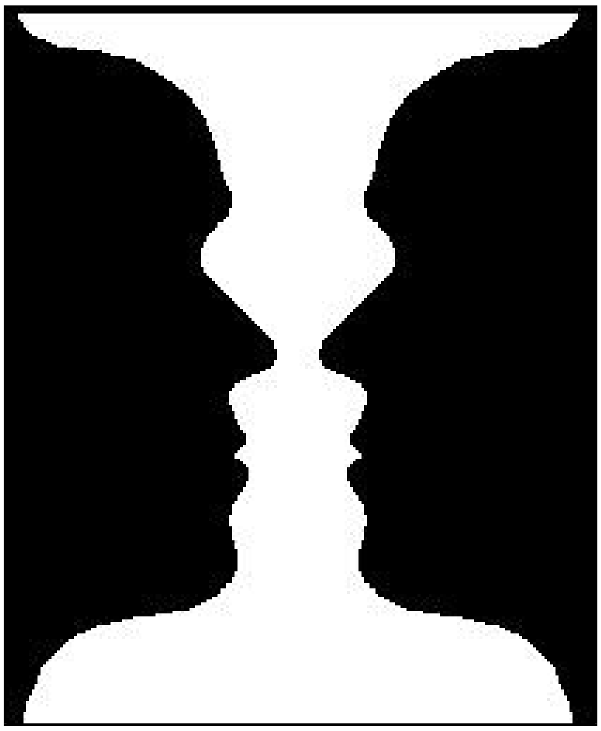


Figure 6.1: Ruben's Vase is a well known example of the Law of Figure and Ground. The image can be perceived as a vase or as two faces, but not both at the same time.

6.4 Visualization Requirements

6.4.1 Selectable colors

A good portion of the human population is color blind. The WrongDiagnosis website [Dia] claims that at least 10% of the male population have red-green color blindness. It is important to realize this while creating a visualization. By making the colors selectable it should be possible for a semi color blind person to chose the colors he can still differentiate. A total color blind person could even opt to select a greyscale palette, increasing his comfort with the use of the visualization.

6.4.2 Overview of evolution

When evaluating a developer, it is important to understand if the developer is slowly improving or is doing just the opposite. When the visualization shows that the developer is slowly increasing his skills, it's a good sign that he is well on his way and probably needs nothing but an occasional encouragement to keep him on this path. When the opposite is true, it will be a good indication that the developer should be talked to, to halt this declining process.

6.4.3 Optional filtering by developer

When evaluating the developer it is important to filter out all the unnecessary information. This will make the evaluation process a lot simpler, since the evaluators mind isn't distracted by it.

This requirement should appear as an option, so that the evaluator can choose to see either the progress of all the developers or only of just one. By having a view of all the developers the evaluator will get a better overview of the whole group.

6.4.4 Multiple perspectives: component, metric or time views

The visualization should contain the following views:

component There should be a view where you can see how good the system and its sub-components are adhering to the project goals and how this evolved

metric There should be a view where you can see where the metric measurements have taken place, in order to pinpoint the bad areas of the program

time In a situation where an end product needs to be evaluated, the last version of the completed program should be reviewed

6.4.5 No occlusion

Some visualizations like a line chart have the possible occurrence of occlusion. In the case of the line charts, the lines could overlap making some of them harder to see or even invisible. Since this makes the visualization unclear and could lead to confusion, it should be prevented.

6.5 Selection Process

Knowing that the EvoChart visualization would be based on a 2 dimensional visualization, there was still the question of creating the visualization itself. To that end, the domain of visualizations was explored, to obtain a better understanding of what would work best for visualizing evolutionary data. This was done by looking for a taxonomy that would clearly divide the visualization domain into easily comprehensible parts. Because of the complexity of the visual domain, the domain itself having an infinite amount of possible visualizations, only restricted by the artist's imagination and creativity, lots of taxonomies exist.

To be able to better choose a visualization to base the EvoChart on, these taxonomies were used as a basis for understanding what there is to chose from. The taxonomy based on Axiomatic Geometry was chosen, because it made the distinction between visualizations types based on the shapes that were drawn, making it relatively easy to place a visualization in the correct category.

6.5.1 Axiomatic Geometry based Taxonomy

Axiomatic Geometry was introduced by Euclid, who introduced certain axioms/postulates, expressing the primary properties of points, lines and planes. The basic concepts lead to different categories that use these concepts. Each category has positive and/or negative aspects.

- Basic
  - Point
  - Line
  - Plane
  - Text (doesn't actually exist in axiomatic geometry but is added for personal understanding)
- Categories

  textbased: Good visualization of actual content and details. Bad for overviews.

  chart: Good for combining lots of statistics

  map: Good for creating an overview, but hard to discern between elements

  nodes & links: Good for showing structures, containment, other relations

  landscape: Good for creating detailed overviews

  matrix: Good for data aggregation and discovery thereof

The categories in Table 6.1 were not deemed to be useful as a good visualization base.

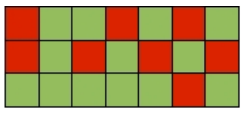
Table 6.1: Rejected visualizations

Visualization	Explanation
textbased	The textual representation of the data already exists.
map	The Treemap has no way to encode the timing information necessary.
nodes & links	Graphs and other nodes & links like trees lack a clear perception of time. This makes it hard to encode the timing information in the visualization.
landscape	The landscape is a 3D visualization in nature. It suffers heavily from occlusion, but it also requires 3D drawing technology that is currently not implemented in the Meta Environment. The implementation time would exceed the time allotted.

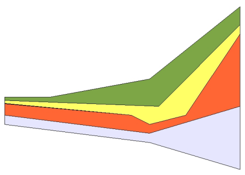
The remaining two categories, charts and matrices, were then investigated. Five visualization types were proposed as candidates for the EvoChart visualization. The proposed visualizations were found during a creative brainstorming session in which these possible candidates came forward. Because visualizations have a very creative nature by themselves, and since the domain is infinite in nature, the selection process was limited to these five.

6.5.2 Choosing the Visualization

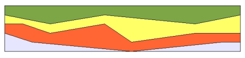
Decomposition chart: The decomposition chart is a visualization that creates a matrix of squares, using color to represent the value of the metric.



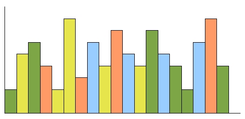
Flowchart with absolute values: The flowchart with absolute values creates a fan like visualization, using colors to represent the author and size to represent the metric value. Because the numbers are absolute, the width of the visualization starts small and becomes bigger when the application grows in size. Zooming the visualization so that the biggest end will fit on the screen would make the smaller end crowded[1].



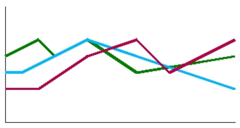
Flowchart with percentage based values: The flowchart with percentage based values creates a spacefilling banner like visualization, using colors to represent the author and size to represent the metric value. Because the numbers are percentages, the width of the visualization remains the same. The more the application grows, the more lines appear and the more crowded[1] the visualization becomes.



Barchart: The barchart is good for creating overviews of the performance of the individual developers, but makes comparisons harder. The more developers that are viewed at the same time, the more crowded[1] the visualization becomes.



Linechart: The linechart is good for showing lots of developers at the same time, but because lines can overlap and cross each other, the visualization becomes crowded[1].



[1]Crowded visualizations have lots of information in a small area, making it hard to discern between elements

These visualizations were judged using the following criteria:

Information How much information can they display at the same time? The amount of information that can be displayed was subjectively judged on a scale from 1-5, 1 being the lowest amount and 5 being the highest amount.

Crowded Can the visualization get crowded? Crowded visualizations are bad, because they make the results harder to understand.

Occlusion Can the visualization have occlusion? Occlusion is bad, because it hides information.

Technique What kind of filling techniques can be used? The more techniques that can be applied, the more creative the visualization can be, which could lead to interesting results where more data can be analyzed at the same time.

The filling techniques that were examined during the selection process were:

Color Mapping: A color is used to represent a value

Light Mapping: The lightness of the colors are used to represent a value

Texture Mapping: A texture is used to represent a value

Space Filling: The most of the screen can be used in the visualization

Because of the amount of information that can be shown in a decomposition chart and because it isn't crowded, has no occlusion, and can benefit from all the examined filling techniques, the decomposition chart visualization was used as the basis of the EvoChart.

Table 6.2: Reviewed visualizations

Visualization	Information	Crowded	Occlusion	Tech [2]
decomposition	5	n	n	CLTS
flowchartABS	4	y	n	CLT
flowchartPER	4	y	n	CLTS
barchart	3	y	n	CLT
linechart	3	y	y	C

6.6 EvoChart

After selecting the decomposition technique for the main visualization, a few more features were added:

[2] C = Color Mapping, L = Light Mapping, T = Texture Mapping, S = Space Filling

Coloration Range: The values representing the lowest and the highest range are selectable, giving the evaluator the option to use his own judgment on how good or bad a value actually is.

Zooming: Using the mouse, the image can be zoomed to get a better overview in case the visualization becomes too large to view

Dragging: Using the mouse, the image can be dragged around for cases when the information is too big to fit on the screen and the user prefers not to zoom

Selectable axis: The presented data has 4 columns: programmer or goal, location, revision and value. The first three columns can be used as either the x or y axis.

6.7 The final EvoChart implementation

The EvoChart was created as a Java visualization. It was written as a Meta-Environment visualization plugin and it made use of the already existing Meta-Environment visualization framework. This framework allows the visualization to be plugged in to the Meta-Environment without having to modify any existing Meta-Environment Toolbus scripts.

The visualization code receives an RStore Fact from the Meta-Environment, which it then translates into a Prefuse Table. By examining the information in the Fact and based on the user's preference, the axial information is obtained from the first three columns of the data. These columns include either the programmer or goal, the location and the revision number. Three different views can be obtained from this:

Goal or programmer oriented view: Depending on how the data has been transformed, the information of one goal or developer is observed. The location and the revision number are used as axis. The view shows either how well all of the programmers adhered to the selected goal or how well a selected programmer adhered to all the goals.

Location oriented view: The information of one location is observed. The goal or programmer and the revision number are used as axis. The goal either shows how the programmers changed the adherence to the selected project goal, or how the adherence to the project goals were changed by the selected programmer.

Revision oriented view: The information of one revision is observed. It either shows how the programmer changed the adherence to the project goals for the locations he worked on, or it shows how the adherence of a selected goal was changed for the locations by the programmer.

