Second IEEE International Workshop on Visualizing Software for Understanding and Analysis

22nd September 2003
Amsterdam, Netherlands

Sponsored by NWO

Also sponsored by the IEEE Computer Society Technical Committee on Visualization and Graphics (TCVG) and the IEEE Computer Society Technical Council on Software Engineering (TCSE), and the IEEE Computer Society.

Editors: Arie van Deursen, Claire Knight, Jonathan I. Maletic, and Margaret-Anne Storey

Welcome

We would like to extend a warm welcome to the 2nd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2003)! This exciting event gathers software visualization researchers from all over the world. The goal of this workshop is to showcase state-of-the-art research in software visualization and be a breeding ground for new advances.

This one-day working event is organized to help promote interaction, investigation, and collaboration among participants. We are very happy to start the day with a keynote tutorial from Professor Colin Ware. Professor Ware is a leading researcher in information visualization. Following his talk, a set of short presentations relating to tools and techniques for software visualization will be presented. The afternoon will start with a number of hands-on tool demonstrations. This session will have a free format and we encourage everyone to explore and discuss these working research efforts. A half hour will be set aside for a group discussion concerning the demonstrations and the morning talks. The remainder of the afternoon will be organized into short talks with group discussion at the end of each session. These sessions focus on a number of key issues and broader challenges for software visualization.

We are indebted to Hans van Vliet and ICSM 2003 who agreed to our co-located event and we thank them for their help. We are also indebted to NWO, the Netherlands Organization for Scientific Research, for covering the costs of the keynote tutorial by Colin Ware.

We sincerely hope you find this event to be stimulating and rewarding and that you have a very enjoyable stay in Amsterdam!

Arie van Deursen, Claire Knight, Jonathan I. Maletic, and Margaret-Anne Storey

Organisers

Table of Contents

Welcome ***ii***

Table of Contents ***iii***

Agenda ***v***

Tutorial/Keynote ***1***

Thinking with Interactive Visualization *2*
Colin Ware

Workshop Part I ***29***

UML Class Diagrams – State of the Art in Layout Techniques *30*
Holger Eichelberger and Jurgen Wolff von Gudenberg

Techniques for Reducing the Complexity of Object-Oriented Execution Traces *35*
Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge

ADG: Annotated Dependency Graphs for Software Understanding *41*
Ahmed E. Hassan and Richard C. Holt

Exploring the Many Architectures of a Very Large Component-Based Software *46*
Jean-MarieFavre, R. Sanlaville, and J. Estublier

AutoCode: Using Memex-like Trails to Improve Program Comprehension *48*
Richard Wheeldon, Steve Counsell, Kevin Keenoy

Hands-on Collaborative Demo ***53***

Exploring the Many Architectures of a Very Large Component-Based Software *46*
Jean-MarieFavre, R. Sanlaville, and J. Estublier

CodeCrawler – A Lightweight Software Visualization Tool *54*
Michele Lanza

AutoCode: Using Memex-like Trails to Improve Program Comprehension *56*
Richard Wheeldon, Steve Counsell, Kevin Keenoy

Demonstration of Advanced Layout of UML Class Diagrams by SugiBib *58*
Holger Eichelberger and Jurgen Wolff

GENISOM: Self-Organizing Maps Applied in Visualising Large Software Collections *60*
James Brittle and Cornelia Boldyreff

Source Viewer 3D (sv3D): A System for Visualizing Multi Dimensional Software Analysis Data *62*
Andrian Marcus, Louis Feng, Jonathan I. Maletic

Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse *64*
Robert Lintern, Margaret-Anne Storey, Xiaomin Wu, Jeff Michaud

Workshop Part II ***66***

Program Visualization Support for Highly Iterative Development Environments *67*
Michele Lanza

Challenges in Visualizing and Reconstructing Architectural Views *73*
Juergen Rilling and Michel Lizotte

Visualization to Support Version Control Software: Suggested Requirements *80*
Xiaomin Wu, Adam Murray, Margaret-Anne Storey, Robert Lintern

Visualization for Software Risk Assessments 87
Jordi Vidal Rodriguez and Tobias Kuipers

MetaViz – Issues in Software Visualizing Beyond 3D 92
Juergen Rilling, Jianqun Wang, and S. P. Mudur

KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs 98
Timothy A. Davis, Kenneth Pestka, and Alan Kaplan

Self-Organizing Maps Applied in Visualising Large Software Collections 104
James Brittle and Cornelia Boldyreff

The end of the line for Software Visualisation? 110
Stuart M. Charters, Nigel Thomas, and Malcolm Munro

CFB: A Call for Benchmarks – for Software Visualization 113
Jonathan I. Maletic and Andrian Marcus

Notes ***117***

Agenda

8:30-8:45	**Welcome/Coffee**
8:45-10:15	**Tutorial/Keynote** *Thinking with Interactive Visualization* by Dr. Colin Ware, Data Visualization Research Lab. University of New Hampshire. This talk will outline a theory of how reasoning can be augmented with visualizations of data. According to current cognitive theory both visual and verbal-logical processes rely on limited capacity working memories. Active attention is a core process guiding problem solving through a set of nested cognitive control loops. To take advantage of visualizations, problems are converted into visual hypotheses in the form of prototypical patterns. Active attention, "grasps" relevant patterns from the visual display to test hypotheses and find solutions. In addition visual symbols extend our memory capacity by evoking verbal/logical constructs causing them to be loaded into non-visual working memory. This theory is elaborated with examples from the way people work with maps and software diagrams.
10:30-11:00	Coffee Break
11:00-12:00	**Workshop Part I: Short presentations** ***Tools & Techniques*** *Chair: Arie Van Deursen, CWI, The Netherlands* UML Class Diagrams-State of the Art in Layout Techniques Holger Eichelberger and Jurgen Wolff von Gudenberg, Wurzburg University, Germany Techniques for Reducing the Complexity of Object-Oriented Execution Traces Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge School of Information Technology and Engineering (SITE), University of Ottawa ADG: Annotated Dependency Graphs for Software Understanding Ahmed E. Hassan and Richard C. Holt, Software Architecture Group (SWAG), University of Waterloo Exploring the Many Architectures of a Very Large Component-Based Software Jean-MarieFavre, R. Sanlaville, and J. Estublier, Adele Team, Laboratoire LSR-IMAG, University of Grenoble, France AutoCode: Using Memex-like Trails to Improve Program Comprehension Richard Wheeldon, Steve Counsell, Kevin Keenoy, Dept of Computer Science, University of London
12:00-2:00	**Hands-on Collaborative Demo (with a short Break for Lunch)** Exploring the Many Architectures of a Very Large Component-Based Software Jean-MarieFavre, R. Sanlaville, and J. Estublier, Adele Team, Laboratoire LSR-IMAG, University of Grenoble, France CodeCrawler - A LightweightSoftwareVisualizationTool Michele Lanza, Software Composition Group-University of Bern, Switzerland AutoCode: Using Memex-like Trails to Improve Program Comprehension Richard Wheeldon, Steve Counsell, Kevin Keenoy , Dept of Computer Science, University of London Demonstration of Advanced Layout of UML Class Diagrams by SugiBib Holger Eichelberger and Jurgen Wolff von Gudenberg, Wurzburg University, Germany GENISOM: Self-Organizing Maps Applied in Visualising Large Software Collections James Brittle and Cornelia Boldyreff, Dept. of Computer Science, University of Durham Source Viewer 3D (sv3D): A System for Visualizing Multi Dimensional Software Analysis Data Andrian Marcus, Louis Feng, Jonathan I. Maletic , Affiliation: Kent State University

	Integrating A Visualization Tool with Eclipse Robert Lintern, Margaret-Anne Storey, Xiaomin Wu, Jeff Michaud, CHISEL Group, University of Victoria
2:00-2:30	**Discussion on Hands on tool demonstration** Tool users and participants will be invited to share their insights and observations from the tool demonstrations.
2:30-3:30	**Workshop Part II** ***Requirements and Challenges for Software Visualization*** *Chair: Arie Van Deursen* Program Visualization Support for Highly Iterative Development Environments Michele Lanza, Software Composition Group-University of Bern, Switzerland Challenges in Visualizing and Reconstructing Architectural Views Juergen Rilling and Michel Lizotte Department of Computer Science, Concordia University and Defense R&D, Canada Requirements for Visualizing Version Control Information Xiaomin Wu, Adam Murray, Margaret-Anne Storey, Robert Lintern, University of Victoria Visualization for Software Risk Assessments Jordi Vidal Rodriguez and Tobias Kuipers, Software Improvement Group **Discussion**
3:30-4:00	**Break**
4:00-4:45	***Session: Visualizing Software in 3D -- When Should We?*** *Chair: Colin Ware, University of New Hampshire* MetaViz – Issues in Software Visualizing Beyond 3D Juergen Rilling, Jianqun Wang, and S. P. Mudur, Department of Computer Science, Concordia University KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs Timothy A. Davis, Kenneth Pestka, and Alan Kaplan, Clemson University and Panasonic Technologies Inc Self-Organizing Maps Applied in Visualising Large Software Collections James Brittle and Cornelia Boldyreff, Dept. of Computer Science, University of Durham **Discussion**
4:45-5:30	***Session: Improving Software Visualization -- How Can We?*** *Chair: Susan Sim, University of California at Irvine* The end of the line for Software Visualisation? Stuart M. Charters, Nigel Thomas, and Malcolm Munro, Visualisation Research Group, Durham University CFB: A Call For Benchmarks - for Software Visualization Jonathan I. Maletic and Andrian Marcus, Kent State University **Discussion**
5:30-5:45	**Wrap-up and summary discussion**
6:00-9:00	**Dinner** Drinks at 6pm; Dinner at 7pm Liefhebber Restaurant, Kloveniersburgwal 5, 1011 JT Amsterdam http://www.liefhebber.com/ Tel. +31 20 4200418

Tutorial/Keynote

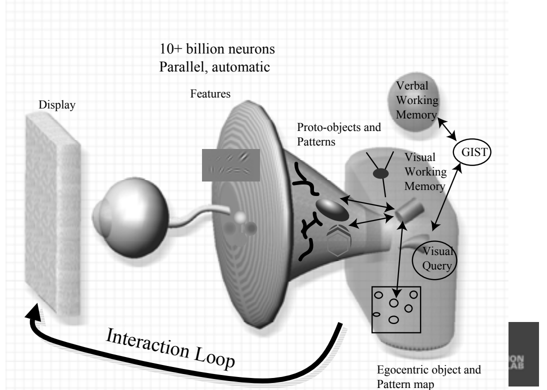
Thinking with Interactive Visualization

Colin Ware
Data Visualization Research Lab
University of New Hampshire

Outline

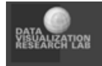
- The problem solving system
- Pre-attentive (what is low cost)
- Patterns
- 2D vs 3D?
- Visual thinking and the cost of knowledge

Architecture for visual thinking



Pre-Attentive Processing

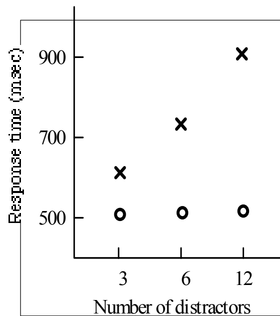
897390570927940579629765098294
08028085080830802809850- 802808
567847298872t y4582020947577200
21789843890r 455790456099272188
897594797902855892594573979209



Color is Pre-Attentive (Pops out)

897390570927940579629765098294
08028085080830802809850- 802808
567847298872t y4582020947577200
21789843890r 455790456099272188
897594797902855892594573979209

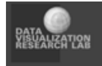
Generic Pre-Attentive Experiment



- Number of irrelevant items varies
- Pre-attentive 10 msec per item or better.

Preattentive popout cues

- Color
- Shape
- Motion
- Size
- Simple Shading
- Conjunctions do not popout



Conjunctions of motion and shape do pop out. (color also?)

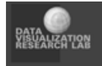
- McLeod, P., Driver, J. and Crisp, J. (1988) Visual search for a conjunction of movement and form is parallel. Nature 332, 154-155.
- Driver, J., MacLeod, P. and Dienes, Z. (1992) Motion coherence and conjunction search: Implications for guided search theory. Perception and Psychophysics. 51, 1, 79-85.

MEGraph: Experimental system

- Allows for various topological range highlighting methods

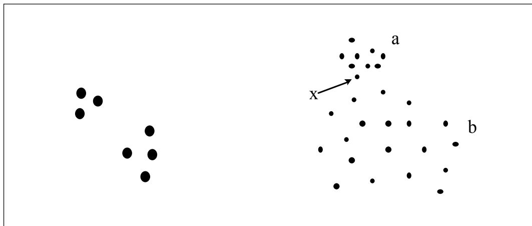
Stage 2 Pattern perception

- Gestalt principles
- Proximity
- Continuity
- Connectedness
- Closure



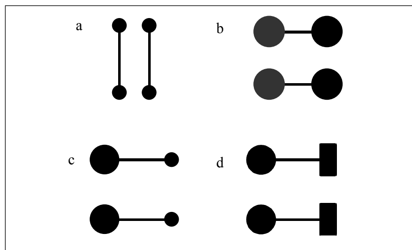
Proximity

- Emphasize relationship by proximity
- Spatial Concentration



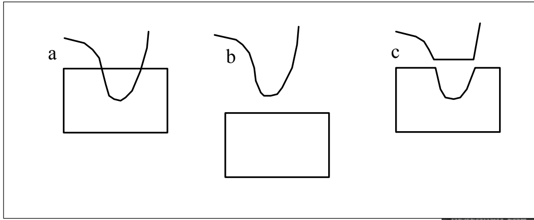
Connectedness

- Connectedness assumed in Continuity

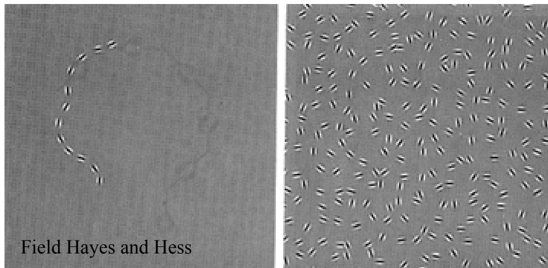


Continuity

- Visual entities tend to be smooth and continuous

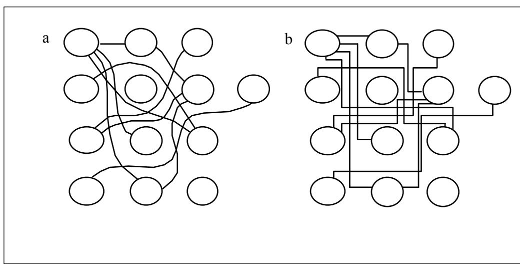


Neural basis



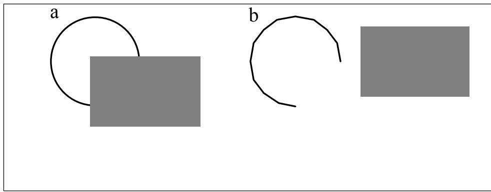
Continuity in Diagrams

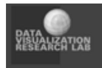
- Connections using smooth lines



Closure

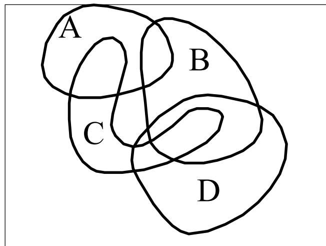
- Prefer closed contours

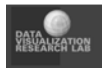




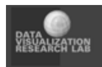
Closure (cont.)

