FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Titel der Arbeit

Optionaler Untertitel der Arbeit

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Pretitle Forename Surname Posttitle
Matrikelnummer 0123456

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Pretitle Forename Surname Posttitle
Mitwirkung: Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle

Wien, 1. Jänner 2001

Forename Surname	Forename Surname

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Title of the Thesis

Optional Subtitle of the Thesis

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Pretitle Forename Surname Posttitle
Registration Number 0123456

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Pretitle Forename Surname Posttitle
Assistance: Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle
Pretitle Forename Surname Posttitle

Vienna, 1st January, 2001

Forename Surname

Forename Surname

Erklärung zur Verfassung der Arbeit

Pretitle Forename Surname Posttitle
Address

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2001

Forename Surname

Danksagung

Ihr Text hier.

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

200-250 words or 10-15 lines Enter your text here.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

- OK software, due to its(steady growing) complexity [LB85](need to read)
  maybe better than the lehman85 cus available [LR03]
  structured programming http://dl.acm.org/citation.cfm?id=1243380
- OK software evolution
  Evelyn Barry , Sandra Slaughter , Chris F. Kemerer, An empirical analysis of software evolution profiles and outcomes, Proceedings of the 20th international conference on Information Systems, p.453-458, December 12-15, 1999, Charlotte, North Carolina, USA
- OK maintenance [LS80] [ISO06]
  T. H. Ng , S. C. Cheung , W. K. Chan , Y. T. Yu, Do Maintainers Utilize Deployed Design Patterns Effectively?, Proceedings of the 29th international conference on Software Engineering, p.168-177, May 20-26, 2007
  code has to be understood [Boe76] in order to make changes or add features [SLea97]
  integrate somewhere here: software -> bug -> understand(up to 60% [Bas97](is this really related? thorough reading may be better) [Pig96]) to fix

Software under lies a continuous changes, throughout its live cycle. The evolution process from the beginning of development until its release and maintenance. Large software[1] and most of all software classified as type E [CHLW06] gets more complex over time. If there are more than a few developers/development teams are involved or the developers/development teams are spread allover the world, there exists more foreign code than self written.

[1] "The term large is, generally, used to describe software whose size in number of lines of code is greater than some arbitrary value. For reasons indicated in [leh79], it is more appropriate to define a large program as one developed by processes involving groups with two or more management levels."[LR03]

- OK program comprehension

Since changes, enhancements or fixes of existing code demand the developers involved to gain a high level of understanding for the software at hand. Due to [CZvD11] "... up to 60% of the maintenance effort is spent on gaining a sufficient understanding of the program ...". This task is referred to by the scientific community as "program understanding" or "program comprehension" and thus these words are considered synonym in this thesis. This thesis addresses the task of improving program comprehension of the concatenative programming language forth on several level.

- OK proper reading as of [Bas97](?) [RCM04]
  systematic approach, strategy may depend on various attributes

- OK mental model(LaToza et al., 2006)
  read: @inproceedingsLieberman:1995:BGC:223904.223969, author = Lieberman, Henry and Fry, Christopher, title = Bridging the Gulf Between Code and Behavior in Programming, booktitle = Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, series = CHI '95, year = 1995, isbn = 0-201-84705-1, location = Denver, Colorado, USA, pages = 480–486, numpages = 7, url = http://dx.doi.org/10.1145/223904.223969, doi = 10.1145/223904.223969, acmid = 223969, publisher = ACM Press/Addison-Wesley Publishing Co., address = New York, NY, USA,

- OK strategies as stated by [SFM99]

- OK dynamic analysis as defined by [Bal99] [CZvD+09]

- OK static analysis as defined by [Bal99]

Namely the the reading of source code, static analysis, dynamic analysis and the assistance of writing readable and easy to understand source code.

- OK concatenative languages -> forth, postscript, factor -> implications from the concatenative nature... ie potential to be more natural to read cause of reverse polish notation
  David Shepherd , Lori Pollock , K. Vijay-Shanker, Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task, Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, p.49-54, June 13-14, 2007, San Diego, California, USA

- OK comparison to oo langs

- OK higher abstraction, hard structure boundaries

- OK paradigm promotes a single shared data structure of high importance and thus may simplify the task of putting all the necessary run-time information visually together(cite someone who says that its important to have all information visible at every point in time). Although there are several stacks, features like arbitrary memory allocation, the focus on stacks is clearly stated.