The visualization also has the option to show the textual value that the color is representing. The EvoChart uses a palette with a set amount of colors. It uses the greenest color for the lowest value and the reddest color for the highest value. Because the values can also be represented as text, a color blind person will still be able to use the visualization.

Axial information can be swapped. A user might prefer to scroll down instead of scrolling to the right. By swapping the axis, the user can decide how he wants to scroll through the visualization. The swapping is done by re-transforming the RStore Fact and recreating the visualization.

The EvoChart can also be zoomed in and out. This was done by using Prefuse's user interface plugins. The zooming allows the user to see an overview even if there is a lot of information that is being displayed.

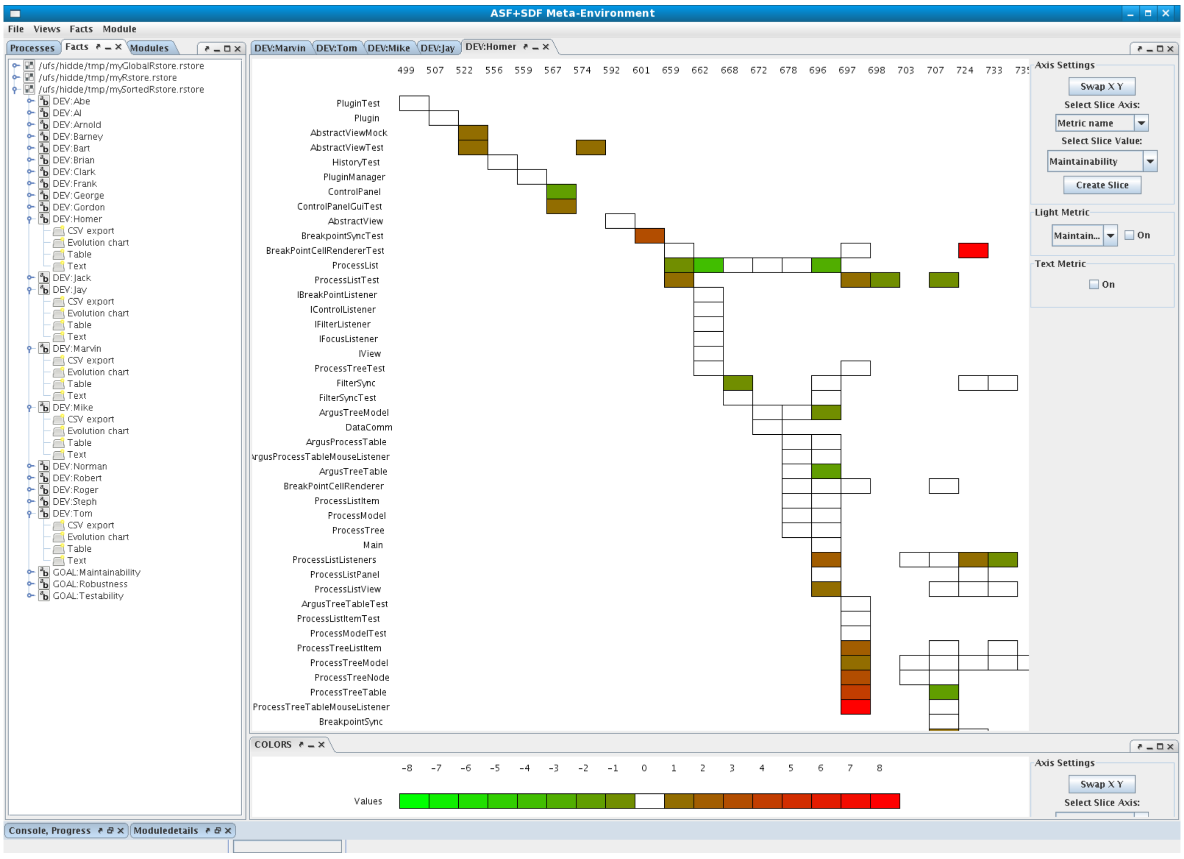


Figure 6.2: The EvoChart visualization, integrated within the Meta-Environment. It is showing how well one of the programmers adhered to the Maintainability goal, as set by the project's criteria.

Chapter 7

Validation

This chapter contains the validation of the EvoChart visualization. Using the Argus-viewer case study, the necessary data was obtained to create an EvoChart visualization. The visualization depicts the performance of the developers. To validate the EvoChart, a comparison was made with both a line and barchart.

7.1 Evaluating the developers

With the metrics determined, the next step in the validation was to see how the developers lived up to the project goals. By checking out the program states of every Argus-Viewer, extracting the metric values and linking them to the programmers the EvoChart was able to visualize the programmer performance.

Most of the selected metrics are integer based. They count the number of occurrences of things that are against the project goals. This makes it possible to do two kinds of evaluation: A global evaluation where for every project goal all the metrics are combined into a single number, and a more detailed evaluation where the metrics remain single entities. The global evaluation will be used to detect the outliers of the developers and discover which of the developers are having a bad impact on the project goals. A teacher could even use this information to make a judgment on the grade a student programmer should get.

After reviewing the global data a more detailed view will then be used to pinpoint which specific goal was being badly influenced and where. Using this information, the team leader could then extract what the developer did and help the programmer to improve on his skills. Since the project in the case study is finished this step was omitted during the EvoChart evaluation.

The detailed information was gathered using the aforementioned metric extraction tool, but for the creation of the global overview data, a database application was used. This was done for simplicities sake: normal database applications are very advanced in the way that they can group and transform tables of detailed information into a more global table, using functions such as averaging, summing and rounding the grouped values.

7.2 Creating the global overview data

Since there is no indication on the level of importance of the individual goals, all goals were valued as equally important. The number of maintenance related metrics do seem to be almost twice as big as the number of testability and robustness metrics. It could therefor weigh more heavily in the total overall score, but since there is no indication of which goal is deemed the most important, there is no need to compensate. When certain goals are deemed more important than others, weights can be added to the test results of those goals, increasing the effect that they have on the overall global score.

The data that is extracted contains the number of errors or violations of the project goals that they made, the location of these errors and the revision in which the violation took place. The global overview was constructed by averaging the number of faults made over all the files changed during a certain revision. Because the EvoChart visualization is integer based, this number was multiplied by three and rounded afterwards, creating a bigger range for the visualization to display.

7.3 Choosing the benchmark visualizations

Chapter 6.5 describes how the following visualization techniques were chosen as candidates for the EvoChart tool. The goal of this chapter was to find the best possible candidates. The goal of this chapter is to select the best one using a benchmarking test, by simulating the real world usage. The scenario will both test the overview capabilities as well as the detailed information capabilities of the visualization.

The following five visualization techniques were chosen in chapter 6.5:

- decomposition
- flowchartABS
- flowchartPER
- barchart
- linechart

Of the five visualizations, the decomposition chart had to be made since no suitable applications were found that could produce such a chart. The flowcharts were also hard to find. The closest resemblance to these visualizations were the areacharts that can be created in Microsoft Access. The problem with the areacharts is that they only looked like what the proposed flowchart should look like, but it seemed to create visualizations that were so unreadable that it would not make a fair evaluation. The overview picture in Figure 7.1 and the detailed picture in Figure 7.2 point out the areacharts difficulties.

There wasn't enough time to create proper flowchart visualizations that would prove that they would be harder to use than the EvoChart evaluation. Instead, a theoretical study was done to see how they would have compared to the EvoChart visualization. The line and barchart visualization were much easier to locate, and were therefor used in the visualization comparison test of chapter 7.4.

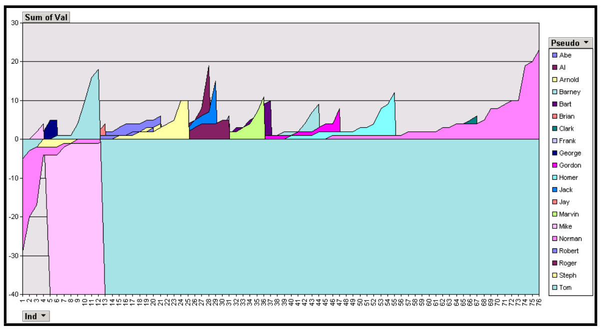


Figure 7.1: The Areachart of the overview information. The image is hard to decipher. It seems that the different areas are overlay-ed, making it hard to see the underlying areas. The overlays create occlusion which the flowchart doesn't have.

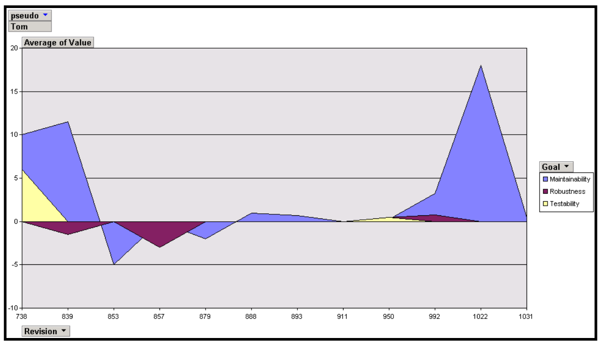


Figure 7.2: The Areachart of the detailed information. The same overlaying technique can be seen here.

7.3.1 Flowchart with absolute values

The flowchart with absolute values looks like a fan, starting small in the beginning and ending bigger as more files and code is added. This would create a poor overview. Let's assume that there is a visualization that an image as is shown in Figure 7.3.

As can be seen in the image, the beginning of the project can be properly visualized by zooming into the left side but this would leave no space to draw the rest of the project. The end of the project can be properly visualized by zooming out but this would make the beginning unreadable since it would become too small. There is no way to create an overview that can display both the beginning and the end of the project, unless the project is really small.

Using this technique to draw detailed information would have the same problem. The more work someone has done, the harder it will be to create a good image that shows all the data at once.

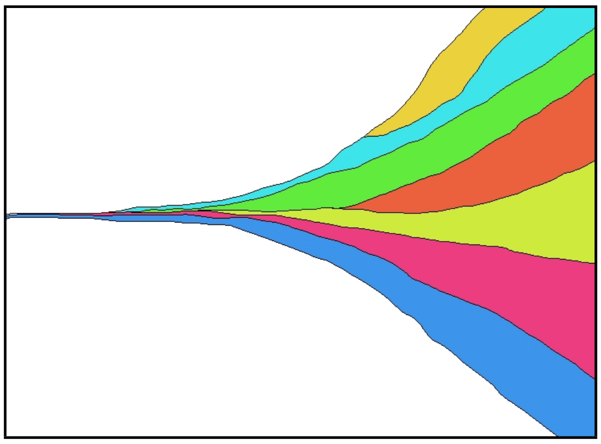


Figure 7.3: A flowchart with absolute values, creating an overview image of an imaginary project. The x axis is displaying the time information. The programmers are displayed by using different colors for each of them. Their impact on the project goal is displayed by the size of the colored area.