- Closed contours to show set relationship



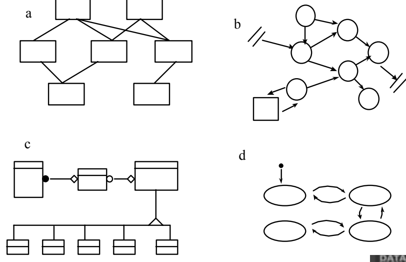


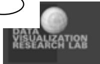
Extending the Venn-Euler Diagram



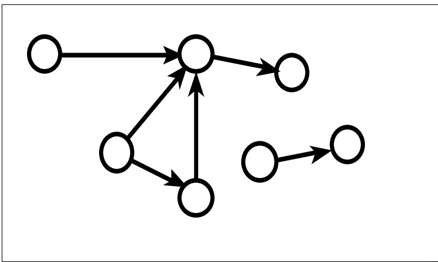
Patterns in Diagrams

- Entities – objects
- Relationships – links, color, etc

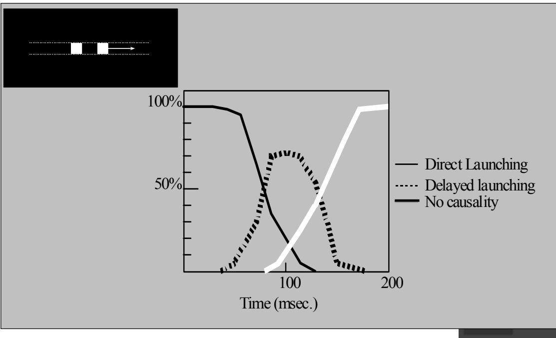




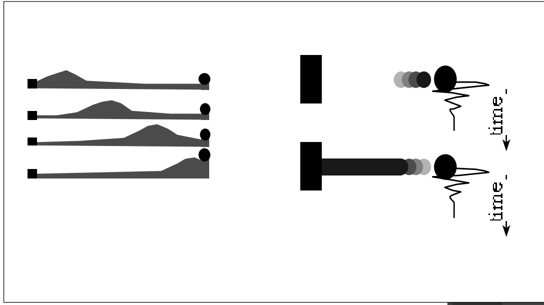
A causal graph



Michotte's Causality Perception

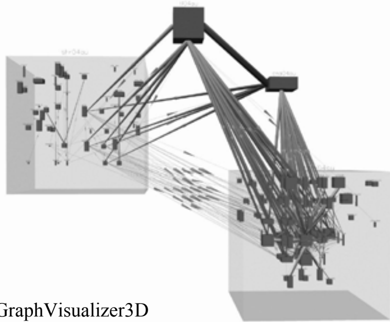


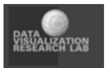
Visual Causal Vectors

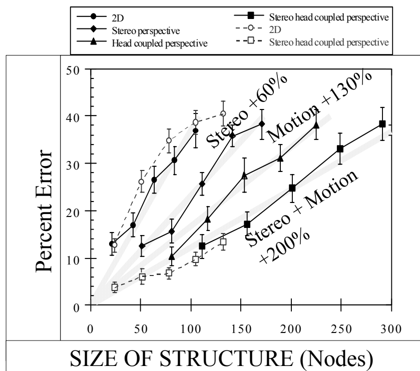


3D pattern perception

- Use 3D?

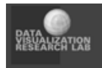




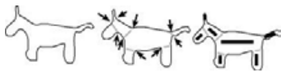


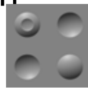
Issues

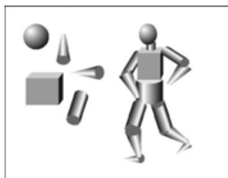
- Can see a larger graph
- Must have stereo and motion
- But consider the cost of interaction

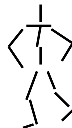


Another form of 3D Structured Object Perception









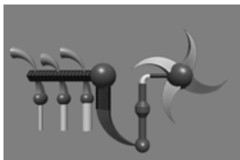
3D Primitives "Geons"
Structural skeleton

Shape from shading
is also primitive

Color and texture are
Secondary attributes

Geon Diagram (Pourang Irani)

- Major entities should be represented with simple 3D shape primitives
- Links can be represented by connecting geons (the structural skeleton)
- Geons should be shaded to make 3D shape visible
- Secondary attributes -> color and surface texture
- Layout of structure should be primarily in 2D plane



Structure diagrams (UML)

Pattern finding & Recognition

13% errors: 4.3 sec sub-structure

22% memory errors

26% errors 7.1 sec sub-structure

42% memory errors

3D versus 2D

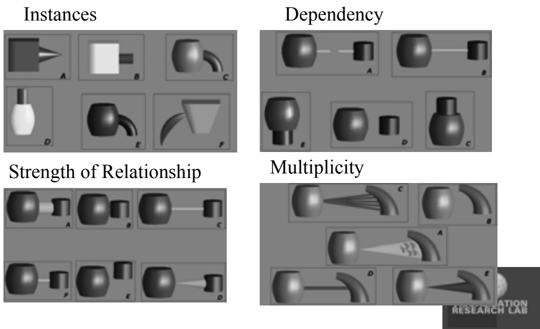
11.4% errors 3.7 sec sub-structure

20% memory errors

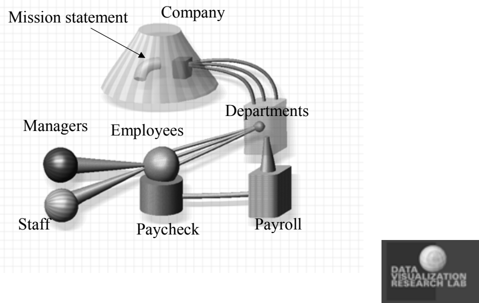
21% errors 5.1 sec sub-structure

34% memory errors

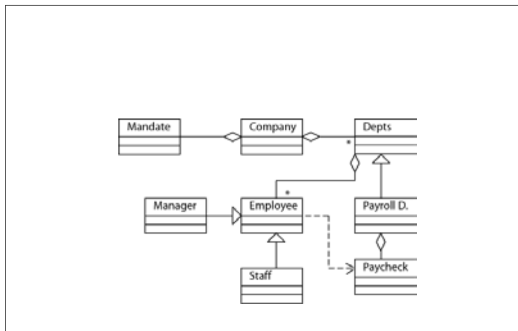
Natural semantics



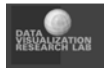
Geon Diagram with semantics



UML MODELING

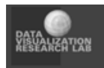


Geon Diagrams

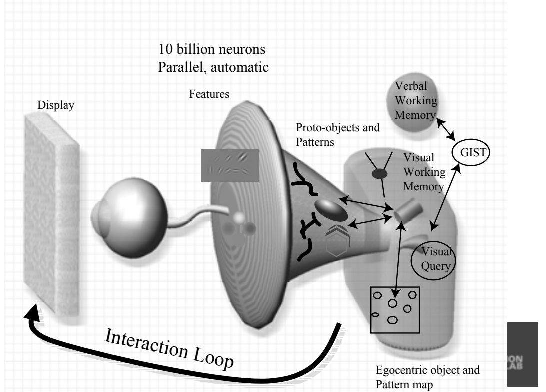


Geon Diagrams

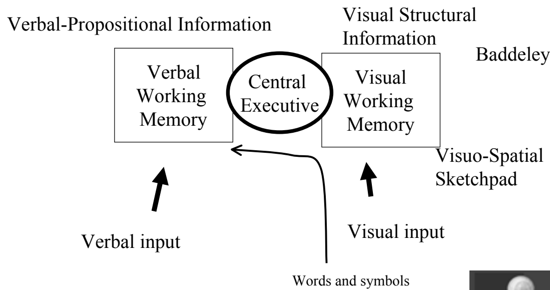
- Advantages
  - More memorable
  - Easier to interpret
- Disadvantages
  - Do not work well with text
  - Inflexible wrt layout

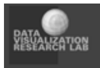


Architecture for visual thinking



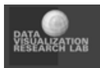
Dual Coding Theory





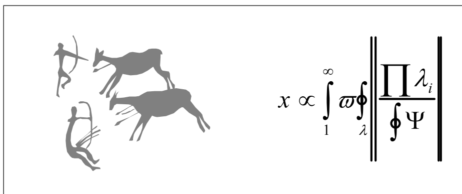
Pictures and Words

- When should we use a visual display?
- What is a visual language?
- Dual coding theory?
- How to integrate images and words



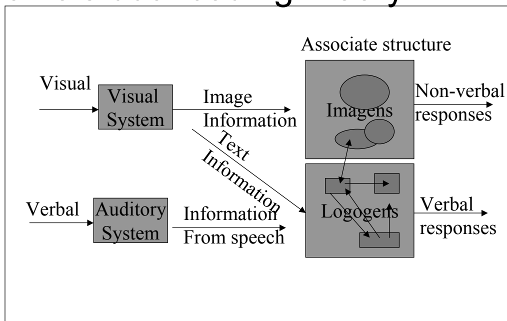
Consider that hieroglyphs gave way to more abstract symbols

- Why turn back the clock?



$$x \propto \int_1^\infty \varpi \oint_\lambda \left\| \frac{\prod \lambda_i}{\oint \Psi} \right\|$$

Pavio's dual coding theory

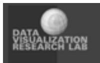


Theory: Graphics and Words

- Graphics for structural logic
- Words for procedural logic: conditionals, qualifiers, if-then else, while.

The nature of language

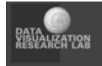
- Chomsky, innate deep structures.
- Common to computer languages
- Critical period for language development
- But being verbal is not essential to language development
- Sign languages for the deaf are the most perfect examples of visual language



What is language

- Description
- Communication of intention
- The ability to communicate procedures and sequences of operations – including logic – if, but, causes, do ***a*** then ***b*** then ***c***

- ***Thus far we have only dealt with description***



Sign languages

- Are true languages
- Developed spontaneously
- Developed independently
- Start as representations
- Become more abstract over time

Can there be a true visual language?

Yes,
But not for most of us!!

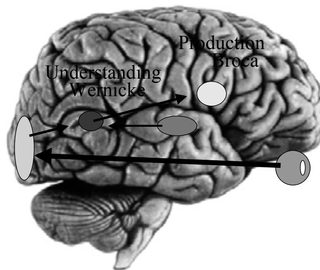
Consider verbal language

A critical period

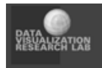
Abstraction, logic
(if, while, perhaps)

Based on speech

Sign languages are true
Visual languages

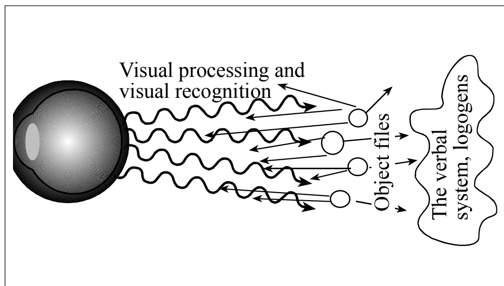


To be fluent in visual language we should be trained from early in life



The visual system gives us

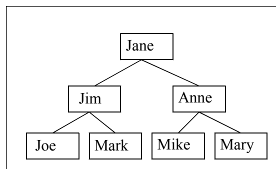
Rapid recognition and pattern finding

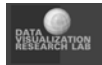


Abstraction

- Jane is Jim's boss
- Jim is Joe's boss
- Anne works for Jane
- Mark works for Jim
- Anne is Mary's boss
- Anne is Mike's boss

Pattern

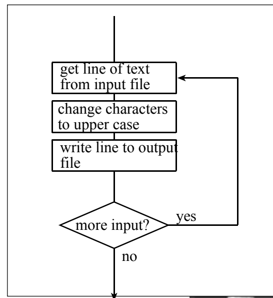




Visual and verbal pseudo-code

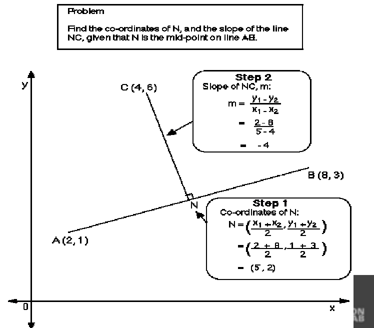
- While letters in stack
  - Take a letter
  - Put a stamp on it
  - Put it in the 'out tray'

Visual programming languages have a history of failure

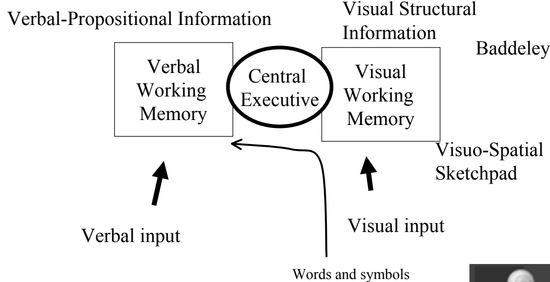


Data flow diagrams are defunct

Integrated pictures and words more Effective: Chandler and Sweller 1991

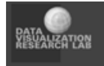


Working memory capacities ~ 3



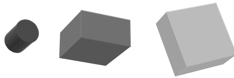
Capacity of verbal working memory

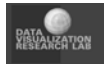
- Used to be though of a 7 +/- 2
- It is now thought of as more a duration of proto-verbal codes.



Capacity of visual working memory (Vogal, Woodman, Luck, 2001)

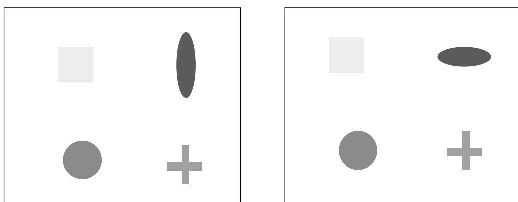
- Task – change detection —— 1 second
- Can see 3.3 objects
- Each object can be complex

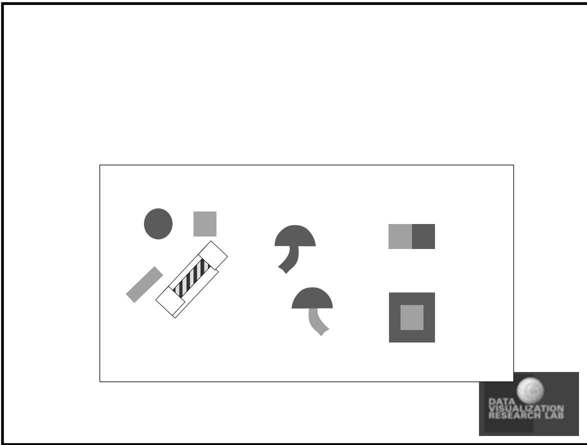


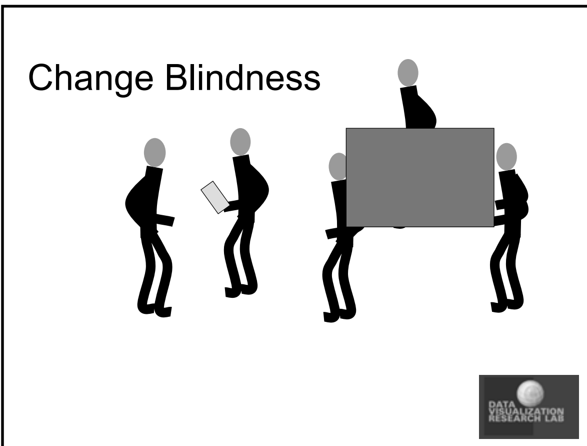


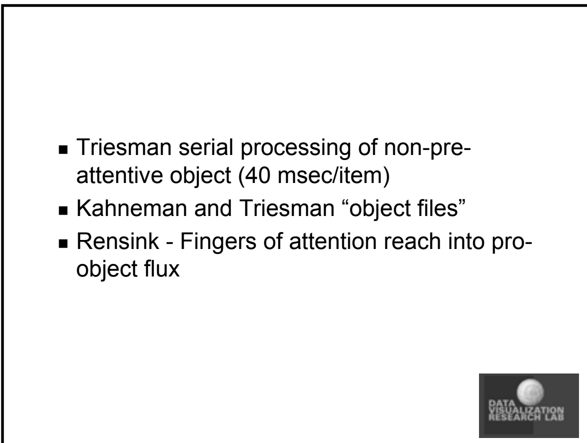
Sequential comparison task

1 sec delay

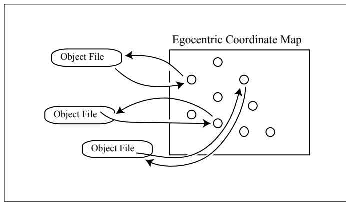



DATA
VISUALIZATION
RESEARCH LAB

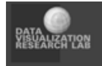

Change Blindness
DATA
VISUALIZATION
RESEARCH LAB


Triesman serial processing of non-pre-attentive object (40 msec/item)
Kahneman and Triesman "object files"
Rensink - Fingers of attention reach into pro-object flux
DATA
VISUALIZATION
RESEARCH LAB