Due to the nature of concatenative languages, it is possible to write source code which ready very similar to natural language. There are no hard boundaries to the structure of the source code(custom defined loops and control structures) as in object oriented languages. Since forth directly operates only on stacks and memory, the information which is immediately needed to follow program execution is limited to those structures. In contrast, in object oriented languages there is also object state, object life cycle and concurrency of interest.

Darren C. Atkinson , William G. Griswold, The design of whole-program analysis tools, Proceedings of the 18th international conference on Software engineering, p.16-27, March 25-29, 1996, Berlin, Germany

1.2 problem statement (which problem should be solved?)

- OK much work and tools on oo- or procedural languages
- OK not so much on concatenative ~~stack oriented~~ languages... nothing in fact(except maybe kgforth), although maybe similarities to procedural
- OK applicability of oo- and procedural methods for concatenative ~~stack oriented~~ languages at the example of forth
- OK applicability of oo-visualization methods
- OK suggestions of (new) methods(lineout style wordlists/words)

There is plenty of work done on the task of program comprehension in object oriented and procedural languages[CZvD+09], but nearly none on concatenative languages. The qualitative exploratory approach of this thesis does not encourage the formulation of specific hypothesis. Therefore the first question, is the applicability of existing methods and their visualization techniques. The second question to be answered, concerns new approaches, which may be exclusive to concatenative languages or gforth/forth.

1.3 aim of the work

This work aims to better understand how program comprehension is performed in concatenative languages and how it can be made more efficient. The secondary goal is the analysis of the applicability of existing analysis- and visualization methods and

provide modifications to existing visualization methods(and maybe suggestion of new methods). The forth programming language is used as a representative of concatenative languages.

~~demonstration by enhancing the gforth stepping debugger(trace recording, trace visualization, goal-based approach possible)~~

1.4 methodological approach

- qualitative approach , exploratory approach(?)
- proposal
- Preliminary evaluations as defined by [CZvD+09]
- outcome is a subjectiv view of the available methods, and proposed enhancements which have been implementet
- case study of the implemented enhancement
- suggestions of further enhancements

1.5 structure of the work

At first, the available information of a forth program is identified. The next step is to characterize the information and its necessity for program comprehension is investigated. The differences of forth and object oriented languages are summarized and then the applicability of existing analysis and visualization methods is presented. The last part of this thesis investigates probable enhancements and modifications to existing methods and proposes new approaches. After the conclusion, the thesis presents further suggestions to support program comprehension and further topics of research in this direction.

CHAPTER 2

State of the art / analysis of existing approaches

This section presents an overview of the work relevant to program comprehension regarding the aim of the work.

2.1 literature studies

To properly approach the stated problem, the first thing is to understand if and when program understanding is required. Although intuitively obvious, this section will discuss the both, since the approach which developers use to understand programs, can be very different during the life cycle of software. Next the means of understanding programs itself is investigated. Third, the nature of concatenative languages and in particular gforth/forth will be investigated.

2.2 analysis

selected work?

2.2.1 software evolution

[LB85] and [LR03] (beide vllt kritisch zu betrachten und evt out of scope; wenn dann noch in den jÃijngeren citedbys schauen; die grund aussage hier kÃűnnte sein, dass E-Type software immer im wandel befinden wird und immer Ãďnderungen unterliegen wird()aus leh2003))

2.2.2 software maintenance

- types of maintenance

- find bugs and fix them
- find the right place to implement a new feature.
- find the right place to modify a feature.

2.2.3 program comprehension

- structured approach
- thorough reading is the most efficient[cite]
- about the mental model building
- keeping the mental model up to date
- keeping artifacts up to date

2.2.4 program comprehension strategies

- top down
- bottom up
- knowledgebased
- systematic and as-needed
- integrated approaches