7.3.2 Flowchart with percentage based values

The flowchart with relative values looks like a bar, staying the same height throughout the entire project. Where the flowchart with absolute values creates a crowded visualization at the beginning of the project, the flowchart with percentage based values would do the opposite. The more programmers worked and the more code is written, the more cluttered the image would become. The overview image can not be properly constructed by this technique.

7.3.3 Summary

The flowcharts were proposed as EvoChart visualization techniques, but were deemed to be less effective. Unfortunately, no proper implementations were found of the flowcharts. Because the time limit didn't allow the creation of these charts, a theoretical study was performed to see if they could better the EvoChart visualization.

Both charts have a problem with creating overview and detailed information for big projects, which would make them hard to use during the comparison test, so the EvoChart seems to be a better candidate. The line and barchart visualizations were much easier to find, so they were used in the real visualization comparison test in chapter 7.4.

7.4 Visualization Comparison

The created EvoChart was used to see if it was capable of evaluating developers and how it would compare to the visualizations that were not chosen. For this last validation step, Microsoft Access was used to create the images.

Note 1: The data used in this comparison has been anonimized. Even though the data of the system itself is as is, the names of the developers were altered.

Note 2: The number of commits doesn't accurately reflect the amount of work a programmer did. The project made use of pair programming. In pair programming, one programmer works while the other reads and helps. The programmer that works commits the joined effort under his own name. While it is customary to swap seats during pair programming, so each developer would receive the same amount of commits, it can not be guaranteed that this was practiced meticulously. There were also a few programmers who worked by themselves, which could potentially double the amount of commits made while doing the same amount of work as a pair of programmers who each share the commit load.

7.4.1 Evaluation with EvoChart

The global overview data was then transformed into the RStore format. The legend in Figure 7.4 shows how the EvoChart was configured. The range was selected between -8 and +8 because most of the values were within this range. In some occasions a programmer made a great deal more mistakes but these were only incidental and they were considered insignificant to increase the coloration range, because of their incidental nature and because it would make the value distinction harder for the more common values.

The values display the amount of violations that a programmer created. A negative number means that the programmer actually removed violations, while a positive number means that the programmer added violations. The scale goes from <-8 (brightest green) to >+8 (brightest red).

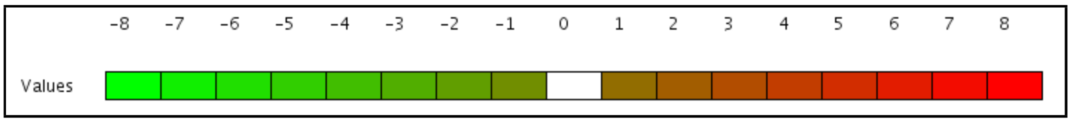


Figure 7.4: This image displays the configuration of the color scale used by the EvoChart visualization. -8 and lower values were colored in the most brightest green color, +8 and higher values were colored in the brightest red color.

The first step of the evaluation was establishing if the image was capable of creating an oversight of the performance of all of the programmers. When comparing programmers to each other it is necessary to look at the amount of commits they have made and to determine how many of those commits led to good or bad results. For this reason, the commits were ordered from good to bad. The positive effect the programmer had on the project goals are depicted by green squares on the left hand side. White squares in the middle are the commits the programmer made that didn't affect the project goals in either a good or bad way. Red squares on the right depict the bad impact the programmer had on the project goals.

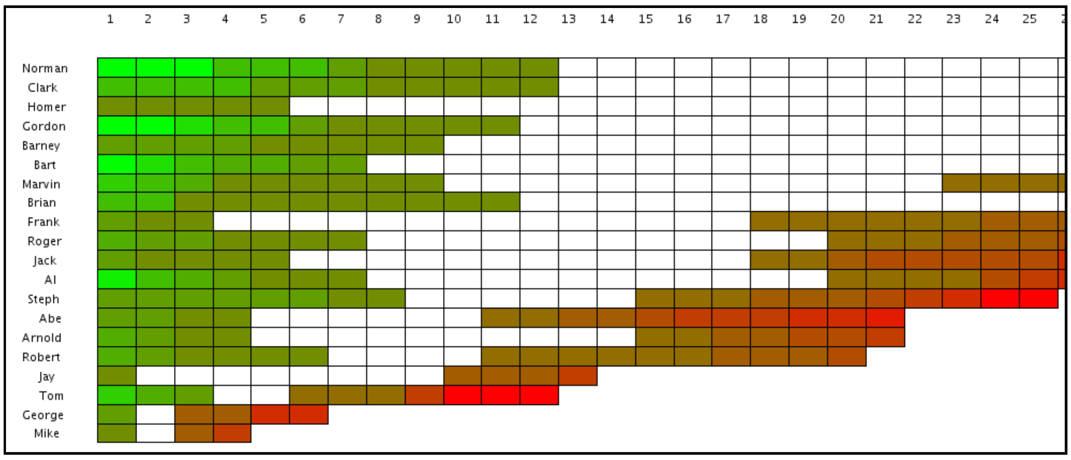


Figure 7.5: EvoChart depicting the global evaluation data of all the developers

Figure 7.5 shows the overview that is created by the EvoChart. The X-axis is in this case not a revision axis, it simply counts the number of squares drawn. Since every square also represents a commit, it could be seen as a commit counter. The commits are not in a chronological order though, since they are sorted by metric value. The Y-axis was sorted by the amount of commits that a developer made.

The chart clearly shows who made the most commits and who made the least. The programmer with the most commits had a total of 79, while the programmer with the least amount had a total of 4. Figure 7.5 only shows 25, as to keep the names and numbers legible on paper.

Mike, the person with the lowest amount of commits, spent most of his work as a pair programmer. Because he forgot that he should commit at least half of the amount of work, his commit count is very low. George who comes right behind him in number of commits actually stopped working on the project somewhere midway. These two programmers were therefor not further examined during the detailed review.

The overview clearly shows that most programmers have a few positive commits, some more bad and even more neutral (white box, score of 0) commits. Since a neutral commit actually means that they are adhering to the project goals, these could be seem as good commits as well. On average, most programmers performed well, hardly making a serious impact on the project goals at all.

The one that really stands out is Tom. Tom has more bad than good commits, even with the neutral commits taken into account, and a red spike can be seen as well. Tom is clearly not doing as well as the others. The further review will take a look into why Tom is performing worse.

Tom's data, split into the separate goals.

Looking at Tom's data, it is clear where the problem lies. Most of the goal violations apply to Maintenance. While his Robustness and Testing charts are mostly white, his Maintenance chart is showing mostly reds. There is an odd occurrence that can be seen in his work on the ToolPerformanceInfo that isn't consistent: it turns from red to green. It is also a class where he made more than a few commits on. To see if the EvoChart can find a reason, this class was further examined.

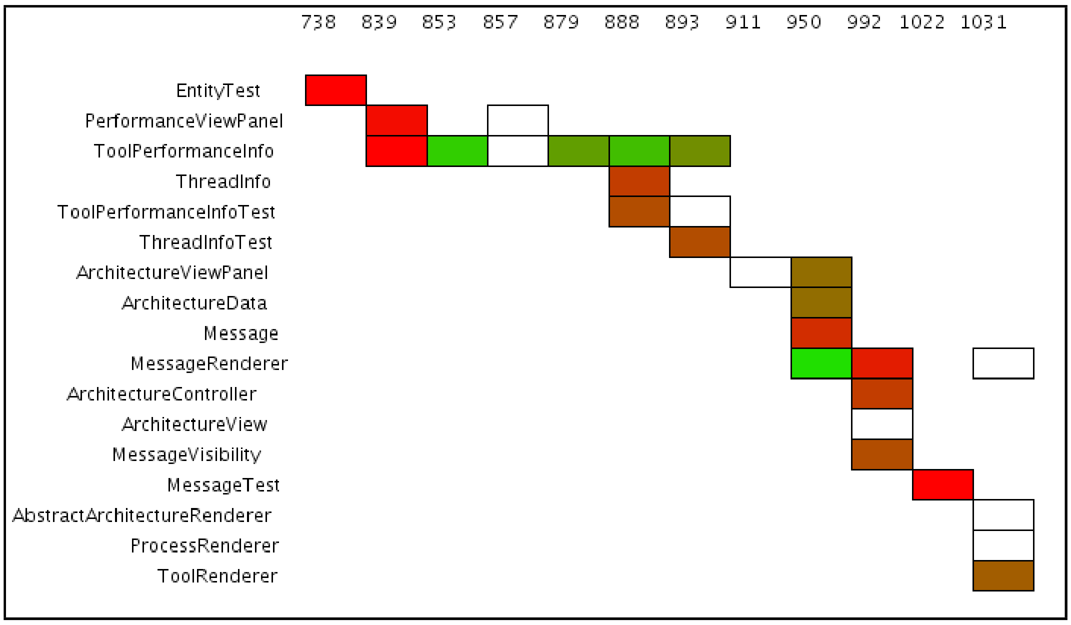


Figure 7.6: Tom's Maintenance adherence

ToolPerformanceInfo

Tom, who started on the ToolPerformanceInfo class, started by violating a lot of the project maintenance rules. Figure 7.7 shows that soon after, Frank started working with him. It appears that they worked in a team, and since that moment on Tom improved. No longer violating the rules, he cleaned up after himself, most likely inspired by Frank. It seems that when Tom is paired up with Frank, he is a better programmer, more concerned about writing maintainable code than he was when working alone.

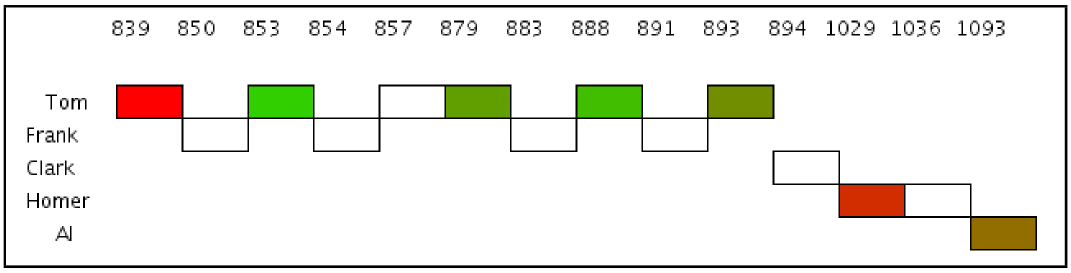


Figure 7.7: The ToolPerformanceInfo class where Tom is apparently pair programming with Frank

Summary

The EvoChart worked in both visualizing an overview of all the programmers as well as give an insight into the programming details. The overview shows that Tom is making more project goal violations than he should, and the detailed information shows that this can be prevented by pairing him up with another programmer who inspires him to do better.