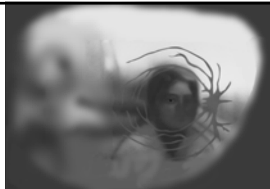
Other components of working memory

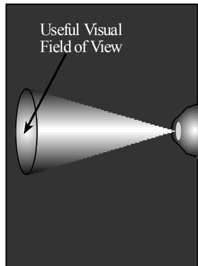


Gist Semantic content



Visual search





Visual Search or Monitoring Strategy

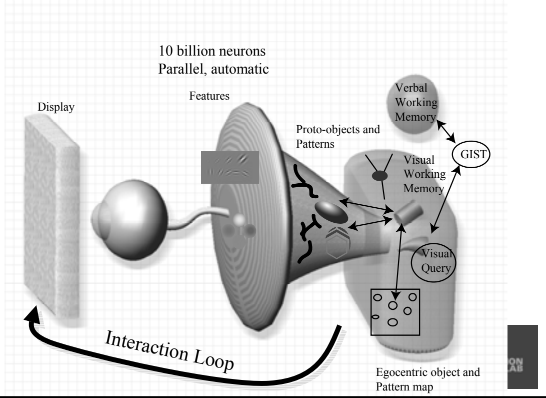
Eye Movement Control

Eye movements

- Two or three a second
- Preserves Context

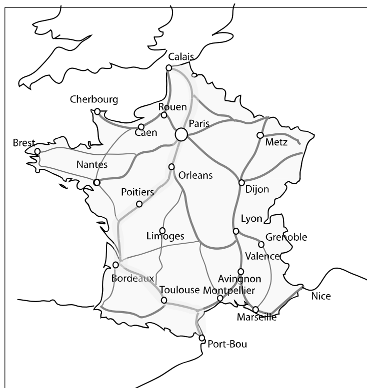
- The screen is a kind of buffer for visual ideas – we cannot see it all at once but we can sample it rapidly

Architecture for visual thinking



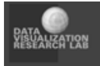
Thinking visually
Embedded processes

- Define problem and steps to solution
  - Formulate parts of problem as visual questions/hypotheses
    - Setup search for patterns
      - Eye movement control loop
        - IntraSaccadic Scanning Loop
          (form objects from proto-object flux



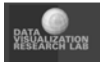
Problem

- Trip Port Bou- Calais (5 days 3 citise)
  - Visual Problem Mayor Highways
    - – Distance < 1.2 min = red smooth path
    - Eye movements to identify major candidate pathways
      - Pattern Identification: smooth, red, connected segments / reject non-red-wrong direction
- Part solutions into vwm – spatial markers
- Parts may be handed to verbal wm



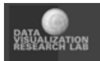
Software Engineering Example - with Graph Representation

- Segment Big Module into parts
  - High Cohesion (semantics)
  - Low Coupling
    - Find highly connected subgraphs with minimal links
      - Scan for candidate patterns
        - Look for Low connectivity
        - Look for Semantic similarity (symbols)
- Important question: what are relevant pattern that can fit in vwm



Cost of Knowledge

- How do we navigate.
- Intra-saccade (0.04 sec)
- An eye movement (0.5 sec)
- A hypertext click (1.5 sec but loss of context)
- A pan or scroll (3 sec but we don't get far)
- Walking (30 sec. we don't get far)
- Flying (faster can be tuned)
- Zooming, fisheye, DragMag

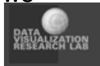


Walking Flying (30 sec +)

Naïve view that does not take perception or the cost of action into account.

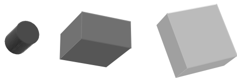
How to navigate large 21/2D spaces? Zooming Vs Multiple Windows

- Key problem: How can we keep focus and maintain context.
- Focus is what we are attending to now.
- Context is what we may wish to attend to.
- 2 solutions: Zooming, multiple windows



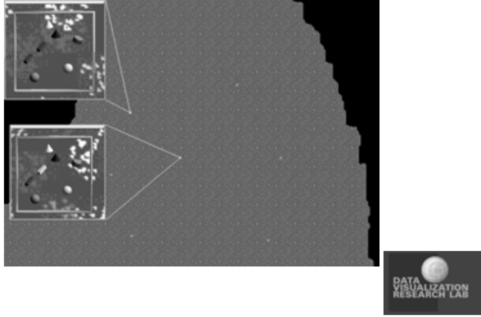
When is zooming better than multiple windows (Matt Plumlee)

- Key insight: Visual working memory is a very limited resource. Only 3 objects



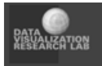
GeoZui3D

Task: searching for target patterns that match

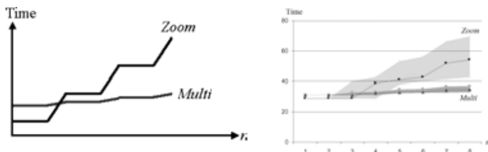


Cognitive Model (grossly simplified)

- Time = setup cost + number of visits*time per visit
- Number of visits is a function of number of objects (& visual complexity)
- When there are too many multiple visits are needed

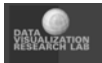


Predictions

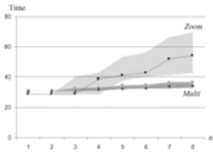


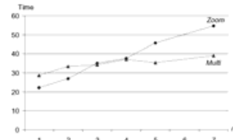
Time =

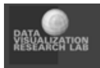
As targets (and visual working memory load) increases, multiple Windows become more attractive.



Prediction Results



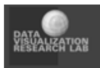




Critical issues: Cognitive costs

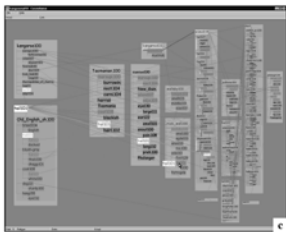
- Clickless queries and cognitive costs
- Medium level – pattern perception
- High level vwm and cognitive costs

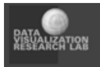
- Assumption: topologically close nodes are more important



Need low cost and low cognitive cost interactions

Constellation: Hover queries (Munzer)





Lessons for design

- Low level- what stands out
- Medium level – pattern perception
- High level vwm and cognitive costs

A large high resolution screen may be the best alternative - because of eye movement

Interactive techiques hold promise

DATA VISUALIZATION RESEARCH LAB

Implications for design of information spaces

- Design for pattern perception, three chunk patterns
- Consider attention
- Implications for navigation
  - Make all navigation devices visible.
  - Do not ever make people walk
  - Maintain context
  - Minimize cognitive load of navigation
  - Use multiple linked views for more complex pattern integration

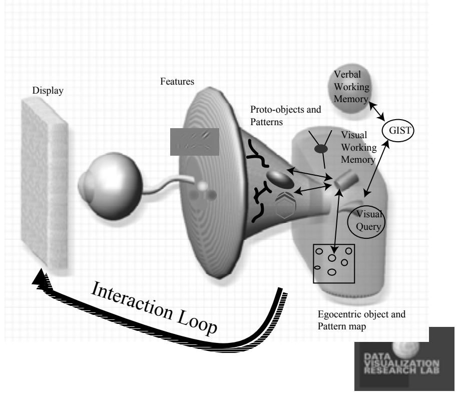
DATA VISUALIZATION RESEARCH LAB

Research topic What are easy visual queries

Easy= single object comparison in vwm

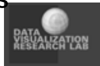
DATA VISUALIZATION RESEARCH LAB

Architecture for visual thinking



Cognitive Systems

- Humans with cognitive tools functioning groups
- Visualization for pattern finding
- Coding for pre-filtering
- Slogan: “Tighten the loop”
- Large displays – interactive diagrams



Acknowledgements

- NSERC (Canada)
- NSF (USA)
- NOAA
- ARDA

Workshop Part I

UML Class Diagrams - State of the Art in Layout Techniques

Holger Eichelberger
chair of computer science II
Würzburg University
Am Hubland, 97074 Würzburg, Germany
eichelberger@informatik.uni-wuerzburg.de

Jürgen Wolff von Gudenberg
wolff@informatik.uni-wuerzburg.de

Abstract

Even if the standard for specifying software, the Unified Modeling Language, is known in different versions to everybody, CASE tool vendors did not implement all basic features. Even with class diagrams, many features of the standard are ignored. Applying the layout algorithms of these CASE tools to the user defined diagrams, usually horrible results are produced, because state-of-the-art techniques in drawing these diagrams are not respected by the vendors, too.
In this paper we give an overview on the current UML tool implementations, the research in the field of drawing class diagrams automatically and the efforts in convincing the community of an agreement on basic aesthetical principles for UML class diagrams in order to simplify reading and understanding of standardized visualization of static aspects of software.

1. Introduction

In software engineering the Unified Modeling Language (*UML*) has advanced as the standard for graphically specifying static and dynamic aspects of software. In 2003 the Object Management Group (*OMG*) released the version 2.0 of the UML. The number of different diagram types increased from 9 to 13, new types of diagrams have been introduced and some of the older types have significantly increased in complexity. In [4] we compared the implementations of the UML features and the automatic layout facilities of 42 current computer aided software engineering (*CASE*) tools and agree to the informal statements of others, that most of the CASE tools have not reached the conformity of UML versions older than 1.3 Regarding the automatic layout features, the tools usually produce horrible results by transforming the layout and implicitely and accidentally changing the semantics of the complete diagram.
In the next section we summarize the results of the CASE tool overview, then we mention the set of basic aesthetic principles which we advocate as a set of basic rules to be respected by software engineers as well as tool vendors and finally we present the results of state-of-the-art graph drawing algorithms for class diagrams. References to other work are included in the individual sections.

2. UML Tools - an Overview

"For more then one decade vendors have under delivered the promises of object modeling technologies. As a result, object modeling tools are in disrepute in many development organizations."[10] "As for modeling tools, the author knows of none that fully implements the UML 1.1 semantics and notation (adopted three years ago), let alone one that completely or correctly implements the current UML 1.3 specification (which was adopted a year ago)" [10, october 2000]. As shown in [4] even in July 2002 the situation on UML conformity and implementation of layout algorithms was nearly the same.
For our test, we tried to use the diagram shown in figure 1 as input to the tools and then applied the automatic layout mechanisms if present. First problems arised while trying to define the test diagram within the tool:

1. Most tools are too implementation-specific. Model elements visualizing abstract concepts which cannot be directly realized in a programming language like association classes, higher associations, constraints and even comments are not present. Since most tools are not able to correctly produce code for the different responsibilities of associations, these tools should omit at least associations, too.

2. A lot of tools implement packaging mechanisms as logical view only. It is not possible to use classes within packages (extremely useful when visualizing coupling and cohesion or applying the facade design pattern). The alternative concept, the anchor edge, is

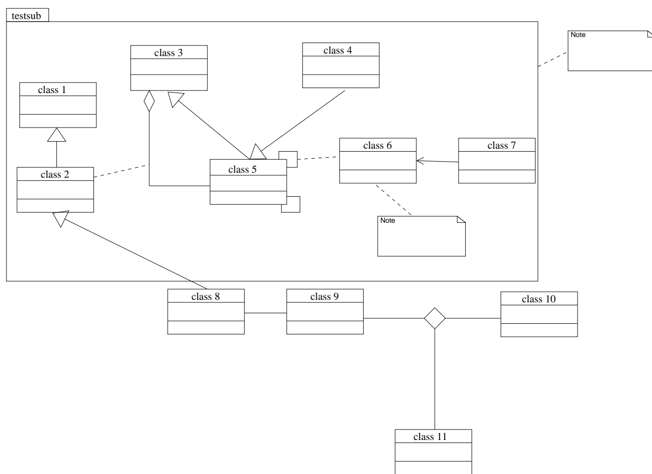


Figure 1. The test diagram in [4] which shows classes within a package, relations across package borders, two association classes in further relations, a ternary association, two nodes and two reflective associations.

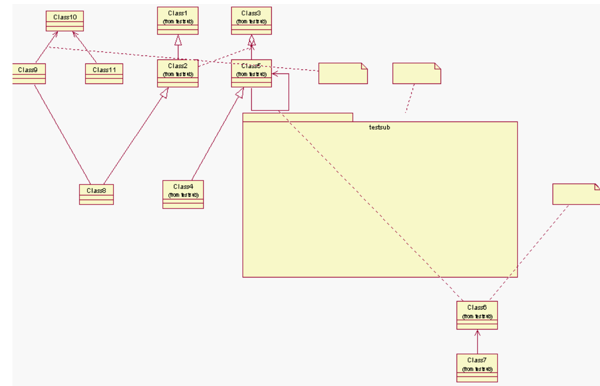


Figure 2. Rational AnalystStudio 2002.05.20

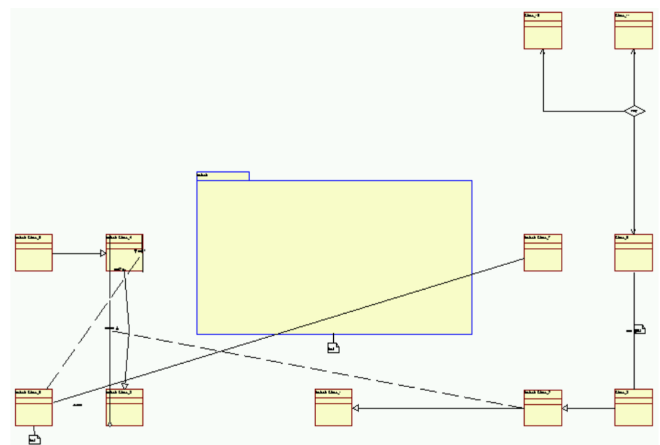


Figure 3. Popkin System Architect 8.5.16

usually not present. Other package-like elements like subsystems or models are not present in most tools, either.

3. Nearly each tool implements its own input philosophy - as a users wish to the vendors we propose the specification of a user interface standard for CASE tools in order to increase usability and to simplify the use of multiple tools.

A tool which claims to be conformant to the UML in any given version should realize the complete specification without any restrictions!
The result of applying the automatic layout mechanism on individual tools is depicted in the figures 2 to 5. Applying the layout mechanism twice or more times sometimes produced different results. Screenshots of the entire screen are shown in [4].

3. Rules for the Layout of a Class Diagram

Unfortunately most layout algorithms on class diagrams do not adhere to any aesthetic principles. Different surveys [13, 14, 15] on class diagrams have been published but most of them rely on aesthetical principles taken from graph drawing without respecting the underlying semantics. The latest results show that there is not a uniform user preference on the regarded aesthetic principles.
In [3, 6] we discussed basic issues for semantic based aesthetic criteria in order to provide intuitive rules for drawing class diagrams. The rules are validated by references to the UML specification, to HCI results and results from software engineering. Some of the rules are mentioned below in a compressed form:

1. Enforce hierarchy as the most appropriate ordering criterion for edges in a class diagram. Since software engineers are used to thinking hierarchically, containment, inheritance, realization, aggregation, composition and user defined hierarchies should be taken into account. Even the latest publications on other layout algorithms [6, 9] adhere to that principle.

2. Respect spatial relationships to encode coupling, cohesion and importance of parts of the diagram.

3. Visualize the natural clustering of nodes according to semantical reasons like containment, n-ary associations and patterns.