2.2.5 analysis to support program understanding

Several analysis types

- source code reading
- documentation reading(everything except source code)
- static analysis
- dynamic analysis
- post mortem analysis
- realtime analysis

dynamic analysis

- about realtime/interactive vs post mortem
- actual behavior
- incomplete view [Bal99]
- observer effect
  Andrews, J. (1997). Testing using log file analysis: tools, methods, and issues. In Proc. International Conference on Automated Software Engineering (ASE), pages 157âĂŞ 166. IEEE Computer Society Press
- scalability
  Zaidman, A. (2006). Scalability Solutions for Program Comprehension through Dynamic Analysis. PhD thesis, University of Antwerp
- debugging -> different kind of paradigms and languages and tools
  see @incollectionreiss1993trace, title=Trace-based debugging, author=Reiss, Steven P, booktitle=Automated and Algorithmic Debugging, pages=305–314, year=1993, publisher=Springer
- about debugging
- dataflow analysis(Backward Analysis)(not sufficient in demo)
  Darren C. Atkinson , William G. Griswold, Implementation Techniques for Efficient Data-Flow Analysis of Large Programs, Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), p.52, November 07-09, 2001

static analysis

- OK complete view
- OK no actual data present
- OK architecture and design documents

Static analysis is ...[cite]... not running code. Therefore and in contrast to dynamic analysis, it has the capability to provide a complete view of the software at hands. The drawback is that there is no actual data present and thus there is no mean of covering the actual data and follow its manipulation. ...[cite] This makes it a most valuable tool for architecture, design, and algorithm analysis. ...[cite]

2.2.6 applicability to concatenative languages

existing methods abstract(abstract like print debugging and stepping and so on) furthermore the abstraction of all those methods mentioned above to find similarities and then adapt them to fit the characteristics of concatenative languages. applicability for concatenative languages

2.3 visualization to support program understanding maybe some examples(and tools)

- sequence diagram
- circular diagram and interactive interaction sequance diagram [Cor09]
- interaction diagrams (Jacobson, 1992)/ scenario diagrams (Koskimies and MÃűssenbÃűck 1996)
- information murals (Jerding and Stasko, 1998)
- polymetric views (Ducasse et al., 2004)
- fisheye views (suggested by George W. Furnas, 1986, and formulated by [SM96] and [SB94])
- hierarchical edge bundling (Holten, 2006)
- structural and behavioral views of object-oriented program (Kleyn and Gingrich, 1988)
- matrix visualization and âĂIJexecution patternâĂİ notations [PLVW98] to visualize traces in a scalable manner(De Pauw et al. 1993, 1994, 1998)
- architecture oriented visualization (Sefika et al. 1996)
- a continuous sequence diagram, and the âĂIJinformation muralâĂİ (Jerding and Stasko, 1998)
- architecture with dynamic information (Walker et al. 1998)
- frequency spectrum analysis (Ball 1999)

2.4 comparison and summary of existing approaches

existing approaches for gforth/forth and relation to above mentioned stuff

- kgforth http://sourceforge.net/projects/kgforth/
- existing methods(actual methods)
  - factoring (http://en.wikipedia.org/wiki/Modular_programming https://www.complang.tuwie
    html/Factoring-Tutorial.html http://www.ultratechnology.com/Forth-factors.htm)
  - aliasing
  - organization of word lists
  - source code documentation

– other documentation artifacts
– dump
– ., / and type
– dbg
– see and code-see
– ~~

CHAPTER 3

Methodology

- qualitative approach
- exploratory case study
- prototype
- sketches
- trying to understand programs developed withing stackbasierte programmierung vl?

3.1 used concepts

- prototyping
- reading codes
- print-debugging
- step-debugging

3.2 methods and/or models

- prototyping

3.3 languages

- postscript
- forth

- shell script
- c
- m2

3.4 design methods

?

3.5 data models

?

3.6 analysis methods

- reading code
- tail and error

3.7 formalisms

?

CHAPTER 4

Suggested solution/implementation

kind of an ~~ide~~ development environment
light table ide(js) continuous reverse engineering idea of [MJS+00] to provide immediate response of the systems output... although probably not applicable or very time consuming in setup(or not more than integration testing...) for most industrial scale software
eclipse ide(java)

4.1 suggested solution

- OK emphasis on on comprehension code while writing. factoring suggestion, documentation, aliases(same code with multiple aliases to read more natural at different points in programs), expressive naming, hard to generalize cause of the flexibility the language provides
- OK adequate search and cross reference facilities to support systematical investigation to benefit from effective program understanding as stated by [RCM04]
- OK display of the 'vocabulary' [cite moore: remember all the words]