7.4.2 Evaluation with linechart

The linechart visualization is capable of creating an overview picture, just like the EvoChart, but there is a big downside: only outlying values are really clear. Because the lines are intersecting each other it is hard to clearly distinguish one programmer from another. Figure 7.8 shows on overview of all the developers and all the work they have done.

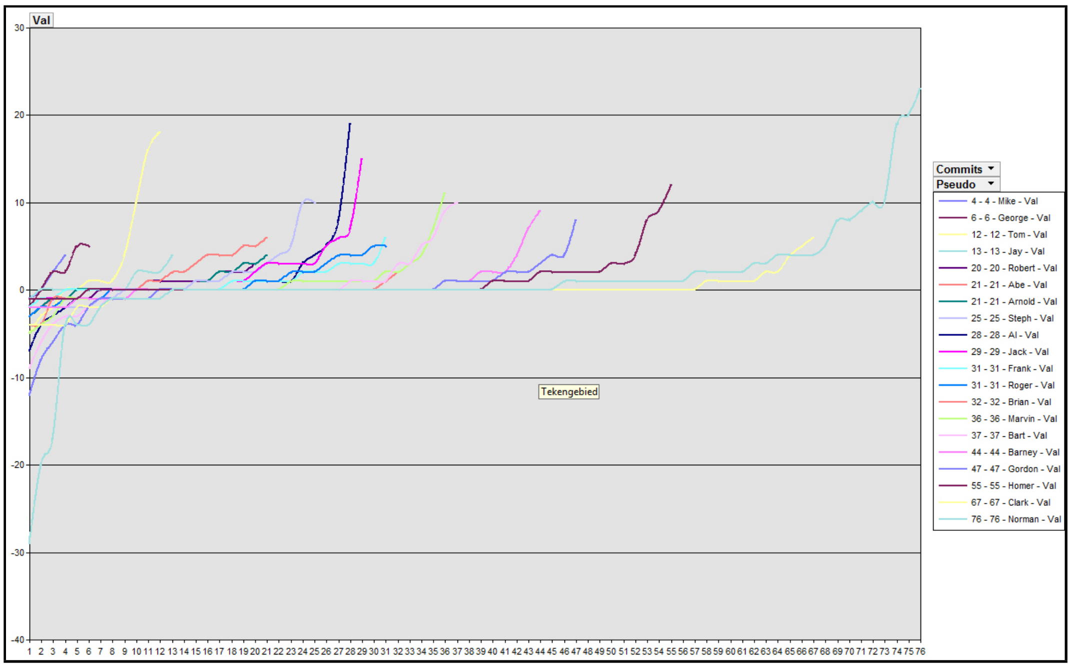


Figure 7.8: Linechart overview of all the data

To make the comparison a bit more fair, I also extracted an overview containing only the best 25 commits, just as is shown by the EvoChart Figure 7.5. Because of the coloration of the lines it is hard to see all the outlying values. The other real downside is that there are only so many colors that a person can really recognize and differentiate between. Because of the amount of programmers, some of them ended up with the same base color. To make things worse, it is hard to differentiate between red and orange and the color yellow isn't even noticeable. The greater the number of developers there are, the harder it becomes to review them all at once because of this.

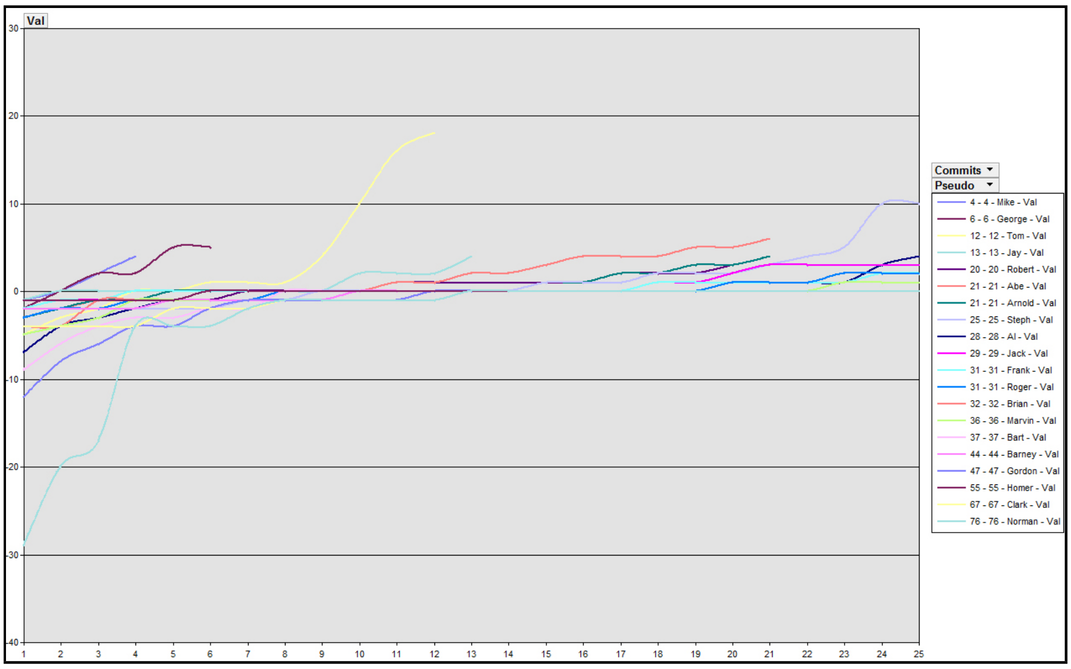


Figure 7.9: Linechart overview of the best 25 commits of each developer

Tom's data, displayed on a linechart

The linechart visualization of Tom's work does a poor job of accurately reflecting it. Since a line chart needs two points at least to create a line, it is inaccurate in cases where Tom only created a single commit. In fact, these lines do not even appear on the visualization at all. The data would have to be manipulated in a certain way to give the visualization a chance to draw these lines. Still, the same trend can be seen here: The ToolPerformanceInfo is an inconsistency: where Tom first created a bad implementation, he later starts to clear up his mistakes.

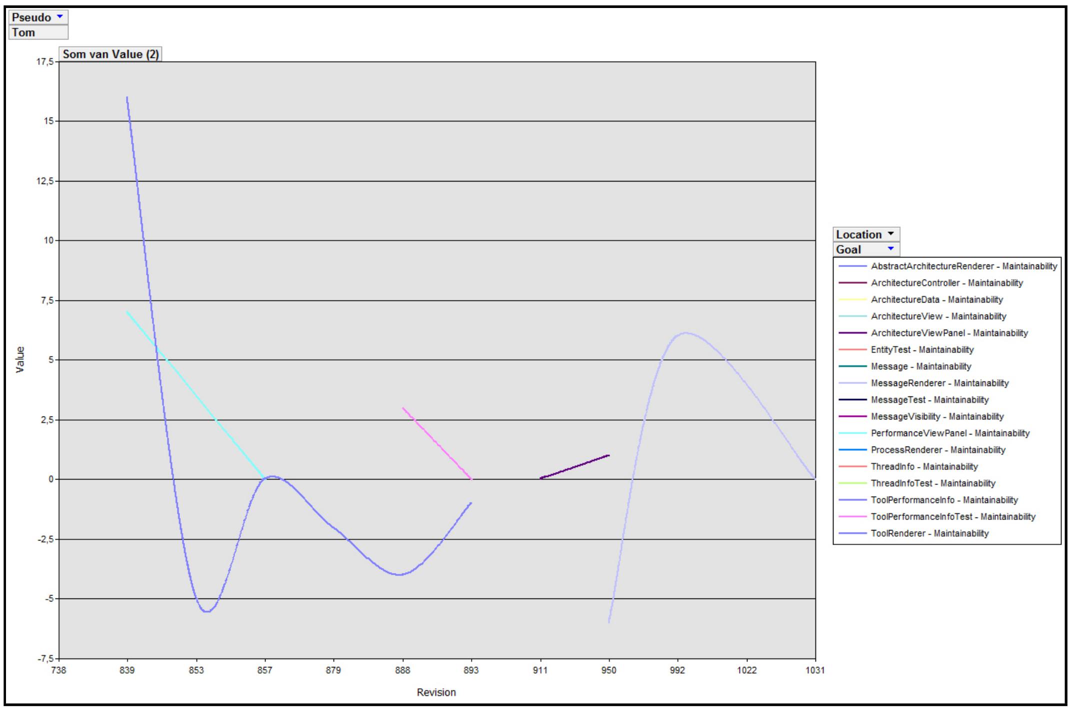


Figure 7.10: Tom's Maintenance linechart

ToolPerformanceInfo, displayed on a linechart

While the linechart still shows how Tom and Frank worked together and how Tom's performance drastically improved during this time, it isn't showing an accurate image of what happened. The line chart suggests that both Tom and Frank committed code at the same time. Since Tom and Frank made commits one after another (It is not possible to make a single commit with 2 different SVN accounts at the same time) the linechart is giving an inaccurate picture of what actually happened.