4. Avoid crossings and overlappings of model elements.

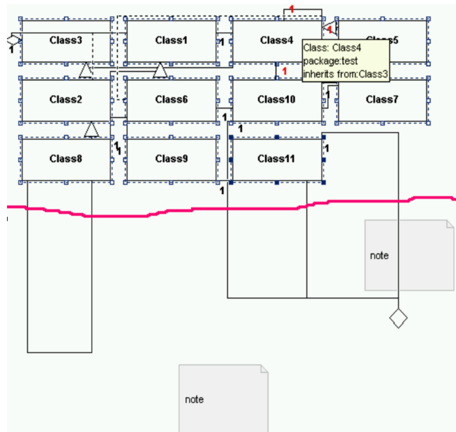


Figure 4. TNI OpenTool 3.2.15 (poor layout award) - part of the diagram was cutted of due to space limitations (red line)

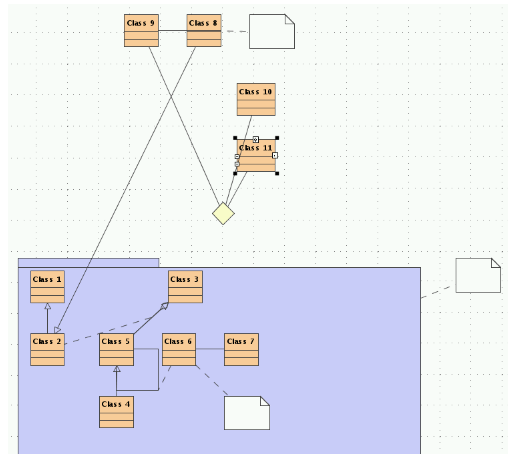


Figure 5. NoMagic MagicDrawUML 7.0 beta (best layout and UML conformity award)

5. Center position of selected nodes (n-ary associations, pattern nodes).

6. Respect the vicinity of association classes, notes and constraints.

7. Clearly assign adornments to edges and reflective associations to the connected classes.

8. With the minimum priority respect other graph drawing criteria.

Intuitively these rules lead to readable diagrams and therefore can reduce the cost of communication when interchanging software development diagrams. Additionally these rules can be used as definition of a measurement framework for the objective comparison of tool features and layout algorithms. Unfortunately validating these rules by user experiments is a hard task: high degree of freedom induced by the number of criteria, the need for qualified and experienced software engineers instead of UML-novice students as users to be questioned and low UML tool support so far since no standardized diagram exchange format for UML diagrams was defined.

4. Drawing a Class Diagram

For other UML diagrams like activity diagrams (flow layout) and state charts [2, 1] appropriate algorithms have been proposed but unfortunately these algorithms are usually not implemented in CASE tools so far. The large variety of model elements available for use in UML class diagrams are not respected by most of the algorithms proposed so far [6, 9, 8, 16, 17]. These algorithms mainly focus on classes, inheritance relations and association relations but not on nested package and class structures and more sophisticated model elements like association classes, higher associations or constraints.

The following listing is a brief description of our current (revised) approach. Detailed descriptions and relations between the algorithm and the aesthetic rules mentioned in section 3 can be found in [3, 5, 7].

1. Identify a pseudohierarchy by heuristics or by respecting a user defined hierarchy.

2. Perform a semantic ordering to release implicit dependencies between the sequence of definitions of the model elements in the input and the layout result.

3. Insert containment relations of model elements as hierarchical edges.

4. Compress association classes and their edge connector nodes into compound nodes.

5. Convert annotations and connected model elements to compound nodes.

6. Remove reflexive associations in order to simplify the implementation. Represent the edge information within the connected classes in order to be drawn as edges later on.

7. Transform the graph to an acyclic graph.

8. Guarantee a virtual root.

9. Calculate the ranking of the hierarchically connected nodes in one step, calculate the layer positions of only

non-hierarchically connected nodes in a second step. Optimize the layered structure of the graph for UML class diagram layout.

10. Calculate edge crossing minimization on hierarchical and non hierarchical edges by an incremental crossing reduction approach. Respect cluster and containment relations in this step.

11. Remove containment information.

12. Calculate the coordinates of nodes and edges. Contained model elements are treated in the same step in order to respect non-hierarchical edges.

13. Expand compound nodes for association classes.

14. Expand and layout notes.

Preserving the the mental map [11] is extremely important when iteratively changing class diagrams while analysis and design phase as well as in roundtrip-engineering. To implement incremental layout, phases for compressing and preserving the positions of unchanged nodes can be inserted at the beginning and the end of the algorithm. As

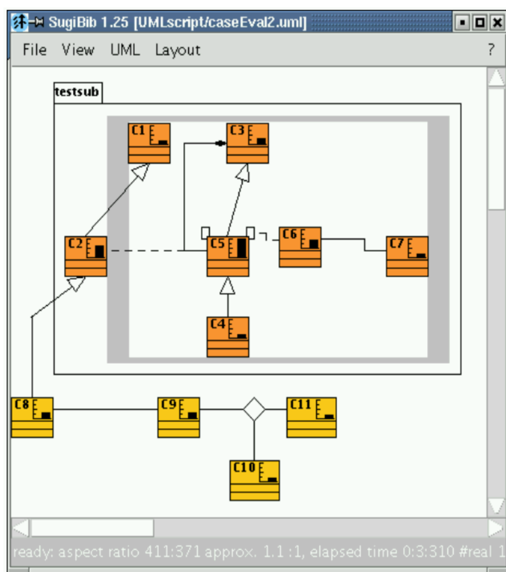


Figure 6. The test diagram from [4] drawn by SugiBib. Visualization of coupling and cohesion is enforced (the shaded area is a non-UML feature and drawn for demonstration purpose only), the relative complexity of classes is shown as a decorative stereotype. Notes are not implemented so far.

a proof of concept the algorithm has been implemented as

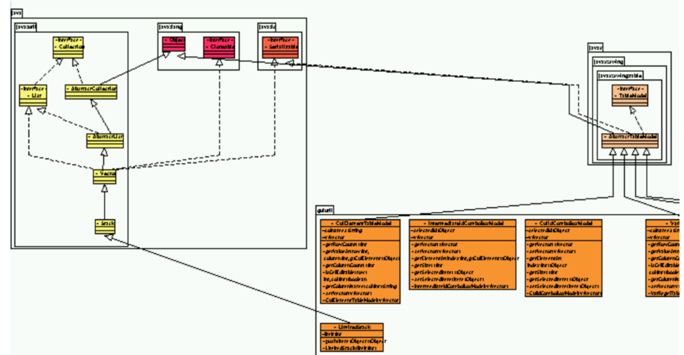


Figure 7. Part of the relations between different Java library packages and a simple graphical user interface implementation.

a prototypical framework written in pure Java. The figures 6 to 8 show different results produced by our algorithm. Current information on SugiBib can be obtained from `www.sugibib.de`.

4.1. Conclusions

In this paper we have shown that the current implementations of CASE tools neither implement an appropriate version of the UML nor provide layout algorithms which represent the state-of-the-art in drawing class diagrams. The first restricts the user to software models which are implementation-specific and far away from the desired level of abstraction. Defining an own subset of the UML restricts the usability and conformance to future standards like MDA [12]. The second requires more manual adjustments, requires more time and disables effective engineering techniques like reverse-engineering and roundtrip-engineering. Since designing layout algorithms is not one of the core competences of a CASE tool vendor, it is advisable to disable their individual algorithms (especially if the undo function is not fully functional) or to implement a warning message as long as more appropriate algorithms are implemented.

Since different algorithms may produce different drawings which might be nice to different individuals, a UML based standard for diagram layout and interchange aesthetics should be proposed. We have shown a subset of our proposal for aesthetic principles for class diagrams. As a consequence of applying these principles the readablility and understandability is enhanced.

Finally we have mentioned the problems of the other approaches to realize the automatic layout of class diagrams. Most of the other algorithms extend the topology-shape-metrics approach without a description on how to realize

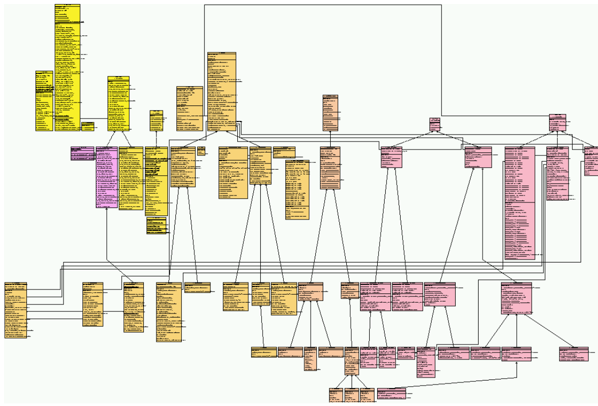


Figure 8. A class hierarchy (respecting inheritance relations). The nodes are colored and grouped according to package containment without showing the packages itselves.

the more complex situations which arise from using more sophisticated model elements. Sometimes other hierarchical or even force-directed approaches are mentioned in literature but usually only the layout of classes and simple relations is respected.

Since most tool vendors do not fully implement older versions of the UML it will be a long road for a complete realization of the new UML version 2.0 Since the complexity of most diagrams has increased, most of the traditional algorithms (only working on structure not on semantics) like flow-layout for activity diagrams are not appropriate any more. For class diagrams at least component classifiers (class-like model elements furtherly structured by classes or components) and the graphical representation for provided and required interfaces/components have to be respected by a layout algorithm. Since our algorithm is capable of working on nested and structured elements it can easily be updated. Even if it is known, that adding more constraints to an algorithm the runtime is increased and low-quality results (if even a result can be computed) is the risk we believe, that this can be respected by introducing additional criteria into the new edge crossing reduction algorithm. Additionally a slight update to our set of aesthetic principles is necessary.

References

[1] R. Castello, R. Mili, and I. G. Tollis. Automatic layout of statecharts. *Software – Practice and Experience*, 32(1):25–55, 2002.

[2] R. Castello, R. Mili, and I. G. Tollis. A framework for the static and interactive visualization of statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.

[3] H. Eichelberger. Aesthetics of class diagrams. In *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 23–31. IEEE, IEEE, 2002.

[4] H. Eichelberger. Evaluation-report on the layout facilities of UML tools. TR 298, Institut für Informatik, Universität Würzburg, jul 2002. Institut für Informatik, Universität Würzburg.

[5] H. Eichelberger. Sugibib. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, 9th International Symposium, GD '02*, volume 2265 of *Lecture Notes in Computer Science*, pages 467–468. Springer, Springer, 2002.

[6] H. Eichelberger. Nice class diagrams admit good design? In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 159–ff. ACM, ACM Press, 2003.

[7] H. Eichelberger and J. W. von Gudenberg. On the visualization of Java programs. In S. Diehl, editor, *Software Visualization, State-of-the-Art Survey*, volume 2269 of *Lecture Notes in Computer Science*, pages 295–306. Springer, Springer, 2002.

[8] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. Caesar automatic layout of UML class diagrams. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, 9th International Symposium, GD '02*, volume 2265 of *Lecture Notes in Computer Science*, pages 461–462. Springer, Springer, 2002.

[9] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188. ACM, ACM Press, 2003.

[10] C. Kobryn. Modeling components and frameworks with UML. *Communications of the ACM*, 43(10):31–38, 2000.

[11] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[12] OMG. Model driven architecture specifi cation. Version 1.0, May 2003 via `http://www.omg.org`.

[13] H. Purchase, J.-A. Allder, and D. Carrington. User preference of graph layout aesthetics: A UML study. In J. Marks, editor, *Graph Drawing - 8th Internation Symposium*, volume 1984 of *Lecture Notes in Computer Science*, pages 5–18. Springer, Springer, 2001.

[14] H. Purchase, J.-A. Allder, and D. Carrington. Graph layout aesthetics in UML diagrams: User preferences. *Journal of Graph Algorithms and Applications*, 6(3):255–279, 2002.

[15] H. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. *Proceedings of the Australian Symposium on Information Visualisation*, 9, 2001.

[16] D. Spinellis. On the declarative specifi cation of models. *IEEE Software*, 20(2):94–96, 2003. March/April.

[17] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfi les: Visualization and automatic layout of graphs. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Proc. Graph Drawing, 9th International Symposium, GD '02*, volume 2265 of *Lecture Notes in Computer Science*, pages 453–454. Springer, Springer, 2002.

Techniques for Reducing the Complexity of Object-Oriented Execution Traces *

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
{ahamou, tcl}@site.uottawa.ca

Abstract

Understanding the behavior of object-oriented systems is almost impossible by merely performing static analysis of the source code. Dynamic analysis approaches are better suited for this purpose. Run time information is typically represented in the form of execution traces that contain object interactions. However, traces can be very large and hard to comprehend. Visualization tools need to implement efficient filtering techniques to remove unnecessary data and present only information that adds value to the comprehension process. This paper addresses this issue by presenting different filtering techniques. These techniques are based on removing utility methods and the use of object-oriented concepts such as polymorphism and inheritance to hide low-level implementation details. We also experiment with 12 execution traces of an object-oriented system called WEKA and study the gain attained by these filtering techniques

Keywords:

Reverse engineering, program comprehension, dynamic analysis, object-oriented systems, and software visualization.

1. Introduction

Understanding object-oriented systems is a challenging task. Such systems are designed with the idea of interactions between objects in mind and in order to fully understand them we need to analyze these interactions rather than merely performing static analysis of the source code.

Information about the execution of an object-oriented system is typically represented in the form of traces of object interactions. Figure 1 shows an example of a very simple trace of method calls where specific objects are substituted by their class type – the term trace of class interactions would be more appropriate in this case. An alternative representation consists of labeling the edges with the messages and nodes with object identifiers or class names.

However, traces can be very large and hard to understand. This is due to the fact that important interactions are mixed with low-level implementation details. To overcome the size explosion problem, many visualization tools and techniques [1, 2, 4, 5] proceed by detecting repeated sequences of object interactions as distinct patterns of execution, which are then rendered in a way that helps a software analyst notice them easily and explore their content.

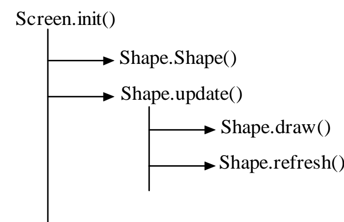


Figure 1. Trace of method calls. Objects are substituted with their class type

In this paper we present a set of techniques that aim at filtering the trace by removing unnecessary data with respect to program comprehension. We call this process: *Trace Compression.* For example, utility methods can be removed safely if the goal of the maintenance activity is to understand the overall design of the system, which in turn, can be very useful for design recovery.

Our approach consists of three main steps. First, we preprocess the trace by removing repeated interactions due to loops. Then we detect different types of utilities and remove them. Finally, we use object-oriented concepts, namely, polymorphism and inheritance to hide low-level implementation details.

We also present an experiment that we conducted on 12 execution traces of an object-oriented system called WEKA to estimate the compression gain attained by these techniques.

* This research is sponsored by NSERC

The rest of this paper is organized as follows; the next section discusses the size problem of the traces. In section 3, we present the compression techniques. In Section 4, we describe the experiment and discuss the results.

2. The Size Problem

Although traces can be very large, a closer analysis of their content shows that they contain many redundancies. From the comprehension perspective, a software engineer needs to understand a repeated sequence of calls only once and reuse this knowledge whenever it occurs. Therefore, a more accurate way of reasoning about the size problem of a trace should be based on analyzing distinct subtrees of calls instead of the number of lines. We refer to each distinct subtree as *a comprehension unit*.

Figure 2a. shows a trace *T* (the class and method names are represented with one letter to avoid cluttering) that contains 9 calls but only 6 comprehension units as shown in Figure 2b.

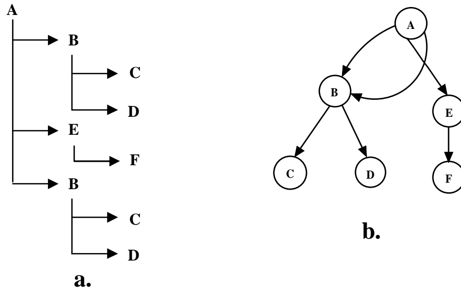


Figure 2. a. The trace T has 9 calls. b. an acyclic graph that represents the compact form of T and shows 6 comprehension units. Note that the crossing line represents the order of calls

In order to reduce the trace overhead problem, we need to find ways to group different subtrees as instances of the same comprehension units. The compression techniques[1] presented in this paper aim at accomplishing this.