A very important question in this concern is, how can developers be assisted to write readable code. Experienced developers may do that intuitively, but how can novice developers be encouraged and supported to write readable code. Concatenative languages are flexible enough to produce code very similar to natural languages, but how can this attribute be supported?
One answer is to provide hints based on static analysis.
It is not possible to make every word completely readable and the perceived readability also depends on the experience of the developer. At some point it always comes down to

longer combinations of "nip tuck over rot", this is hardly avoidable at the lowest level. Thus, proper documentation of words is essential. Its pretty The obvious, that stack effect comment[1] in forth, are a must have, but also the behavior of the word should be explained if complex or not very natural to read words[2]. Another advantage of word definition comments is the possibility of automated documentation generation.
Very long word definitions tend increase the amount of brain capacity required to understand its behavior. The way to account this problem is to break down the overall task into manageable pieces. It is called factoring[3] in context of concatenative languages. An approach could be to place a hint on word definitions which exceed a certain amount of lines or words or different words and suggest further factoring.
Another tool to make code read more natural, is aliasing[4]. By defining aliases for a certain word, its functionality can be used in different contexts and still read very natural.
Expressive naming, although obvious, it should be mentioned that assigning expressive and fitting names for words is essential. This applies to any language[does it? cite...].
To understand code, the systematic approach turned out to be most efficient[RCM04].
To ease afford of finding the definition of words used at a certain point, a hyperlink like referencing mechanism can be used[cite the visualization paper with the hyperlink feature].
As stated by Charles D. Moore in [BW09]: "... The challenge there is 1) deciding which words are useful, and 2) remembering them all.", when programs get larger, the amount of words can grow big. Thus it is suggested to have some sort of a dictionary to search the whole vocabulary by name, stack effect comment, word definition documentation and provide a reverence to where they are used. Auto completion can also help a lot in finding words previously defined.

- other data structures and variables should be displayed
  - memory maybe like [Rei95] or [AKG+10] but since there is no underlying object orientation and no standardized oo system this would be hard do accomplish
  - fisheye or word cloud like display(tree or sugiyama as of [SWFM97])
- interactive program manipulation: state of the system before a word, after a word and by clicking on the word jumping to its definition or inserting it and there also providing those features
- stepping debugger mode: simply stepping through the whole code word by word

[1] See https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Stack_002dEffect-Comments-Tutorial.html#Stack_002dEffect-Comments-Tutorial

[2] Most notable G in gforth. See https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Comments.html#Comments

[3] See https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Factoring-Tutorial.html#Factoring-Tutorial

[4] See https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Aliases.html

- goal-oriented strategy: the definition of an execution scenario such that only the parts of interest of the software system are analyzed (Koenemann and Robertson, 1991; Zaidman, 2006).
- code analysis and visualization facilities see chapter 2 TODO

4.2 implementation

- OK proof of concept by enhancement of stepping debugger on forth code level(cause it has turned out to be the fastest and simples approach) by showing additional data: the other stacks

As a first step, in addition to this work, the stepping debugger of gforth was enhanced. The existing implementation only shows the data stack and the word to be executed. The enhanced version is now able to record the debugging trace as a postscript(trace.ps) file which shows the executed words and the state of the data stack, the float stack and the return stack. Since the return stack contains addresses related to the executed words, the word names were displayed when possible. The algorithm to resolve the word names corresponding to the address was already implemented in gforth's back-trace and not developed by myself. The debugging trace is stored as a postscript file and can be investigated during the debugging session as well as after the actual execution is complete. The implementation involved the modification of some gforth internal files as well as a file(gfvis.fs) which has to be loaded with gforth. The display layout was implemented in postscript. Every trace-file is constructed from a template file(gfvis.ps) which contains the postscript code to layout the recorded trace. (Since the debugger only works with the itc engine, the debugging has still to be performed with this engine.????) There was also made an attempt to accomplish this on the c source code level, but it turned out to be rather complicated and therefore the forth only level was chosen.

CHAPTER 5

Critical reflection

5.1 comparison with related work

Quiet some research has been done on the topic of program comprehension in the last decades, but most of it addressed object oriented or procedural languages. Since there is no standard way to model object orientation, it is not possible to implement a general tool for visualization similar to the existing methods, but these methods should be applicable on an object oriented model implemented in forth in general. The research on trace visualization has proven to be very similar for concatenative languages since a program is also a concatenation of words. The most similar existing work is kgforth, a development environment using qt. Although the outline looks promising, it is not in development anymore and I was not yet able to run it.

5.2 discussion of open issues

- OK not scaling well cause of limited screen real estate and thus the need to scroll
- OK not scalign well cause of unpredictable stack height(maybe show only depth according to stack effect comment)
- OK not suitable for performance meassuring cause debugger...