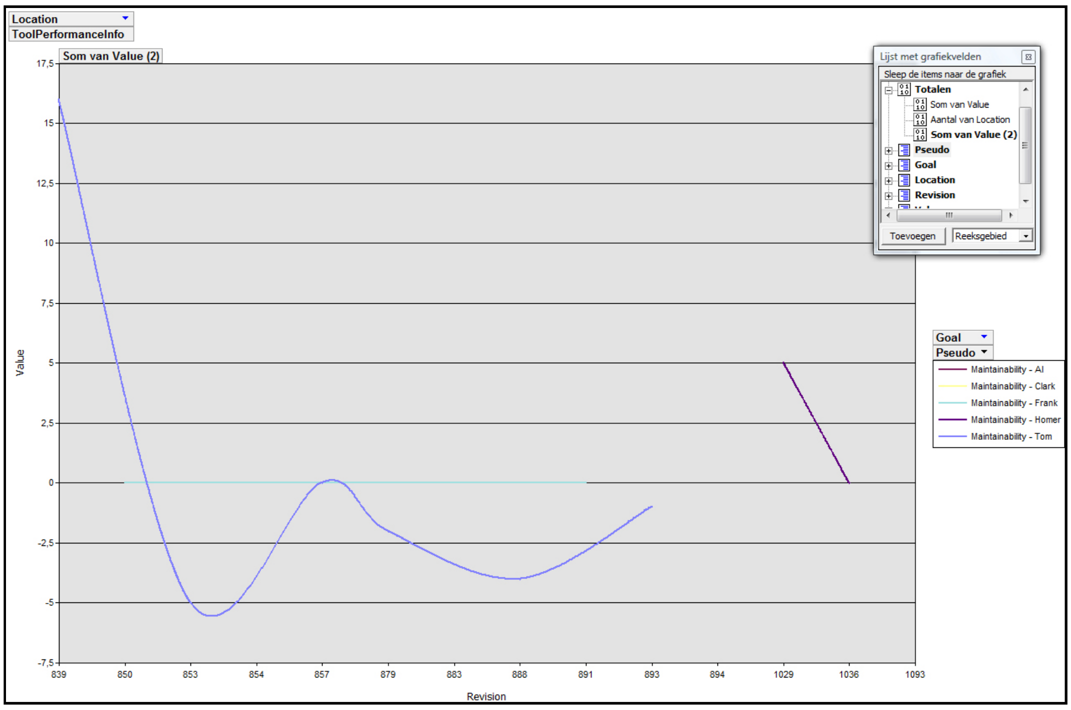


Figure 7.11: ToolPerformanceInfo linechart

Other linecharts

The linechart images in this section have some downsides, like inaccurately showing the details and the crowded overview image, which may be overcome by using additional techniques. Each programmer could have his own section, effectively filtering out the others, and by flipping through the programmers a mental image could be formed to evaluate them. Perhaps multiple charts could be created, using only a part of the programmers, to try to prevent the occlusion that the linechart is typical for. Perhaps the drawing of the detailed information could be improved, by severing lines when there is a gap in the commits.

But those techniques would not create a linechart that could equal the decomposition technique that the EvoChart uses. No techniques could be found that would create a better overview or a better detailed image. Therefor, the basic linechart that Microsoft Access produces was used in this comparison.

Summary

While the linechart can be used as a visualization technique for the same data, it isn't as good as the EvoChart. The overview contains lots of intersecting lines, making it hard to distinguish different developers and even harder to get a total picture. The lack of usable colors is also a concern, since the more lines that have to be drawn, the sooner they start looking like each other.

The linechart is also inaccurately displaying the detailed information. Single commits seem to be lost (since a linechart needs at least 2 points to be able to draw a line) so a developer would have to make at least two commits of a file before it is drawn. The linechart also draws the lines in such a way to suggest that programmers commit together, while in fact this is impossible since SVN only allows one single programmer to commit at a time.

7.4.3 Evaluation with barchart

The barchart visualization is very bad at creating a good overview. This can clearly be seen in Figure 7.12 where every bar looks just as black as every other.

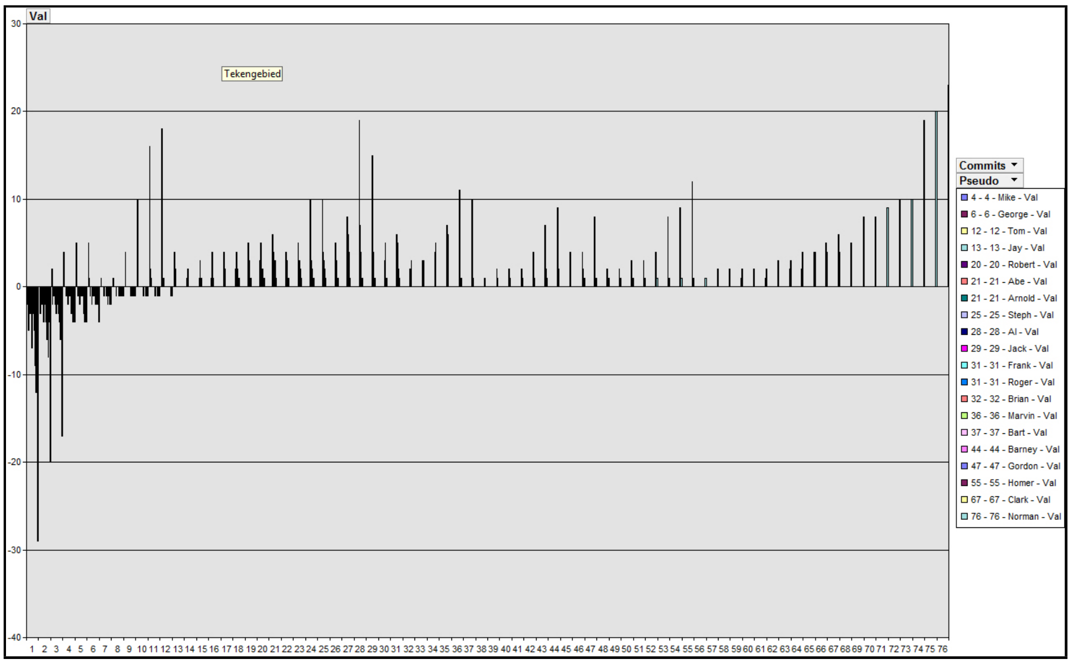


Figure 7.12: Barchart overview of all commits

Even when showing only the 25 best commits the bars are still very hard to discern. With a lot of eye straining it is possible to see which bar represents which developer. This fact alone makes the barchart unusable for creating a good overview. Evaluation would have to be done by only showing one programmer at a time, to make the picture properly readable, essentially creating a situation without an actual overview.

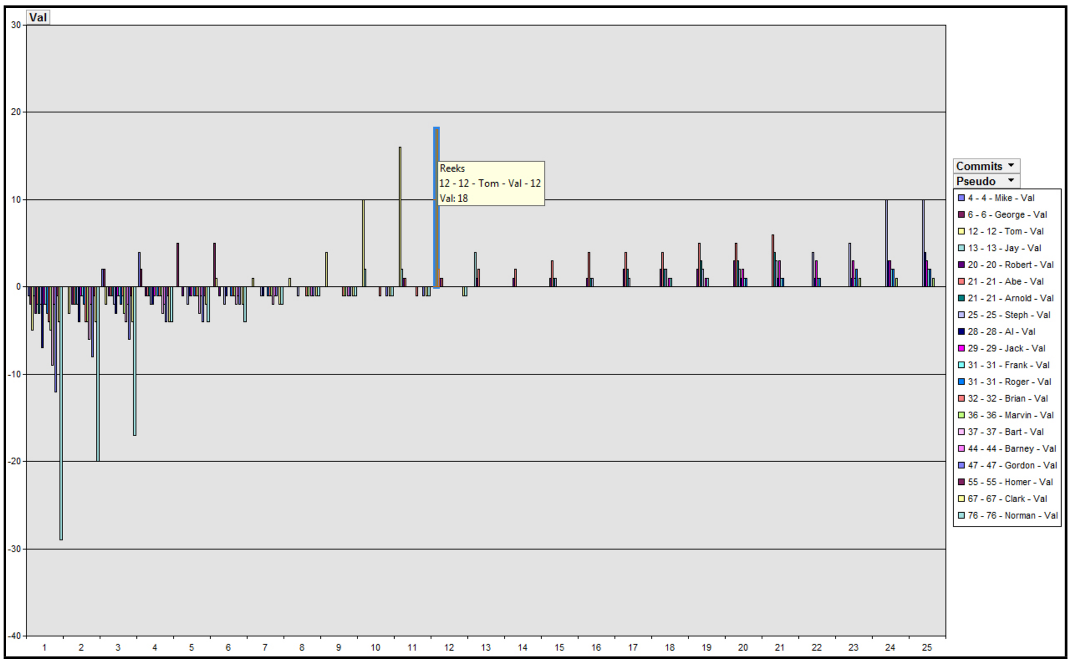


Figure 7.13: Barchart overview of the 25 best commits

Tom's data, displayed on a barchart

The detailed information is a little better. Because there is less data to display, seeing the work that Tom has done is easier, but still takes a while to process. This could perhaps be because of a bad grouping decision on my part. But the main concern is that the barchart is also not showing the "no impact" commits, the commits where no violations were made or fixed.

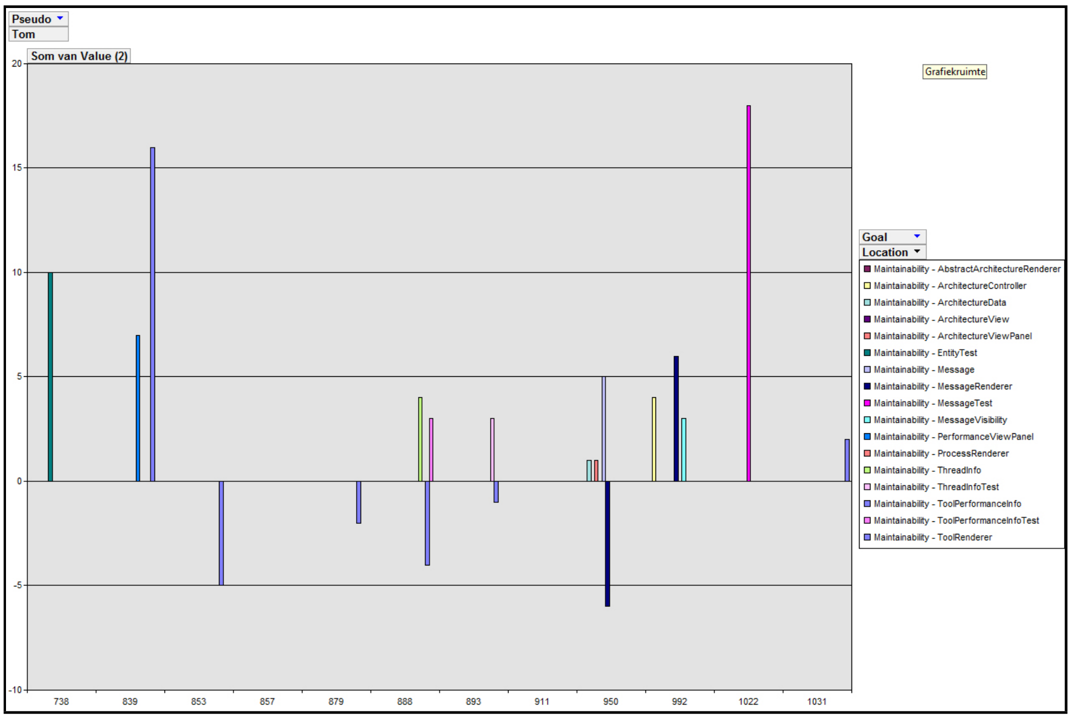


Figure 7.14: Barchart of Tom's data

Using a 3-dimensional version of the barchart this issue can be solved, but it has a drawback that can be seen in Figure 7.15. When small bars are drawn directly after bigger bars, they become obscured and hard to detect. An example of this can be seen in revision 992 where the big yellow bar is obscuring the almost non existent purple bar.