There are different ways for measuring the compression gain. In this paper, we use a compression ratio and we define it as follows:

- Let T_1 be the original trace such as T has CU_1 comprehension units.
- Let T_2 be the resulting trace after compressing T_1 and CU_2 is the number of comprehension units of T_2
- The compression ratio R is:

$$R = 1 - CU_2/CU_1$$

This means that the higher the ratio the better the compression we get.

3. Trace Compression Techniques

3.1 Trace preprocessing

The first step consists of preprocessing the trace by removing contiguous repetitions of method calls or sequences of method calls that are due to loops. However, consider the trace of Figure 3, the two sequences rooted at *B* are not identical but can be considered similar from the comprehension point of view if the number of repetitions of *C* of the first subtree is ignored.

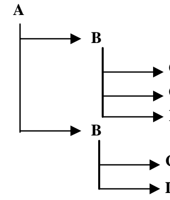


Figure 3. The subtrees rooted at B can be considered the same if number of repetitions is ignored

Therefore, we expand the preprocessing stage to consider two contiguous subtrees as the same even though the number of contiguous repetitions of their nodes is not exactly identical. This will result in a better compression ratio without a considerable loss of the trace content. Other matching criteria such as the ones presented by DePauw et al. [3] can also be used

3.2 Removing utilities:

The following are some criteria that can be used to rate the extent to which a method is a utility.

Constructors and destructors:

Constructors and destructors are used simply to create and delete objects, rather than to implement the core system operations. Therefore, it may be best to ignore them while trying to understand the behavior of a specific scenario. However, if the maintenance task involves such things as performance analysis or detecting memory leaks, then preserving constructors and destructors would be important.

Accessing methods:

Accessing methods are methods that return or modify directly the values of member variables. Accessing methods are used as a means to reinforce information hiding. Although, software engineers tend to follow the same naming convention for accessing methods, which consists of prefixing them with "get/set" followed with the name of the variable, it might be necessary to perform data flow analysis of the class that defines them to automatically detect them.

---

[1] We are not talking about data compression in the conventional sense (which results in unintelligible output), but rather, compression of the visible output so that it can be more easily understood.

Utility classes:

It is a common practice for software developers to create utility classes that can be used by other classes of the system. If those classes are already known by the software maintainer then she or he can remove them from the trace. There may also be a need to automatically detect such classes. For this purpose, the class dependency graph can be of assistance. Utility classes correspond usually to the graph nodes with a very large number of incoming edges and a very small number of outgoing edges [8].

3.3 Techniques based on OO concepts:

Polymorphic methods:

T. Lethbridge and R. Laganière define polymorphism as "a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes" [7]. The methods that implement the operation need to have the same name although they might have different signatures. Polymorphism is typically implemented using method overloading and inheritance.

Although the semantics of these methods should be the same, the execution trees that derive from them can be significantly different due to the way they are implemented. However, it is unlikely that the software maintainer will need to look inside the encapsulation to see the implementation details if only an abstract view of the design is needed, which leads to an opportunity to remove these details (since they merely implement an abstract operation). Removing such details can result in a significant compression. This concept applies to interfaces as well since an interface is considered as a pure abstract class.

4. Experiment

4.1 Description and settings

We experimented with an object-oriented system called WEKA version 3.0.6 [9]. WEKA is a tool that implements several data mining and machine learning algorithms including classification algorithms, association rules generators and clustering techniques. In addition to that, WEKA implements several filters that transform the input datasets in different ways such as adding or removing attributes, removing instances from the dataset and so on. WEKA is implemented in Java and contains around 160 classes and over 1680 methods. For more information about WEKA, please refer to [9].

We used our own instrumentation tool that is based on BIT [6] to add probes at each entry and exit point of the system public methods. Constructors are considered as regular methods. However, private methods are not instrumented to reduce the amount of processing time.

Traces are generated as the system runs and saved in a text file. Although WEKA comes with a GUI version, every WEKA algorithm and feature can be executed from the command line. We favored the command line approach over the GUI to avoid encumbering the traces with GUI components. A trace file contains the following information: Thread name; full class name (e.g. weak.core.Instance); method name and a nesting level that maintains the order of calls

We noticed that all WEKA algorithms use only one thread. Therefore, the thread name information is useless for this experiment. However, in case of a multi-threaded system, one needs to break the trace into different threads and apply the compression techniques to each of them.

Table 1. The traces used in this experiment

Trace	Algorithm or Filter	Description
1	Cobweb	Clustering algorithm
2	IBk	Classification algorithm
3	OneR	Classification algorithm
4	Decision Table	Classification algorithm
5	J48 (C4.5)	Classification algorithm
6	Apriori	Association algorithm
7	Attribute Filter	Filter
8	Add Attribute	Filter
9	Merge Two Values	Filter
10	Instance	Filter
11	Swap Attribute Values	Filter
12	Split Dataset	Filter

The main objective of this experiment is to estimate the gain attained by the compression techniques. We chose to analyze 12 execution traces of WEKA. Table 1. describes the algorithms and filters that correspond to each trace.

4.2 Experiment Design

The compression techniques presented in this paper can be combined in different ways. Each combination will eventually result in a different compression ratio. We narrow down all the possible results to the following:

- Initial information about the trace such as the number of lines, the number of comprehension units, etc.
- The gain attained after preprocessing the trace.
- The gain attained after removing constructors from the preprocessed trace
- The gain attained after removing accessing methods from the preprocessed trace. For this purpose, we noticed by inspecting the source code that WEKA follows the "set/get" naming style.
- The gain attained after removing utility classes from the preprocessed trace. For this purpose, we analyzed WEKA documentation to discover eventual utility classes. We found that WEKA contains a class called

Utils where many utility methods such as *doubleToString, eq, etc* are defined

- The gain attained after removing the details of polymorphic methods from the preprocessed trace. In this paper, we focus on overriding only. Overloaded methods that are defined in the same class are considered identical since we do not take into account the arguments list. However, methods that are overloaded in different classes are not considered in this paper for simplicity reasons.
- Finally, we also combine these techniques together to show the gain attained after removing all utilities (constructors, accessing methods…) and removing details of polymorphic methods from the resulting trace. However, this is not the only approach to combining. Future research can focus on other possibilities.

Table 2. summarizes the variables used to describe the results in a more precise way:

4.3 Results and discussion

Table 3 shows general information about the traces. Although some traces contain over 100 000 lines (e.g. Trace 6), we notice that they do not contain a lot of distinct methods (e.g. only 65 methods in trace 6). The number of comprehension units is also low. This means that there are many repetitions in the trace that are either due to loops or the presence of the same sequences of calls all over the trace. The preprocessing stage reduces considerably the size of most of the traces although Traces 4, 5 and 6 are still considerably large. We also notice that Trace 5 has a very large number of comprehension units, which might imply that it is the most complex trace. It is also interesting to see that ignoring repetitions when removing contiguous repetitions of sequences of calls results in a higher reduction of the number of comprehension units for large traces compared to small traces. For example, Trace 10 and 12 still keep the same number of comprehension units although the number of lines is considerably smaller after the preprocessing stage.

Table 4. shows the results of removing utilities and the call hierarchies that are derived from polymorphic calls. We notice that removing the constructors for large traces (Traces 1 to 6) results in a higher reduction compared to removing accessing methods. This is due to the fact that most of these methods were already removed during the preprocessing stage. Another reason is that, these traces use a large number of objects. On the other hand, small traces do not use a lot of objects and removing constructors might not be that important, which explains why removing accessing methods still gives a slightly better compression ratio. Removing the methods of the class *Utils* seems to give almost the same result for all the traces.

Table 2. Variables used to represent the results

Variable	Description
N_{init}	The number of calls of the initial trace
CU_{init}	The number of comprehension units of the initial trace
Classes	The number of distinct classes of the system that the initial trace contains
Methods	The number of distinct methods of the system that the initial trace contains
N_{prep}	The number of calls of after preprocessing the initial trace. Let us call the resulting trace T_{prep}
CU_{prep}	The number of comprehension units of T_{prep}
R_{prep}	Compression ratio = $1 - CU_{prep} / CU_{init}$
N_{const}	The size of the resulting trace after removing constructors from T_{prep} (preprocessed trace)
CU_{const}	The number of its comprehension units
R_{const}	$= 1 - CU_{const} / CU_{prep}$
N_{access}	The size of the resulting trace after removing accessing methods from T_{prep}
CU_{access}	The number of its comprehension units
R_{acces}	$= 1 - CU_{access} / CU_{prep}$
N_{util}	The size of the resulting trace after removing the methods of the class Utils from T_{prep}
CU_{util}	The number of its comprehension units
R_{util}	$= 1 - CU_{util} / CU_{prep}$
N_{poly}	The size of the resulting trace after removing the details of polymorphic methods
CU_{poly}	The number of its comprehension units
R_{poly}	$= 1 - CU_{poly} / CU_{prep}$
Poly_Meth	Number of polymorphic methods
$N_{cum-utils}$	The size of the resulting trace (let us call it T_{util}) after removing all utilities (constructors, accessing methods…).
$CU_{cum-utils}$	The number of its comprehension units
$R_{cum-utils}$	$= 1 - CU_{cum-utils} / CU_{prep}$
$N_{cum-poly}$	The size of the resulting trace after removing polymorphic calls details from T_{util}
$CU_{cum-poly}$	The number of its comprehension units
$R_{cum-poly}$	$= 1 - CU_{cum-poly} / CU_{prep}$

On the other hand, removing the details of polymorphic methods reduces considerably the size of Trace 1 but reduces its comprehension units by only 45.77% as shown in Table 4. The analysis of Trace 1 showed that the method buildClusterer() is the main cause behind this high reduction. WEKA implements two clustering algorithms and both of them consist of classes that override the method buildClusterer(). Building clusters might involve going through the dataset several times to find relationships between them. This usually generates very large hierarchies of calls, which explains the significant reduction when these details are hidden.

Table3: General information about the trace and the results of preprocessing them

Trace	Classes	Methods	N_{init}	CU_{init}	N_{prep}	CU_{prep}	R_{prep}
1	10	63	193121	108	6015	79	27%
2	12	92	37882	185	3719	113	39%
3	10	89	27554	223	4557	124	44%
4	19	150	154185	305	29576	224	27%
5	23	152	95118	469	25933	306	35%
6	9	65	156792	317	19810	127	60%
7	11	76	1902	83	281	83	0%
8	10	71	2534	80	351	80	0%
9	10	73	2245	84	752	83	1%
10	10	68	1248	73	247	73	0%
11	10	73	2256	83	374	82	1%
12	10	71	1398	79	289	79	0%

Similarly, Traces 3 and 5 represent two classification algorithms represented by two classes that override the buildClassifier() method. This method also generates large hierarchies of method calls. The results presented here go along with the idea of abstracting out the trace to extract high-level interactions. For example, a software engineer might only want to know that, at this point of time, a classifier or a clusterer is being built without having to go into the details.

Table 4: Removing utilities and details of polymorphic methods from the preprocessed traces

T	N_{const}	CU_{const}	R_{const}	N_{access}	CU_{access}	R_{access}	N_{util}	CU_{util}	R_{util}	N_{poly}	CU_{poly}	R_{poly}	Poly. Meth
1	5305	67	15.19%	6009	75	5.06%	6008	73	7.59%	289	46	41.77%	4
2	3329	91	19.47%	3409	99	12.39%	3599	104	7.96%	1973	95	15.93%	2
3	3898	106	14.52%	4260	115	7.26%	4449	116	6.45%	1253	80	35.48%	3
4	27039	183	18.30%	28068	186	16.96%	27759	213	4.91%	20916	158	29.46%	5
5	21408	275	10.13%	25124	290	5.23%	24297	286	6.54%	1633	99	67.65%	5
6	18880	113	11.02%	19610	116	8.66%	19771	119	6.30%	19810	127	0.00%	0
7	228	70	15.66%	252	65	21.69%	267	79	4.82%	227	68	18.07%	3
8	299	68	15.00%	329	68	15.00%	332	75	6.25%	285	65	18.75%	3
9	674	70	15.66%	718	68	18.07%	721	78	6.02%	626	66	20.48%	3
10	208	61	16.44%	219	52	28.77%	232	69	5.48%	192	61	16.44%	3
11	316	69	15.85%	350	68	17.07%	355	77	6.10%	274	66	19.51%	3
12	242	67	15.19%	262	63	20.25%	271	74	6.33%	245	72	8.86%	3

Trace 4 represents an algorithm that creates a decision table and generates classifiers out of it. Although some polymorphic methods were found, the number of lines is still very large; this might be due to the complexity of this algorithm.

It is interesting to notice that the number of overridden methods that appear in the traces as indicated by the variable *Poly_Meth* is low. Trace 6, for example, does not contain any polymorphic method. In fact, Trace 6 corresponds to the only association algorithm that is implemented in WEKA, called Apriori. The class Apriori is created to build association rules which are an important part of this algorithm. However, this class does not have a superclass, which explains why polymorphism was not used here. This result is very interesting because it shows the limitations of using polymorphism (based on overriding) to hide details and requires investigating other means for hiding these details. Perhaps, a more general definition of utility methods can lead the way.

Finally, Table 5 shows the cumulative results of removing utility methods and removing polymorphic methods from the resulting trace. The table shows higher

compression ratios in terms of the number of comprehension units and very high reduction of the number of lines. The compression ratio of all the traces has increased expect for Trace 6 because it does not have polymorphic methods. Different combination of these techniques will lead to different compression ratios. However, future work needs to involve the users in order to discover which combinations suit them the best.

Table 5: Cumulative results

T	$N_{cum-utils}$	$CU_{cum-utils}$	$R_{cum-utils}$	$N_{cum-poly}$	$CU_{cum-poly}$	$R_{cum-poly}$
1	5296	59	25.32%	202	30	62.03%
2	2925	71	37.17%	1320	61	46.02%
3	3514	92	25.81%	734	52	58.06%
4	23770	138	38.39%	17048	96	57.14%
5	19047	247	19.28%	1006	68	77.78%
6	18645	96	24.41%	18645	96	24.41%
7	190	48	42.17%	173	44	46.99%
8	264	53	33.75%	227	45	43.75%
9	615	52	37.35%	540	44	46.99%
10	173	38	47.95%	132	32	56.16%
11	279	52	36.59%	216	44	46.34%
12	205	48	39.24%	180	44	44.30%

Conclusions and future directions

Dynamic analysis is very useful for understanding the behavior of object-oriented systems. The analysis of traces of object interactions can bridge the gap between low level implementation details and high level domain concepts, if effective filtering techniques exist. Interactions between objects are typically depicted in traces of method calls. In this paper, we presented many techniques that can help hide implementation details and reveal only important interactions. The experiment showed several results that can be used by tool builders to improve their tools. First, we showed that the number of calls in the trace is not the major factor of complexity. Traces can be very large but have few comprehension units. Another interesting result is that traces always need to be preprocessed. Although, using these compression techniques separately might result in a good compression ratio, they work best when combined.

Future work should focus on several areas such as considering a broader definition of utility methods. We also need to experiment with many other software systems to understand how to combine the compression techniques in order to extract the most important interactions that many software designers agree about.

References

[1] W. De Pauw, D. Kimelman, and J. Vlissides, "Modeling Object-Oriented Program Execution", *In Proc. 8th European Conference on Object-Oriented Programming*, Bologna, Italy, 1994, pp. 163-182.,

[2] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the Behavior of Object-Oriented Systems", *In Proc. 9th Conference on Object-Oriented Programming Systems, Languages, an Applications*, Portland, Oregon, USA, Oct. 1994, pp. 326-337

[3] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization.", In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems, COOTS, 1998,* pp. 219-234

[4] D. Jerding, S. Rugaber, "Using Visualization for Architecture Localization and Extraction." *In Proc. 4'th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, Oct. 1997

[5] D. Jerding, J. Stasko, T. Ball, "Visualizing Interactions in Program Executions", *In Proc. of the International Conference on Software Engineering (ICSE)*, 1997, pp. 360-370

[6] H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes", *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, 1997, pp. 73-82

[7] T. C. Lethbridge, R. Laganière, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw Hill, 2001

[8] H. A. Müller, M. A. Orgun, S. Tilley, J. Uhl, "A Reverse Engineering Approach To Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice, Vol 5, No 4,* December 1993, pp. 181-204

[9] I. H. Witten, E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999

ADG: Annotated Dependency Graphs for Software Understanding

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT
Dependency graphs such as call and data usage graphs are often used to study software systems and perform impact analysis during maintenance activities. These graphs show the present structure of the software system ($e.g.$ In a compiler, an $Optimizer$ function calling a $Parser$ function). They fail to reveal details about the structure of the system that are needed to gain a better understanding. For example, traditional call graphs cannot give the rationale behind an $Optimizer$ function calling $Parser$ function.