Concerning the implemented demo, most obvious is the lack of usability. First of all the second window turned out to be rather annoying, it would have been better to include the visualization within the gforth window and record the trace in a standardized data structure in a separate file. Another not yet addressed problem is the scalability of the view, the inefficient use of the screen real estate makes it unusable for very long traces. A possible solution would be to limit the displayed float and data stack depth to a certain number since it is not encouraged to manipulate more than some of the

most upper stack elements. Or limit the depth per word to the number of elements as defined in the stack effect comment. Another improvement would be the implementation of an interactive trace sequence view, like the "massive sequence" view implemented by [CZH+08], where in addition nested words as well as the stack state between arbitrary words can be hidden and displayed on demand. The introduction of "watch points" to reduce the visual noise the size of trace file could also address the scale issue. It is also not practical to compare two traces to each other and it is very hard to understand the behavior of large systems. [A visualization of several traces like the massive sequence view where several traces are synchronized by word and differences are visualizes by colors could solve this problem.?????]

Due to the implementation of the trace recording within the debugger, it is not possible to collect traces of live systems. Neither is performance analysis possible while recording traces/debugging.

- nature of gforth
  - interpretation/compilation mix(how to integrate the adhook changes between modes '[]')
  - implementation within the executing system(??)
  - lack of dynamic information(return stack add -> wordname heuristic)

CHAPTER 6

Summary and future work

The pure exploratory approach did not provide any information on the actual impact of the implemented and suggested solutions. This gathering of quantitative date and the formulation of hypotheses remains to be done in further works.

summary of what has been done and the subjective conclusion

- how does software maintenance work in those
- ide
- using a standard data type to store traces
- display of variable content
- display of allocated memory areas
- display of color diff with tooltip of previous values for stacks and memory areas
- (better visualization of loops and control structures) is this even possible?
- (display of the full program as a graph) is this even possible?
- (customizable inspection depth) ?
- static code analysis
  - stack depth per word
  - type system for forth
  - ...

conclusion like what i contributed to the community!!

work on program comprehension of concatenative languages good overview of the field [CDPC11] and [Cor09]

Bibliography

[AKG+10] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[Bal99] Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[Bas97] Victor R. Basili. Evolving and packaging reading technologies. *J. Syst. Softw.*, 38(1):3–12, July 1997.

[Boe76] B. W. Boehm. Software engineering. *IEEE Trans. Comput.*, 25(12):1226–1241, December 1976.

[BW09] Federico Biancuzzi and Shane Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, Inc., 1st edition, 2009.

[CDPC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, April 2011.

[CHLW06] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. Evolution in software systems: Foundations of the spe classification scheme: Research articles. *J. Softw. Maint. Evol.*, 18(1):1–35, January 2006.

[Cor09] Bas Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Wohrmann Print Service, 2009.

[CZH+08] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(12):2252–2268, December 2008.

[CZvD+09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Softw. Eng.*, 35(5):684–702, September 2009.

[CZvD11] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Trans. Softw. Eng.*, 37(3):341–355, May 2011.

[ISO06] ISO. Software engineering – software life cycle processes – maintenance. ISO 14764:2006, International Organization for Standardization, Geneva, Switzerland, 2006.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LR03] Meir M. Lehman and Juan F. Ramil. Software evolution: Background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, October 2003.

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 47–60, New York, NY, USA, 2000. ACM.

[Pig96] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[PLVW98] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS âĂŹ98*, pages 219–234, 1998.

[RCM04] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, December 2004.

[Rei95] S. P. Reiss. *Visualization for Software Engineering – Programming Environments*. 1995.

[SB94] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–83, December 1994.

[SFM99] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44(3):171–185, January 1999.

[SLea97] Janice Singer, Timothy C. Lethbridge, and et al. An examination of software engineering work practices, 1997.

[SM96] Margaret-Anne D. Storey and Hausi A. Müller. Graph layout adjustment strategies. In *Proceedings of the Symposium on Graph Drawing*, GD '95, pages 487–499, London, UK, UK, 1996. Springer-Verlag.

[SWFM97] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Mueller. On integrating visualization techniques for effective software exploration. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, INFOVIS '97, pages 38–, Washington, DC, USA, 1997. IEEE Computer Society.

Glossary

Charles D. Moore Charles D. Moore, is the inventor of the forth programming language.. 14