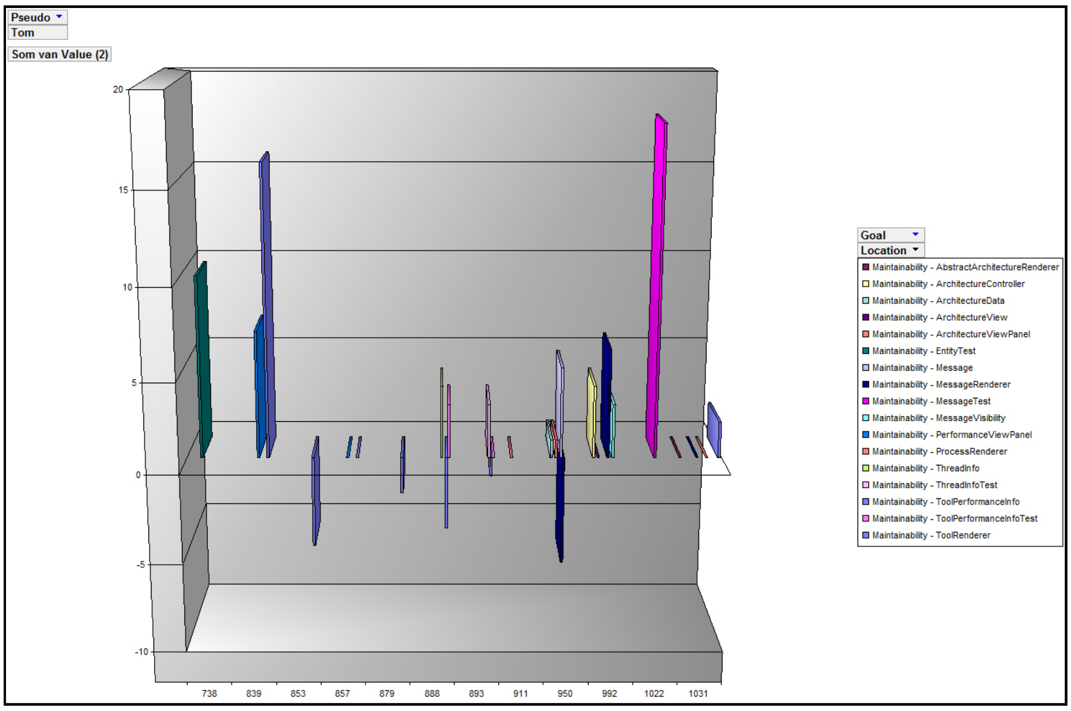


Figure 7.15: 3D Barchart of Tom's data

ToolPerformanceInfo, displayed on a barchart

When looking at the ToolPerformanceInfo it is clearly not showing all the data again. As can be seen in Figure 7.16 it isn't clear at all why Tom is suddenly working better and big gaps can be seen between the bars, indicating the spaces where the missing bars should be.

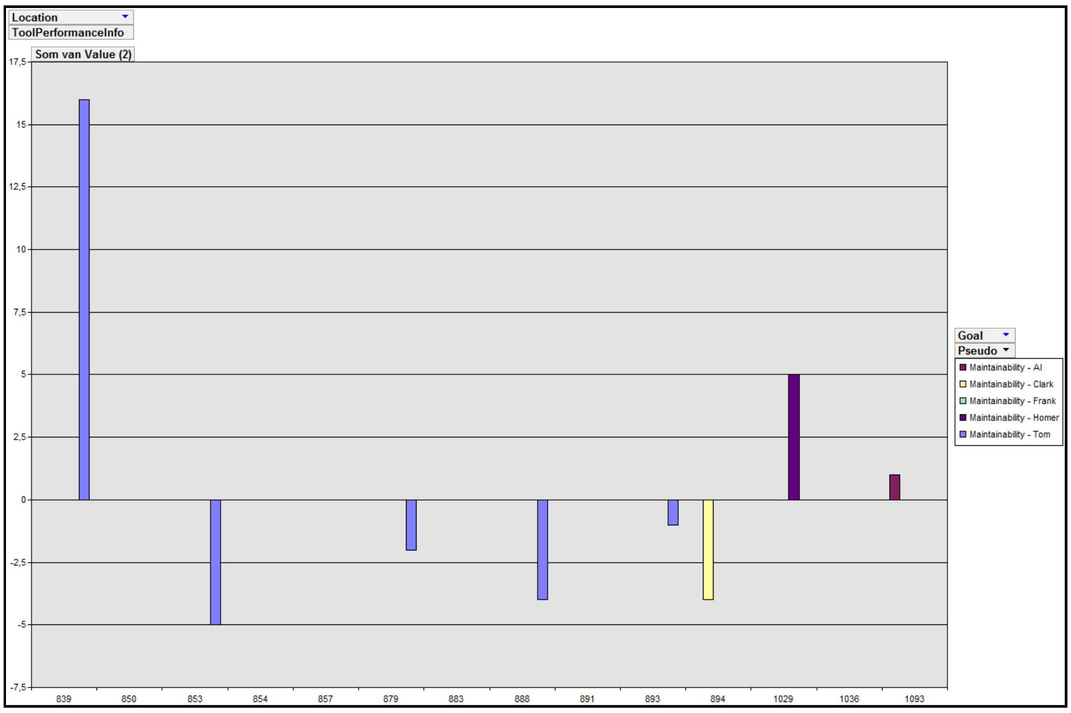


Figure 7.16: Barchart of the ToolPerformanceInfo data

The 3-dimensional version does a better job, since it is showing all the data. The threat of occlusion still remains. Fortunately, in this image, no occlusion can be seen. The effect has however already been shown in Figure 7.15.

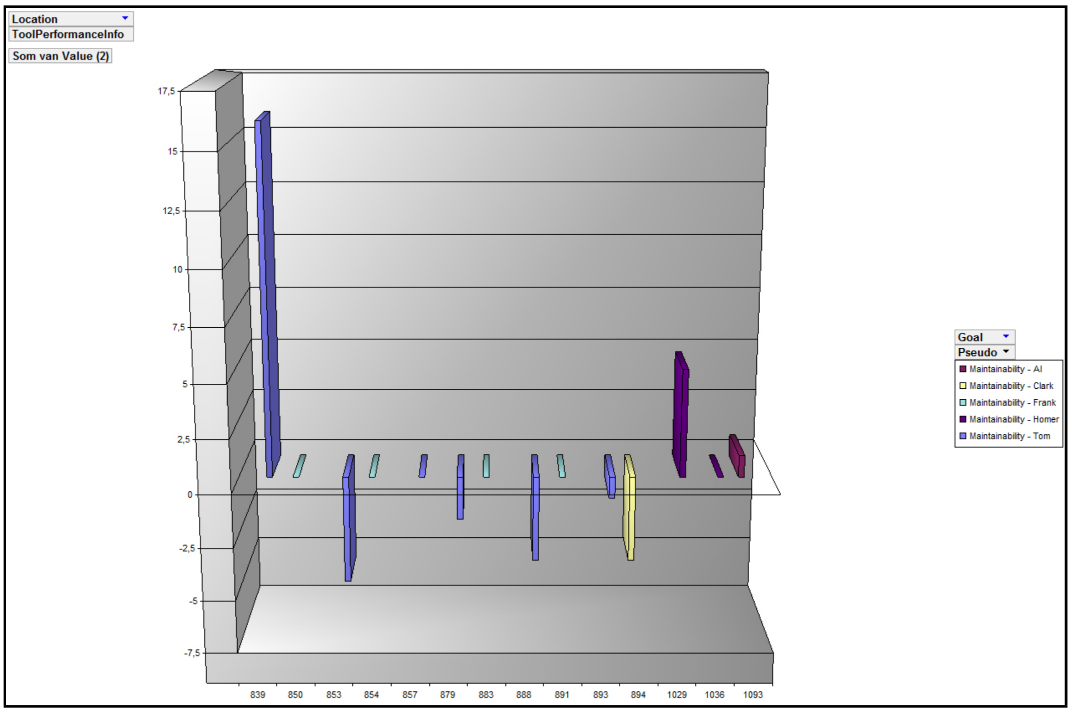


Figure 7.17: 3D Barchart of the ToolPerformanceInfo data

Other barcharts

The barchart has a real problem when it comes to showing lots of information. By cleverly grouping the data, better images could be obtained. Grouping the bars by programmer instead of by commit number, it suddenly becomes easier to keep the programmers apart. But because of the amount of data, it would still be hard to see which section belonged to which programmer, since most of the bars would be rendered black.

Bars could even be stacked on each other, creating an image that would look like Figure 7.18. The stacking technique does help in creating a clearer image, but understanding the data suddenly becomes a lot harder as can be seen in the image. Since the data is now aggregated, it needs to be separated again in the mind of the viewer, making it a difficult task to properly evaluate the programmers.

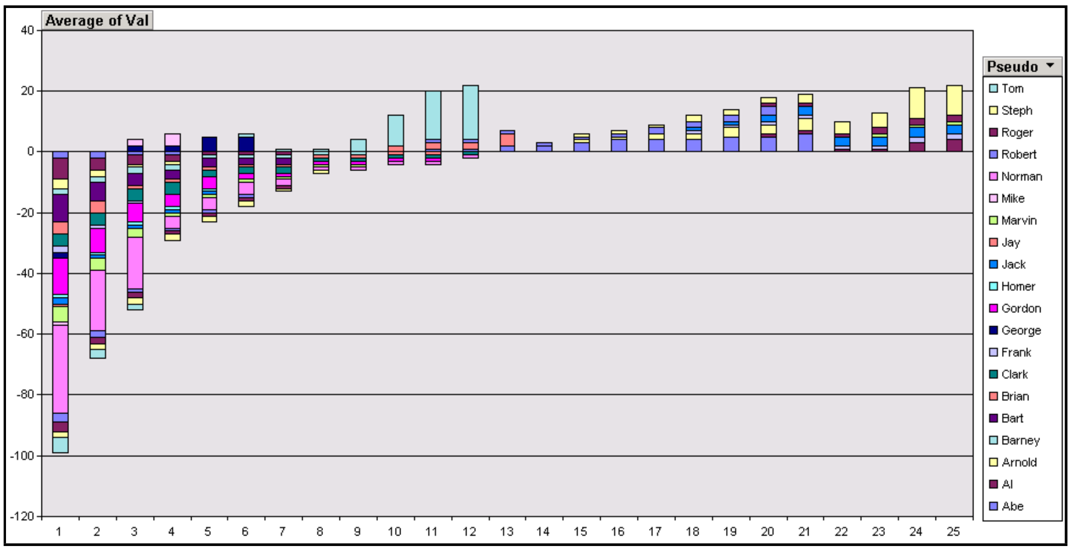


Figure 7.18: A barchart where the bars are stacked on each other.