In this position paper, we advocate a new view on dependency graphs – Annotated Dependency Graphs (ADG). ADG can assist maintainers understand better the current structure of large software systems. We show an example of using an ADG to study $Postgres$, a large DBMS open source software system.

1 INTRODUCTION

To aid in software understanding tasks, documentation is used to narrate different aspects in the life cycle of a software system. Unfortunately software developers are not interested in documenting their work. Documentation rarely exists. If it does it is usually incomplete, inaccurate, and out of date. Faced with the lack of sufficient documentation, developers choose alternative understanding strategies such as searching or browsing the source code. The source code in many cases represents the definitive source of accurate information about the system [11]. Developers search the code using tools such as `grep`. They browse the code using simple text editors or cross-reference code browsers such as LXR, which permit jumping between variables/functions usage and variables/functions declarations while browsing the source files.

Dependency graphs have been proposed and used in many studies and maintenance activities to assist developers in understanding large software systems before they embark on modifying them to meet new requirements or to repair faults. Call graphs and data usage graphs are the most commonly used dependency graphs.

The rationale behind the existence of dependencies between two nodes in a dependency graph are usually based on domain and system knowledge. For example, based on our knowledge of the reference architecture of a compiler, we can reason about the rationale behind the dependencies shown in the graphs [10]. For domains that are not well understood that may not be clear and may prove to be a challenging and daunting task. Moreover, for well understood architectures such as compilers, we may find unexpected dependencies that indicate, for example, that an $Optimizer$ function depends on a $Parser$ function. As a maintainer of such a system, the rationale behind such unexpected dependency is not clear - Are there valid reasons for such dependency? Or was it due to laziness or ignorance of the developer that introduced the dependency?

Much of the knowledge about the design of a system, its major changes over the years and its troublesome subsystems live only in the brains of its developers. Such live knowledge is sometimes called *wet-ware*. When new developers join a team, mentoring by senior members and informal interviews are used to give them a better understanding of the system. Leveraging this knowledge may not always be possible as the software may have been bought from another company, its maintenance outsourced, or its senior developers are no longer part of the company. Thus, answering questions about unexpected dependencies and other discoveries as developers study dependency graphs becomes a challenging and time consuming task. Traditional dependency graph are only capable of giving us a current view of the software system without details about the rationale, the history, or the individuals behind the dependency relations.

In this paper, we propose to extend dependency graphs – Annotated Dependency Graphs (ADG) – to attach more details, in an attempt to assist developers in understanding and studying software systems. In an ideal world, if each developer attached a sticky note to each added dependency to record their name, the rationale behind the addition or removal of the dependency then the job of the maintainer will be much easier. In the fast paced world of software development with tight schedules and short time to market, this is neither possible nor practical. Thus in addition to proposing these extended dependency graphs, we present a technique to build such graphs automatically without any input from the developers of the system.

Organization of Paper

The paper is organized as follows. Section 2 highlights sev-

eral problems associated with traditional dependency graphs and proposes Annotated Dependency Graphs (ADG) to address these shortcomings. Section 3 gives an overview of how to build an ADG. Section 4 presents a short case study of an ADG for *Postgres*, a large open source database management system (DBMS). Section 5 describes related work. Finally section 6 draws conclusions from our work and proposes future directions.

2 SHORTCOMINGS OF DEPENDENCY GRAPHS

As maintainers prepare to modify a software system to add features or repair bugs, they start off by examining any available documentation, and consulting senior developers. Then they browse the source code and use tools to generate dependency graphs such as call and data usage graphs. Following these steps in an iterative manner, maintainers start gaining a better understanding of the software system. Using their newly acquired understanding, they form an internal cognitive model of the software [12]. Dependency graphs assist maintainers in gradually piecing together the software puzzle. Unfortunately, dependency graphs fall short in the following areas:

1. Rationale: They do not indicate the reasons behind the introduction or removal of dependencies between source code entities.
2. Time: They do not indicate how long a dependency has existed for or how long ago has it been removed.
3. Inter-dependency patterns: They fail to show patterns of dependencies. For example, in a compiler system it is hard to deduce from a call graph that once a function depends on the $populate_symbol_table()$ function, it will also depend on $symbol_table_entry$ data type.
4. Creator: They fail to show the name of the developer that introduced the dependency.

In the following subsections, we elaborate on the benefits of each area for software understanding.

Rationale

In a compiler, when a dependency between an $Optimizer$ function and a $Parser$ function is discovered, the maintainer puzzled by the unexpected dependency can contact the senior developer to get a better understanding of the rationale behind the introduction of such a dependency. The senior developer may be too busy or may not recall the rationale. Furthermore the developer who introduced the dependency may no longer work on the software system. Then the maintainer has to examine the source code closer and spend hours trying to understand the rationale behind such unexpected dependency. In some cases the added dependency may be justified due to, for example, optimizations or code reuse; or not justified due to, developer ignorance, or pressure to market.

Another popular usage of dependency graphs is the discovery of dead code. Dead code is code which no other entities in the software system depends on. Again dependency graphs are able to locate the dead code such as unused functions and unused data types, but fail to indicate the reason for the death of the code. A maintainer would like to know if the dead code has been replaced by an optimized or better piece of code capable of performing the same functionality, replaced by a more general piece of code that encourages more reuse, or even decommissioned as the system no longer supports the functionality offered by the dead code.

These two aforementioned examples are some of the many situations where the $Rationale$ for the appearance or disappearance of dependencies is essential in aiding maintainers of large software systems. Unfortunately, attaching the $Rationale$ for each dependency would require many resources and is time consuming. Consequently an automated technique to annotate each dependency with its rationale would be very beneficial.

Time

Current dependency graphs only provide a single current view of the software system. As pointed out in the previous section this prevents the dependency graph from helping developers understand the evolution of dependencies in the software system. Determining the evolution of dependencies for a dead piece of code is a good example. For example, the developer may want to know whether the dead function was introduced in the previous release to go around a bug and the bug is now fixed yet the functions isn't removed or whether the dead function has been around for many releases.

Inter-dependency patterns

The ability to determine that adding a dependency to one entity entails adding dependencies to other entities is a very valuable information that can be used in building tools to assist developers in maintaining large complex software systems. This knowledge would help developers guide developers to other entities in the source code that may require modifications.

Creator

In some cases, the creator of a dependency may pose a concern. The creator is a good indicator of the validity and trustworthiness of unexpected dependencies. For example an unexpected dependency created by a developer that was junior when the dependency was created is a strong sign that the dependency may be an invalid one.

3 BUILDING ANNOTATED DEPENDENCY GRAPHS

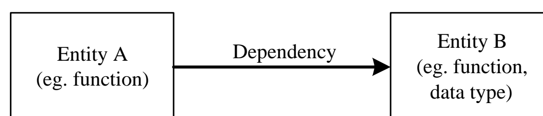


Figure 1: Schema for a Traditional Dependency Graph

To overcome the shortcomings highlighted in the previous sections, we propose Annotated Dependency graph (ADG). Whereas a traditional dependency graph would consist of entities and edges, as shown in Figure 1, an ADG would have several attributes attached to the nodes and the edges. Figure 2 shows the attributes that are attached. These attributes address shortcomings of traditional dependency graphs.

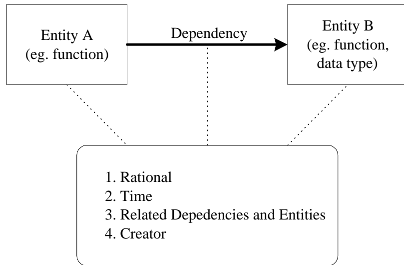


Figure 2: Schema for a Annotated Dependency Graph

Although, The ADG can be created manually by developers as they update the source code, this is not a practical solution for the following reasons:

1. It requires developers of large established software projects to go through a traditional dependency graph of the software system and try to populate the ADG. This is a time consuming and erroneous task. In many cases the developer may no longer recall the reasons for the dependencies and in most cases won't recall the details for the other attributes in an ADG.
2. Another alternative would be to use the ADG at the start of a new project and make sure developers always annotate any new or removed dependencies. Again this is extra work which most developers would not be interested in doing.

Instead of building the ADG manually, we chose to use the change records stored in the source control repository such as RCS [13] or CVS [3, 5].The repository contains details about the development history of each file in the software system. The repository stores the creation date of the file, its initial content and a record of each modification done to the file. A *modification record* stores the date of the modification, the name of the developer that performed the changes, the line numbers that were changed, the actual lines of code that were added or removed, and a detailed message entered by the developer explaining the rationale behind the change.

Source control systems store the details of the modification at the line level of a file which is not sufficient to build the ADG which has as functions and data types as nodes. Thus, we first need to preprocess the modification records to map the changes to the appropriate source code entities (*i.e.* functions or data types). Then we build the Annotated Dependency Graph and annotate it with details from the modification records such as time, and rationale. Due to size limitations, we will not discuss the details of the ADG building as it is detailed elsewhere [7].

4 CASE STUDY

To validate the usefulness of our approach we show a small case study on *Postgres*. *Postgres* is a sophisticated open-source Object-Relational DBMS supporting almost all SQL constructs. Its development started in 1986 at the University of California at Berkeley as a research prototype. Since then it has become an open source software with a globally distributed development team. It is being developed by a community of companies and people co-operating to drive the development of one the world's most advanced Open Source database software (DBMS).

In our case study we build an ADG using data beginning with 1996 when Postgres became an open source project. Building the ADG for Postgres requires 2 hours and 30 mins on a 1.8 Pentium 4 CPU. Luckily building the ADG needs to be done only once, then we can use it to generate graphs as required. All graphs shown in this section have been generated in under 5 seconds once the ADG was generated.

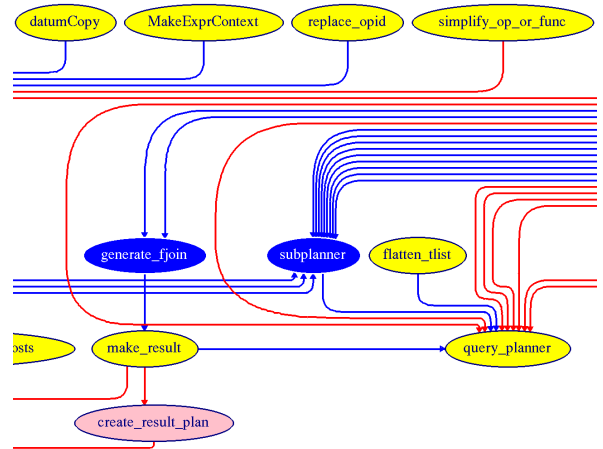


Figure 4: Zoomed-In Low-Level (*Function*) Call Graph for Postgres Optimizations

Using an ADG, we can generate a call graph similar to traditional call graphs generated by parsing the latest source code. Instead we chose to generate a more interesting call graph that showcases the benefits of using an ADG. Figure 3 shows an example of such graph. The displayed call graph shows changes to the call graph that

- occurred in the last month,
- and which were due to optimizations work done to speed up the database.

Each oval represents a function. Blue ovals indicate functions that have been removed from the source code, pink ovals indicate functions that have been added to the source code, and yellow ovals indicate functions that been modi-

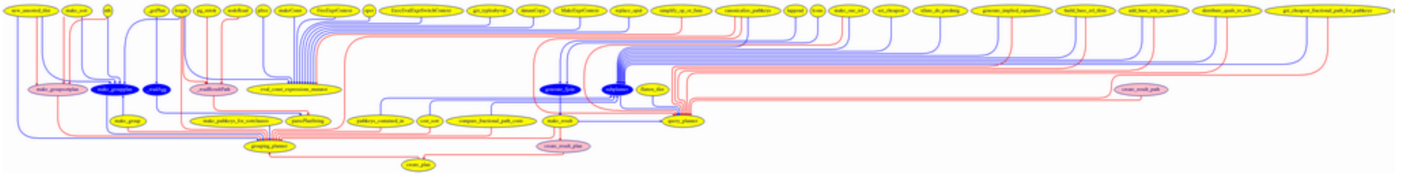


Figure 3: Low-Level ($Function$) Call Graph for Postgres Optimizations

fied in the last month. Also, blue arrows indicate function calls that have been removed and red arrows indicate function calls that have been added in the last month.

The graph shown in Figure 3 is too small to distinguish its various details. In Figure 4, we show a small zoomed-in section of the larger graph to give a better idea of the changes. A developer visualizing the call graph can focus on their area of interest using their visualization software.

Alternatively, we can lift the details of the graph from the function level to the subsystem level [6]. For each function in the call graph we locate the file that defines it, then the subdirectory in which that file resides. We use the subdirectory as the node in that lifted visualization instead of the function. The lifted visualization is shown in Figure 5 where each node represents a subdirectory. The new visualization is much clearer but does not have as much details. If more details are required then the function level visualization can be used.

5 RELATED WORK

The work presented in this paper addresses two main streams of research in particular, visualizing the evolution of source code and locating features in the source code.

Visualizating Evolution

In [8] and [14], two approaches are presented to tackle the issue of visualizing software structural changes. Both approaches are based on studying the changes between releases on a software system instead of basing their study on changes that occurred in the source control system, which are more granular and more detailed. They are only capable of comparing changes from one release to the next instead of being able to compare on a calendar basis or even comparing changes relative to other changes (such as examining changes that occurred before or after a particular change). Furthermore, they do not provide techniques to filter the changes and categorize them according to their rationale.

Locating Feature in Source Code

Chen $et~al.$ have shown that comments associated with source code modifications provide a rich and accurate indexing for source code when developers need to locate source code lines associated with a particular feature [2]. We extend their approach and map changes at the source line level to changes in the source code entities, such as functions and data structures. Furthermore, we map the changes to dependencies between the source code entities.

Murphy $et~al.$ argued the need to attach design rationale and concerns to the source code [1, 9]. They presented approaches and tools to assist developers in specifying and attaching rationale to the appropriate source code entities. The processes specified in their work is a manual and labor intensive process, whereas our approach uses the source code comments and source control modification comments to automatically build a similar structure to assist developers in maintaining large code bases. Moreover as our approach is automated, developers do not need to worry about maintaining the outcome of the process in addition to maintaining their source code, a challenge faced by the aforementioned processes.

Finally, Eisenbarth $et~al.$ presented an approach to locate features in the source code based on the integration of static and dynamic dependencies graphs [4]. Their approach uses a set of test cases to exercise features in the source code, then the static and dynamic call graphs for the specific test are analyzed to locate areas in the code that implement the feature. Whereas their approach uses a combination of static and dynamic source code and a suite of test cases, we only use the source code and the source code repository to locate features. Our approach will only locate features if they were specified in some comment in the source code or the source control system throughout the development history of the project. Thus in some cases using a dynamic analysis such as proposed by Eisenbarth $et~al.$ will be of great value and will complement our work.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented a new view on dependency graphs – Annotated Dependency Graphs (ADG). ADG represents a family of graphs which can assist maintainers as they work on gaining a better understanding of large software system. An ADG provides maintainers with the ability to study dependencies between the software entities and limit the dependencies to various criteria such as to a specific period in time, a specific change reasons, or even a specific developer.

In future work, we plan on using ADG to extract aspects from the source code and to build wizards that assist developers by suggesting source code entities that need to be modified once an entity has been modified based on the comodification history stored in the ADG.