Because no other techniques could be found that would make a better barchart, the standard Microsoft Access barchart was used.

Summary

The barchart is not a good visualization technique to use for this kind of task and this kind of data. The barchart is not good at creating large overviews and like the linechart, it has the tendency to inaccurately display the detailed information.

Chapter 8

Conclusion

The EvoChart is a good visualization technique for evaluating programmers. It can create a good overview showing the work that all the programmers have done, making it possible to compare programmers to each other. In an academic setting this allows an evaluator to pick the best and the worst programmers and grade them accordingly.

The detailed information will help a team leader to locate the areas in which a programmer is performing badly, allowing him to deal with this situation by either coaching the programmer to work better or by assigning the programmer to less important tasks. The visualization can even detect which developers are working together, as shown in 7.7. This knowledge can help in assigning pair programming teams: the visualization shows that working with Frank really inspires Tom to work better than he normally does, ergo, teaming Tom and Frank together would benefit Tom's performance.

The linechart and the barchart have the ability to display the same data, but are unable to create clear overviews. The linechart makes the overview too cluttered, while the barchart makes the overview unusable by creating lots of unrecognizable bars.

There are a few notes that should be taken into account, when using the EvoChart for evaluation purposes. When a project is done using the pair programming technique, the best results are obtained when each paired programmer commits roughly the same amount as his partner. Otherwise, one programmer would seem to be more productive than the other, while both are doing the same amount of work.

The amount of commits shouldn't be used as a real indicator into how productive a programmer is in any case, since some programmers will commit after every small change and others only commit once a day. However, a very low amount of commits can be an indicator that the programmer isn't working as hard as he should and this could then be investigated. The best way to evaluate is by checking the amount of good commits versus the amount of bad commits.

Some might view that pair programming would make the evaluation unfair, since a programmer is not only committing his work but that of his coworker as well. However, the committer is always responsible for the code he sends in, even if he is teamed up with someone else. And a real pair programming pair has the committer doing all the typing, while the partner only points out improvements and oversights, which would only improve the coding quality if done right.

When the EvoChart is used for more than simply grading, it creates a better oversight of who does what than the standard bar and linechart visualizations. It was regrettable that a good flowchart program wasn't available for testing, but as is, the EvoChart is still a good visualization tool for evaluating programmers, be it for grading or for improving their performance.

Bibliography

[CS02] Ben Collins-Sussman, *The subversion project: buiding a better cvs*, Linux J. **2002** (2002), no. 94, 3.

[CSM79] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman, *Third time charm: Stronger prediction of programmer performance by software complexity metrics*, ICSE '79: Proceedings of the 4th international conference on Software engineering (Piscataway, NJ, USA), IEEE Press, 1979, pp. 356–360.

[Dia] Wrong Diagnosis, *Prevalence and incidence of color blindness*, http://www.wrongdiagnosis.com/c/color_blindness/prevalence.htm.

[Die07] Stephan Diehl, *Software visualization - visualizing the structure, behaviour, and evolution of software*, Springer, 2007.

[ESEES92] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner, *Seesoft-a tool for visualizing line oriented software statistics*, IEEE Trans. Softw. Eng. **18** (1992), no. 11, 957–968.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger, *Software metrics: A rigorous & practical approach*, 2 ed., International Thomson Publishing, 1997.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse, *How developers drive software evolution*, Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), IEEE Computer Society Press, 2005, pp. 113–122.

[HCL05] Jeffrey Heer, Stuart K. Card, and James A. Landay, *prefuse: a toolkit for interactive information visualization*, CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems (New York, NY, USA), ACM, 2005, pp. 421–430.

[Hel07] Michael T. Helmick, *Interface-based programming assignments and automatic grading of java programs*, ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (New York, NY, USA), ACM, 2007, pp. 63–67.

[LB85] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*, Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Mye90] Brad A. Myers, *Taxonomies of visual programming and program visualization.*, Journal of Visual Languages and Computing **1** (1990), 97–123.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small, *A principled taxonomy of software visualization*, Journal of Visual Languages & Computing **4** (1993), 211–266.

[PMD] PMD, *http://pmd.sourceforge.net/.*

[RC92] Gruia-Catalin Roman and Kenneth C. Cox, *Program visualization: the art of mapping programs to pictures*, ICSE '92: Proceedings of the 14th international conference on Software engineering (New York, NY, USA), ACM, 1992, pp. 412–420.

[See06] Mauricio Seeberger, *How developers drive software evolution*, Master's thesis, University of Bern, January 2006.

[TM04] Melanie Tory and Torsten Möller, *Human factors in visualization research*, IEEE Trans. Vis. Comput. Graph. **10** (2004), no. 1, 72–84.

[vdBBE$^+$07] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. J. Vinju, *Using the meta-environment for maintenance and renovation*, CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (Washington, DC, USA), IEEE Computer Society, 2007, pp. 331–332.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk, *Cvsscan: visualization of code evolution*, SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization (New York, NY, USA), ACM, 2005, pp. 47–56.

[War00] Colin Ware, *Information visualization: perception for design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

Appendix A

Taxonomies

The domain of visualizations is complex by nature. Creating a visualization is a creative process and can lead to an endless amount of possibilities. It is therefor complex to capture the domain of visualizations and dividing it into separate categories. There are even several methods to create such a taxonomy, by looking at the problem from different perspectives. Below are a few lists containing both my own and several known taxonomies of the visualization domain.

A.1 Personal Taxonomies

Visualizations can be classified using several orthogonal fields. Below is a list of these fields and their subsets.

A.1.1 Time

Visualizations can be split into two timing categories: static or dynamic. A visualization is either a snapshot / unchanging image or it is a changing animated image.

- static
- dynamic

A.1.2 Data

Visualizations can be split up based on the data they represent.

- Linear data
- Relational data (A has a connection with B)
- Attribute data (A is a subcomponent of B)
- Combinations

A.1.3 Axiomatic Geometry

Axiomatic Geometry was introduced by Euclid, who introduced certain axioms/postulates, expressing the primary properties of points, lines and planes. The basic concepts lead to different categories that use these concepts. Each category has positive and/or negative aspects.

- Basic
  - Point
  - Line
  - Plane
  - Text (doesn't actually exist in axiomatic geometry but is added for personal understanding)
- Categories

  textbased: Good visualization of actual content and details. Bad for overviews.

  chart: Good for combining lots of statistics

  map: Good for creating an overview, bad hard to discern between elements

  nodes & links: Good for showing structures, containment, other relations

  landscape: Good for creating detailed overviews

  matrix: Good for data aggregation and discovery thereof

The categories can be combined to create a larger amount of categories. A few examples of these combinations are:

- Combinations
  - textbased-chart
  - node & links-chart
  - textbased-matrix
  - textbased-chart-matrix
  - landscape-node & links-matrix

A.2 Scientific Taxonomies

In the world of scientific visualizations, a number of researcher have made numerous taxonomies. These taxonomies are below and give an insight into the various perspectives when dividing the visualization domain into categories.

A.2.1 Diehl

When looking at the smaller field of scientific program visualizations, there are three distinct fields of research: [Die07]

Structure refers to the static parts and relations of the system, i.e. those which can be computed or inferred without running the program. This includes the program code and data structures, the static call graph, and the organization of the program into modules.

Behavior refers to the execution of the program with real and abstract data. The execution can be seen as a sequence of program states, where a program state contains both the current code and the data of the program. Depending on the programming language, the execution can be viewed on a higher level of abstraction as functions calling other functions, or objects communicating with other objects.

Evolution refers to the development process of the software system and, in particular, emphasizes the fact that program code is changed over time to extend the functionality of the system or simply to remove bugs.

A.2.2 Myers

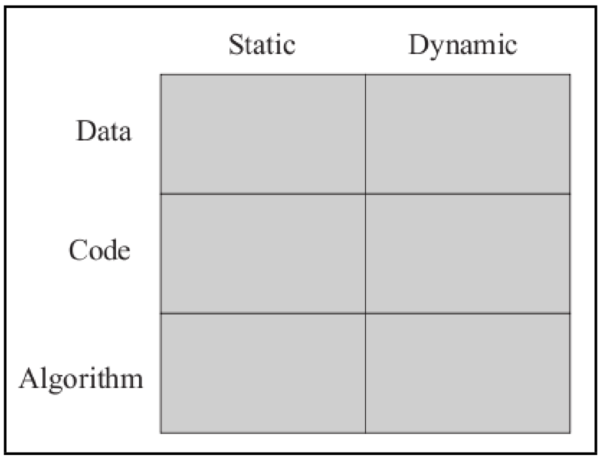


Figure A.1: The Myers Taxonomy. This taxonomy was obtained from [Die07] who used [Mye90] as his source

A.2.3 Price et al.