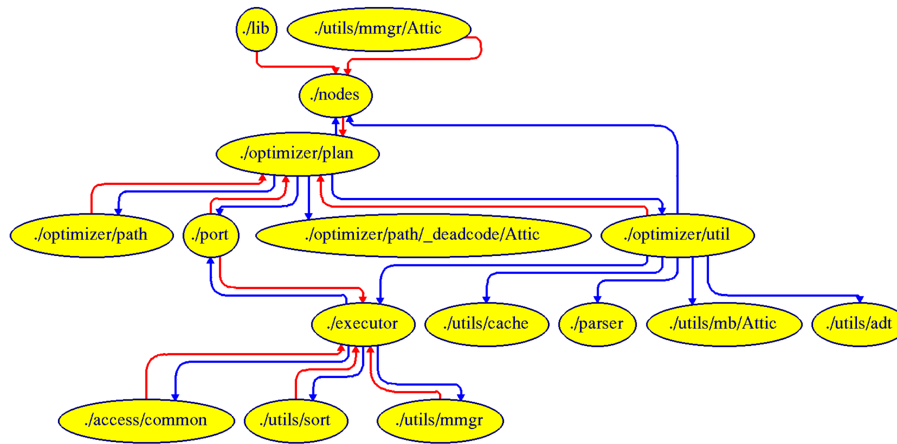


Figure 5: High-Level (*Subsystem*) Call Graph for Postgres Optimizations

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the significant contributions from the members of the open source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted us in understanding and acquiring these valuable repositories.

The figures shown in this paper were generated using the aiSee graph layout software.

REFERENCES

[1] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *IEEE 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[2] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code Using CVS Comments. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, 2001.

[3] CVS - Concurrent Versions System. Available online at <`http://www.cvshome.org`>

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions Software Engineering*, 29(3):195–209, Mar. 2003.

[5] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.

[6] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[7] A. E. Hassan and R. C. Holt. Understanding Change Propagation in Software Systems. In *Submitted for Publication*, 2003.

[8] R. C. Holt and J. Y. Pak. GASE: visualizing software evolution-in-the-large. In *Working Conference on Reverse Engineering*, pages 163–, 1996.

[9] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[10] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ., USA, 1996.

[11] S. E. Sim. Supporting Multiple Program Comprehension Strategies During Software Maintenance. Master's thesis, University of Toronto, 1998. Available online at <`http://www.cs.utoronto.ca/~simsuz/msc.html`>

[12] 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. In *International Workshop on Program Comprehension*, pages 17–28, 1997.

[13] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[14] Q. Tu and M. Godfrey. An Integrated Approach for Studying Software Architectural Evolution. In *Workshop on Program Comprehension (IWPC2002)*, Paris, France, June 2002.

Exploring the Many Architectures of a Very Large Component-based Software

Jean-Marie Favre, R. Sanlaville, J. Estublier

Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr/~jmfavre

Abstract

This paper describes the OMVT, an exploration tool specifically designed to explore the architecture of CATIA, a multi-million LOC software based on a component technology. This software is developed concurrently by more than 1000 software engineers. It can be dynamically extended and changed by third parties without any recompilation. Many techniques are used to deal with these very strong requirements on software architecture. Architectural concepts are however implicit in the source code or are represented by means of very low level techniques. The OMVT enables to cope with this problem by providing stakeholders the architectural views they need at the appropriate level of abstraction. Though the views presented are specific to Dassault Systèmes, the meta-model driven approach we took can be applied in other contexts.

1. Architecture at Dassault Systèmes

Dassault Systèmes (DS) is the CAD/CAM world leader and is one of the largest software editors in Europe. CATIA is one of its best-known software product. In the mid 90s, when DS initiated the development of CATIA V5, it was rapidly discovered that OO technology has serious limitations when developing very large scale software. C++ alone did not satisfy DS' strong requirements. As a result DS developed a proprietary component technology called the OM. DS is indeed with Microsoft one of the pioneers in componen-based software engineering.

All concepts provided by the OM, are implemented in terms of C++ entities or by means of other low level techniques. The mapping from architectural concepts to implementation is not one to one. For instance the realization of a single OM entity can produce a very large set of C++ entities. Moreover, for a given conceptual entity there are many implementation choices: to improve performance and address other non-functional requirements, DS designed and tested a wide range of realization techniques. All these techniques allow to build very efficient component-based software. But at the same time developing and maintaining large amount of components is quite difficult (CATIA is based on more than 60000 classes and about 8000 OM components). A major issue was that the architectural level was implicit and that software engineers had no tool to visualize the component they develop. The problem was even more accute since many people in different teams and sometimes in different companies collaborated to the development of a single component by adding extensions. What was missing was an architectural viewpoint suited to this specific component technology. The collaboration between the ADELE team and Dassault Systèmes lasted 7 years and during this period various other kinds of architecture were identified as well. This includes for instance the physical architecture, but also the collaborative architecture and the commercial architecture[1]. It is now widely recognized that the notion of software architecture greatly depends on the perspective considered and the stakeholders needs [2]. To formalize the various stakeholder needs and the many architectural concepts used within DS, we took a meta-model driven approach for architectural reconstruction [3]. According to the terminology defined by the IEEE standard [2] each *viewpoint* is a subset of the global meta-model, while each *view* is an instance of a viewpoint for a particular portion of the software considered.

2. OMVT: a specific exploration tool

In parallel with the definition of the architectural meta-model, we provided a graphical notation for the logical architecture. This notation was cautiously designed to be compatible with existing habits used internally during informal communications [4]. This notation provided the syntax to express architetural views on CATIA. Thanks to a reverse engineering process, architectural views are automatically extracted from CATIA source code as well as many other sources of information.

The need to use many sources of information in recovering the architecture of component-based software is also described in [5]. While the tool described in [5] was applied on a toy example, the OMVT was successfully

applied on the whole CATIA software which is made of more than 6 millions LOC. Though the OMVT was initially developped to explore the logical architecture and focused on DS proprietary component technology, support for many other architectural viewpoints were later added to this tool in order to explore the many other facets of software architecture at Dassault Systèmes. In total 27 viewpoints were defined to support the specific needs of various stakeholders. Some example of views derived from some viewpoints are represented in the figure below.

3. Scenario

Describing the whole features of the OMVT is impossible here, in particular because some features relie on proprietary architectural concepts such as *medias*, *solutions*, or *frameworks*. The figure below represents a typical OM component displaying some OM interfaces (1). This component is based on one "base implementation" (2) and other "extensions" (rectangles). Contrarily with the COM technology, component inheritance is supported as depicted on the top of window (1). The view depicted in window (3) provides more information about the implementation technique actually used to realize the component. For instance it is possible to distinguish extensions from single to multiple instanciation (3). As shown in (5) contextual pull-down menus are available for each entity displayed to further continue the exploration. A "troubleshooter" viewpoint was implemented. to automatically detect anti-patterns that could potentially lead to errors. In window (6) icons and colors indicate possible inconsistencies. A single click on the warning icon (7) opens window (8). Similarly a click on (9) displays window (10). These windows display only the subgraph of the component graph that reveal the existence of the anti-pattern. Inheritance relationships are shrinked to improve the reading of the figures and increase the usability for the stakeholders directly interested in correcting the error. Window (11) depicts a complement of information that improve the quality of the diagnosis. Note that all the views mentionned above correspond to the logical architecture. The stakeholder can switch very easily to alternative viewpoints on other kinds of architecture. For instance, a view on the physical architecture for the same component is displayed in window (12). This view shows that this simple component is actually implemented by more than 80 CPP classes spread in more 20 DLLs contained in more than one dozen of distinct frameworks.

4. References

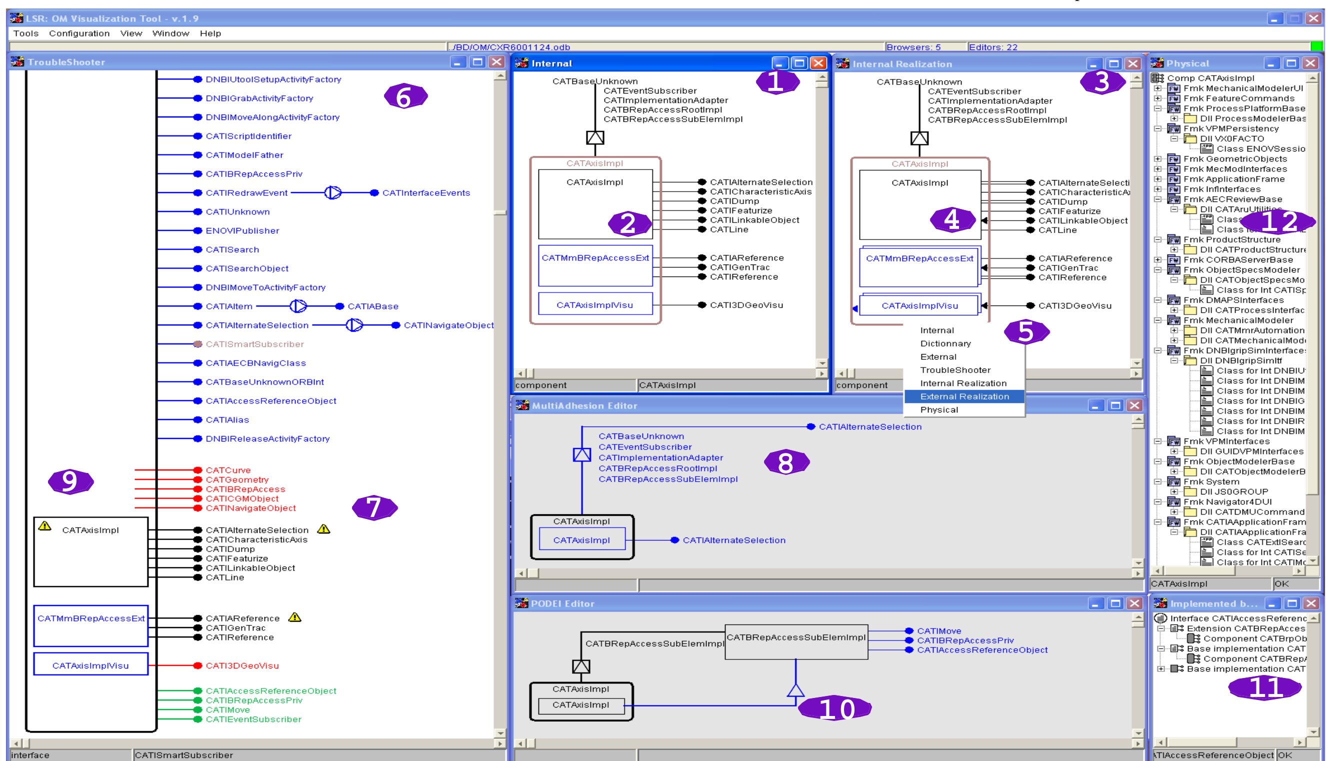
[1] J. Estublier, J.M. Favre, Y. Ledru, R. Sanlaville, "Architectural facts in the concurrent development of a Very Large Software", submitted to IEEE Software

[2] IEEE Architecture Working Group. "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems" . IEEE Std 1471-2000, October 2000.

[3] J.M. Favre, "Meta-Model Driven Reverse Engineering", submitted to WCRE 2003

[4] R. Sanlaville, "Software Architecture: An Industrial Case Study within Dassault Systèmes", PhD dissertation in french, Univeristy of Grenoble, 2002

[5] M. Pinzger, J. Oberleitner, H. Gall, "Analyzing and Understanding Architectural Characteristics of COM+ Components", IWPC 2003



AutoCode: Using Memex-like Trails to Improve Program Comprehension

Richard Wheeldon, Steve Counsell and Kevin Keenoy
Department of Computer Science
Birkbeck College, University of London
London WC1E 7HX, U.K.
{richard,steve,kevin}@dcs.bbk.ac.uk

Abstract

This paper presents AutoCode - a system for identifying "trails" of classes in Java programs. These trails are computed with regard to five coupling relationships (Aggregation, Inheritance, Interface, Parameter and Return Type) and are presented in a Web-based interface.

1 Introduction

In his seminal paper "As We May Think" [1], Vannevar Bush suggested a future machine called a "memex". In doing so, he introduced the world to the concept of linked documents and of the *trail* - a sequence of linked pages. The concept of trails is well established in the hypertext community and many systems have been built which support their construction [3].

Previous work has described a *navigation engine* for automatically constructing trails as a means of assisting users browsing Web sites [5]. This navigation engine was further used to provide search and navigation facilities for Javadoc program documentation. If JavaDoc-style program documention, derived from source code, can be indexed, it seems logical that the source code itself should be indexed also.

We have developed a new tool called AutoCode based upon the navigation engine design. AutoCode provides full-text indexing of Java source code and uses a probabilistic best-first algorithm to identify trails in graphs of coupling-type relationships.

2 Trails on Java Code

Classes and objects in OO systems do not work in isolation. They are connected to each other by various dependencies. The Java language connects classes together via five coupling relationships - Aggregation, Inheritance, Interface, Parameter and Return Type [4]. Each of these coupling relations can be used to construct a graph of dependencies. An illustration of how these graphs can be derived from Java source code can be seen in Figure 1. AutoCode constructs trails on each of these five graphs and presents them in a Web-based interface.

The *NavSearch* user interface used to present the trails (Figure 2) has three main elements. At the top is a *navigation tool bar* comprising a trail of classes considered most relevant (the "best trail"). On the left is a *navigation tree window* showing all the trails. Whenever the mouse pointer moves over these trails, a small pop-up appears which shows metadata and an extract. The rest of the display is dedicated to showing the source code of the selected class. The original Web-site search interface on which it was based was proven to be highly effective at allowing users to complete information seeking tasks [2]. It is hoped the same will apply to the AutoCode interface, a demonstration of which is available at http://nzone.dcs.bbk.ac.uk/.

Each trail is colour-coded according to the type of coupling involved. This coupling type is also shown in the pop-up for each class. Green trails denote parameter type references, cyan trails denote return-type references, gold trails show interface extensions, purple trails shows chains of aggregation links and orange trails show inheritance relationships from subclass to superclass.