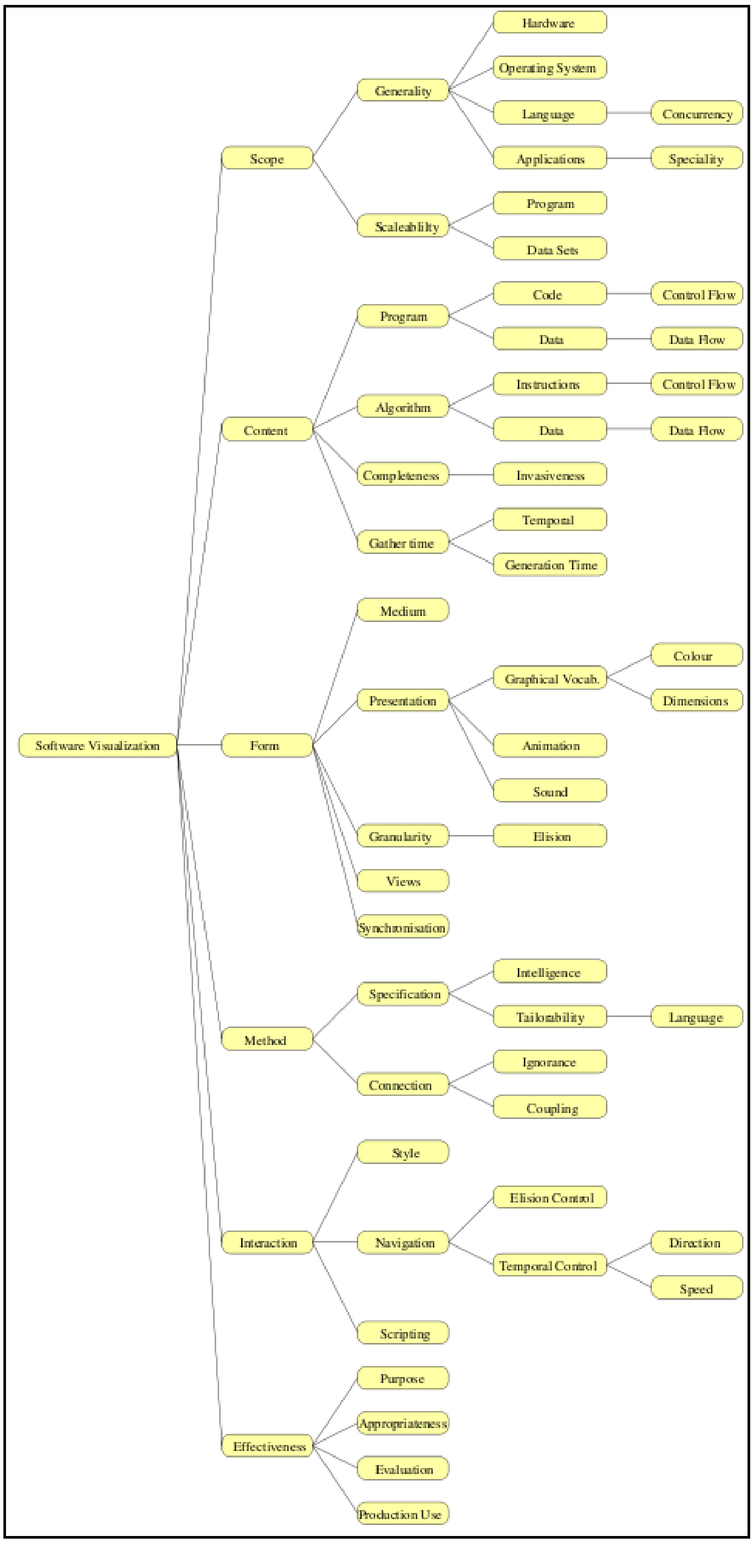


Figure A.2: The Price et al.[PBS93] Taxonomy of Software Visualization

A.2.4 Roman and Cox

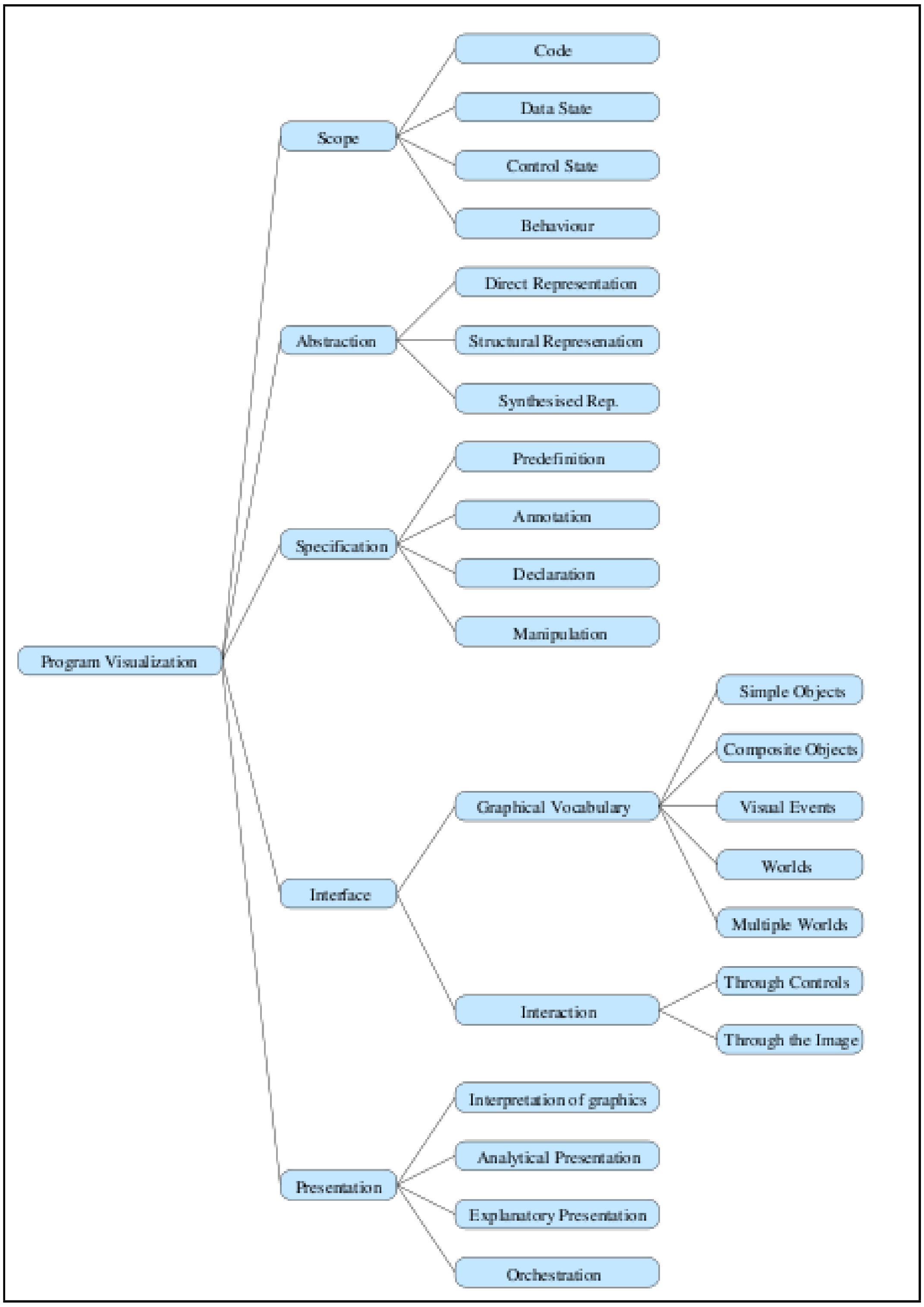


Figure A.3: The Roman and Cox [RC92] Taxonomy of Program Visualization

Appendix B

Chosen Metrics

During the validation process of the EvoChart visualization, I used the Argus-viewer project as the case study. The Argus-project criteria were used as the base in the metric selection process. The table below contains the chosen criteria.

Table B.1: Used Metrics

Goal	**Criteria**	**Metric**
Maintainability	cohesion	DefaultPackage
		TooManyStaticImports
	coupling	CouplingBetweenObjects
		ExcessiveImports
		LooseCoupling
	proper names	ShortVariable
		LongVariable
		ShortMethodName
		VariableNamingConventions
		MethodNamingConventions
		ClassNamingConventions
		AbstractNaming
		AvoidDollarSigns
		MethodWithSameNameAsEnclosingClass
		SuspiciousHashcodeMethodName
		SuspiciousConstantFieldName
		SuspiciousEqualsMethodName
		AvoidFieldNameMatchingTypeName
		AvoidFieldNameMatchingMethodName
		NoPackage
		PackageCase
		MisleadingVariableName
		BooleanGetMethodName

Table B.1: Used Metrics

Goal	**Criteria**	**Metric**
	duplication	AvoidDuplicateLiterals
		CPD (Copy Paste Detector)
		UseCollectionIsEmpty
	dead code	UnusedPrivateField
		UnusedLocalVariable
		UnusedPrivateMethod
		UnusedFormalParameter
	commentary	UncommentedEmptyMethod
		UncommentedEmptyConstructor
	succinctness	EmptyIfStmt
		EmptyWhileStmt
		ForLoopShouldBeWhileLoop
		UnnecessaryConversionTemporary
		EmptySynchronizedBlock
		UnnecessaryReturn
		EmptyStaticInitializer
		UnconditionalIfStatement
		EmptyStatementNotInLoop
		UselessOverridingMethod
		ExcessiveMethodLength
		ExcessiveClassLength
		ExcessivePublicCount
		TooManyFields
		TooManyMethods
		UnnecessaryConstructor
		SimplifyBooleanReturns
		SimplifyBooleanExpressions
		SimplifyConditional
		EmptyFinalizer
		DuplicateImports
		UnusedImports
		ImportFromSamePackage
	layout	IfStmtsMustUseBraces
		WhileLoopsMustUseBraces
		IfElseStmtsMustUseBraces
		ForLoopsMustUseBraces
Robustness	error handling	MisplacedNullCheck
		UnusedNullCheckInEquals
		BrokenNullCheck

Table B.1: Used Metrics

Goal	**Criteria**	**Metric**
		CheckResultSet PositionLiteralsFirstInComparisons ReturnEmptyArrayRatherThanNull SystemPrintln
	exception handling	EmptyCatchBlock EmptyTryBlock EmptyFinallyBlock ReturnFromFinallyBlock ClassCastExceptionWithToArray CloneThrowsCloneNotSupportException AvoidCatchingThrowable SignatureDeclareThrowsException ExceptionAsFlowControl AvoidCatchingNPE AvoidThrowingRawExceptionTypes AvoidThrowingNullPointerException AvoidRethrowingException DoNotExtendJavaLangError DoNotThrowExceptionInFinally
Testability	testability	JUnitStaticSuite JUnitSpelling
	test coverage	TestClassWithoutTestCases
	use of asserts	JUnitAssertionsShouldIncludeMessage JUnitTestsShouldIncludeAssert UnnecessaryBooleanAssertion UseAssertEqualsInsteadOfAssertTrue UseAssertSameInsteadOfAssertTrue UseAssertNullInsteadOfAssertTrue SimplifyBooleanAssertion
	number of discovered bugs	JumbledIncrementer AvoidMultipleUnaryOperators CloseResource BadComparison EqualsNull CompareObjectWithEquals