Figure 2 shows how the trails are presented for the results to the query "zip" on the JDK 1.4 source code. Figure 3 shows the trails more clearly. It can be easily seen from the first trail that there is a member variable

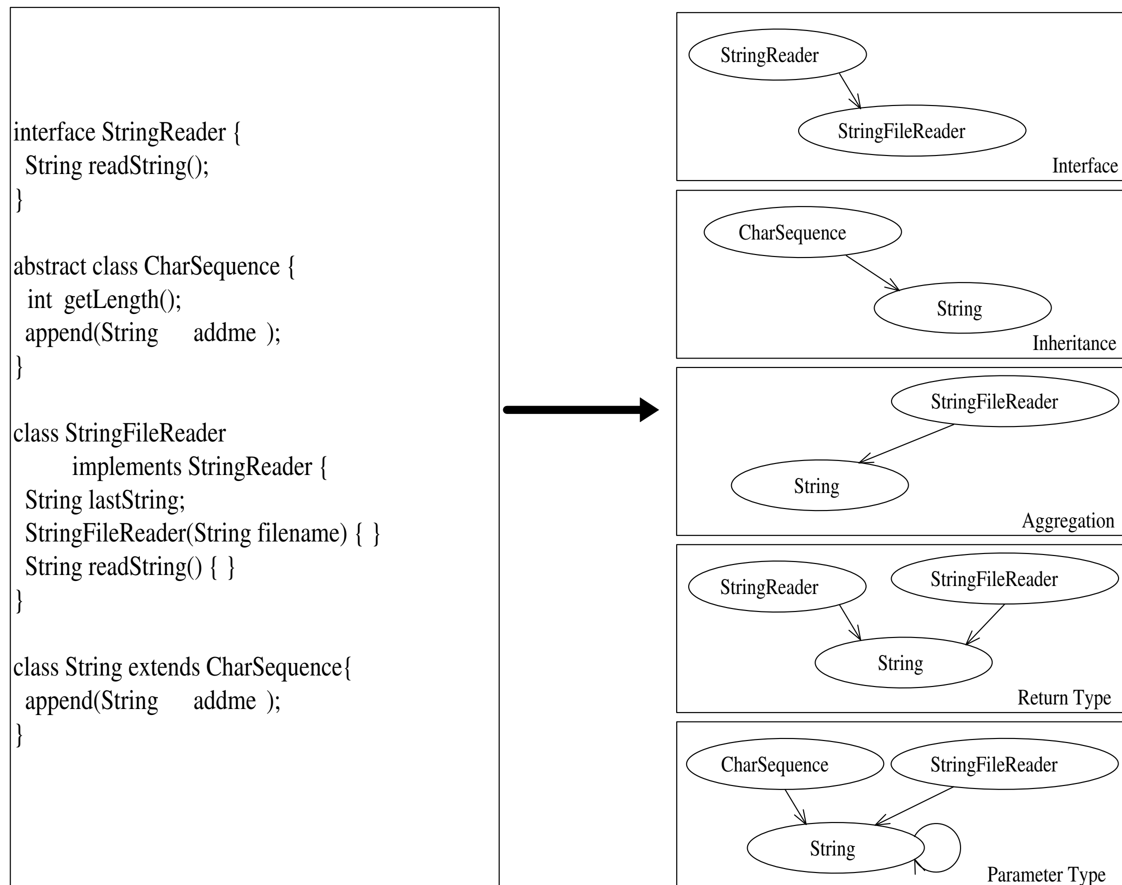


Figure 1. Illustration of coupling types and their graph representations.

of type ZipFile in the class ZipFileInputStream. The second and third trails start with the common root, ZipFile. These show that one or more methods in the ZipFile class must take ZipEntry as a parameter and that ZipFile has a subclass called JarFile. The fourth trail shows that ZipFile implements the interface ZipConstants. The fifth shows that ZipOutputStream has a member variable of type ZipEntry. The sixth and seventh trails show that both ZipInputStream and JarFile have methods which take ZipEntrys as parameters. The eighth trail shows that JarInputStream has at least one method which returns a ZipEntry and the ninth shows that ZipEntry is the superclass of JarEntry which is, in turn, the superclass for JarFile.JarFileEntry.

Figure 3. Trails returned for the query "zip" on the JDK 1.4 source code.

3 Automating Trail Discovery

Given the graphs of related classes, the navigation engine can be used to construct trails. This works in 4 stages. The first stage is to calculate scores (using $tf.idf$) for each of the classes matching one or more of the keywords in the query, and to isolate a small number of these for future expansion, by combining these scores with a metric called *potential gain* [5, 4]. The second stage is to construct the trails using the *Best Trail* algorithm [5]. The algorithm builds trails using a probabilistic best-first traversal. The third stage involves filtering the trails to remove redundant information. In the fourth and final stage, the navigation engine computes small summaries of each class and formats the results for display in a web browser. Jason Shattu's Java2HTML[1] is used to present the source code, since it provides effective syntax highlighting, has a public API and makes links to both Javadocs and between classes in source code.

3.1 Architecture

AutoCode indexes the Java code using a custom doclet. Figure 4 shows how this works with the other elements of the navigation engine. The doclet uses the class structure to construct the five coupling graphs. It also communicates with an external parser, which manipulates the HTML representation of the source code to create an inverted file. This inverted file is used by the *query engine* to compute relevance scores for each class or page. The *trail engine* uses these scores and the coupling graphs to compute the trails. The NavSearch interface presents the trails as shown in Figure 2.

4 Future Work

Object Oriented languages gain particular benefit from the mapping between classes and Web pages. It is intended that AutoCode be extended to support both C++ and C#. It is also hoped that the system can be extended to allow personalized results so that programmers working on a particular field have query results tailored to their needs.

Certain compromises have been made in the development of AutoCode, which should also be addressed in any future development. AutoCode neither shows the relationships between inner and outer classes nor discriminates between static and object references.

Other graphs can be constructed through static and runtime analysis. These include in-memory object references graphs and call-graphs. Any such graphs could be adapted for use with AutoCode.

AutoCode has been developed as a standalone tool operating within JavaDoc. As such it can be updated by any tool which can control JavaDoc, notably build tools such

[1] http://java2html.com/

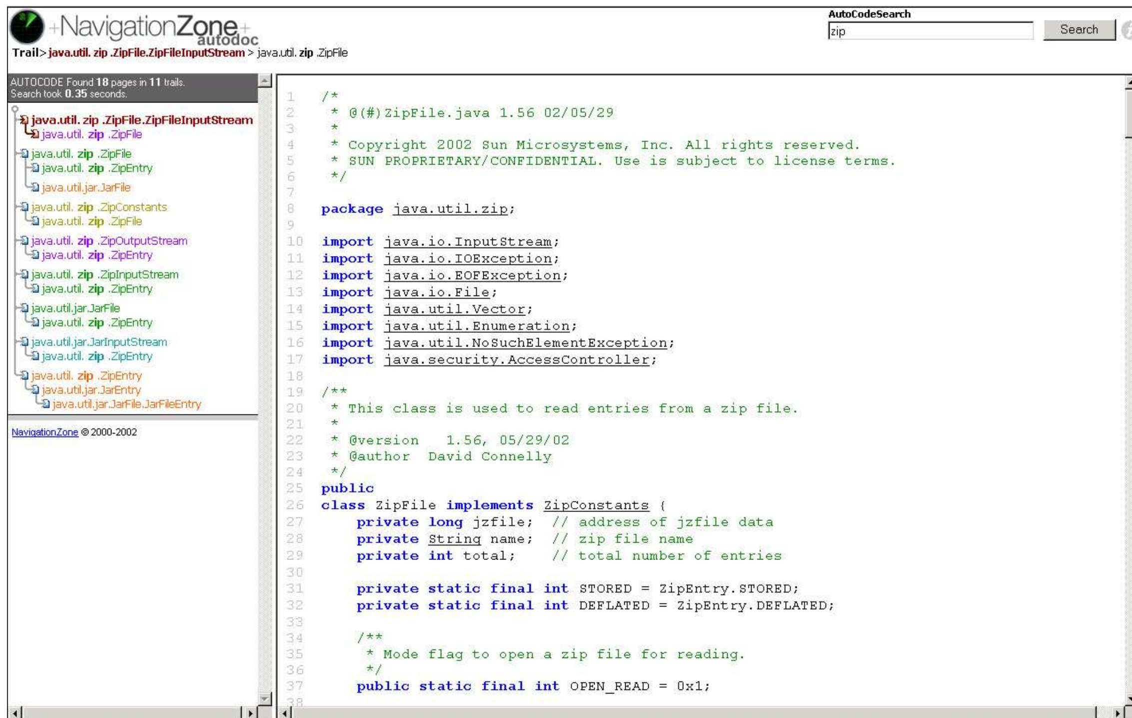


Figure 2. Results for the query "zip" on the JDK 1.4 source code.

as Apache Ant. However, it would be beneficial to embed the interface within a Java IDE so that identified classes can be immediately edited.

Combining these elements would provide developers with a much more flexible tool for identifying relevant classes and the relationships between them.

5 Conclusions

This paper has presented AutoCode - a Web-based tool for computing and presenting memex-like trails across coupling graphs. It benefits from a simple, web-based interface with a strong, well-explored metaphor for displaying class relationships. It works well with very large programs and libraries. For example, the JDK libraries which contain over 6 000 classes and over 1 400 000 lines of code. AutoCode also benefits from platform and IDE independence and uses indexes which can easily be updated during the build process.

However, AutoCode is not without problems. It is restricted to a single language - Java, and ignores certain important relationships between classes. However, it is restricted by a lack of editing features, meaning that identified classes cannot be manipulated without a separate editor.

References

[1] Vannevar Bush. As we may think. *Atlantic Monthly*, 76:101–108, 1945.

[2] Mazlita Mat-Hassan and Mark Levene. Can navigational assistance improve search experience: A user study. *First Monday*, 6(9), 2001.

[3] Siegfried Reich, Leslie Carr, David De Roure, and Wendy Hall. Where have you been from here? : Trails in hypertext systems. *ACM Computing Surveys*, 31(4), December 1999.

[4] Richard Wheeldon and Steve Counsell. Making refactoring decisions in large-scale java systems: an empirical stance. *Computing Research Repository*, cs.SE/0306098, June 2003.

[5] Richard Wheeldon and Mark Levene. The best trail algorithm for adaptive navigation in the world-wide-web. In *Proceedings of 1st Latin American Web Congress*, Santiago, Chile, November 2003.

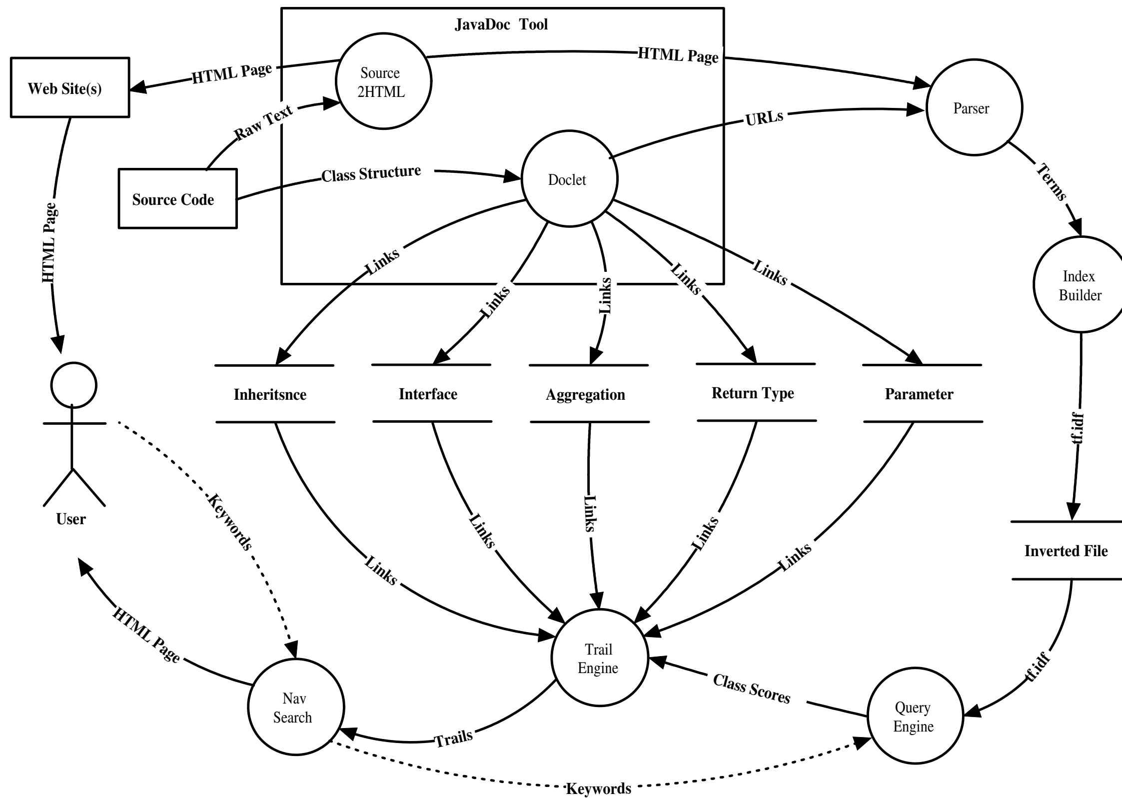


Figure 4. Architecture of AutoCode. Boxes represent external data sources, open-ended boxes represent internal data stores, circles represent processes, solid arrows represent data flow and dotted arrows represent flows of important information (URLs and Queries). Simple keyed "get" instructions (for example in HTTP requests) are omitted for clarity.

Hands-on Collaborative Demo

CodeCrawler - A Lightweight Software Visualization Tool

Michele Lanza (lanza@iam.unibe.ch)
Software Composition Group - University of Bern, Switzerland

Abstract

CodeCrawler is a language independent software visualization tool. It is mainly targeted at visualizing object-oriented software, and in its newest implementation it has become a general information visualization tool. It has been validated in several industrial case studies over the past few years. It strongly adheres to lightweight principles: CodeCrawler implements and visualizes polymetric views, *lightweight visualizations of software enriched with semantic information such as software metrics and source code information.*

1 Introduction

CodeCrawler is a lightweight software visualization tool, whose first implementation dates back to 1998 and it has been implemented as part of Lanza's Ph.D. thesis [3]. In the meantime it has been evolved into an information visualization framework, and has been customized to work in contexts like website reengineering and concept analysis. It keeps however a strong focus on software visualization. CodeCrawler is a language independent SV tool, because it uses the Moose reengineering environment [2] which implements the FAMIX metamodel [1], which among other languages models software written in C++, Java, Smalltalk, Ada, Python, COBOL, etc.

In Figure 1 we see CodeCrawler visualizing itself with a polymetric view called *System Complexity*. The metrics used in this view are the number of attributes for the width, the number of methods for the height, and the number of lines of code for the color of the displayed class nodes.

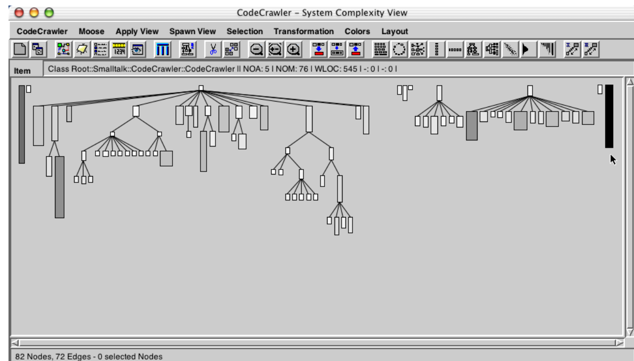


Figure 1. A screenshot of CodeCrawler visualizing itself with a *System Complexity* view.

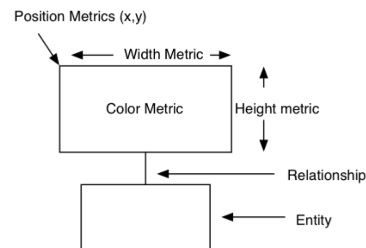


Figure 2. The principle of a polymetric view.

2 The Principle of a Polymetric View

In Figure 2 we see that, given two-dimensional nodes representing entities and edges representing relationships, we enrich these simple visualizations with up to 5 metrics on these node characteristics:

Node Size. The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting.

Node Color. The color interval between white and black can display a measurement. Here the convention is that the higher the measurement the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.

Node Position. The X and Y coordinates of the position of a node can reflect two other measurements. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all layouts can exploit this dimension.

3 Example Polymetric Views

CodeCrawler visualizes three different types of polymetric views:

Coarse-grained views. Such views are targeted at visualizing very large systems (*e.g.,* over 100 kLOC to several MLOC). In Figure 3 we see a *System Hotspots* view of 1.2 million lines of C++ code. The view uses the number of methods for the width and height of the class nodes. We gather for example from this view that there are classes with several hundreds of methods (at the bottom).

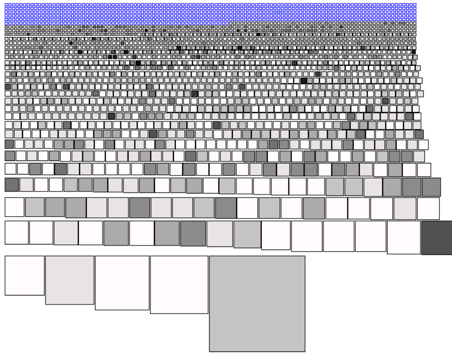


Figure 3. A ***System Hotspots*** **view on 1.2 MLOC of C++ code.**

Fine-grained views. The most prominent view is the *Class Blueprint* view, a visualization of the internal structure of classes and class hierarchies [4]. In Figure 4 we see a class blueprint view of a small hierarchy of 4 classes. The class blueprint view helped to develop a pattern language [3]. In the present example we see the patterns pure overrider, siamese twin, template method design pattern, and template class. The limited size of this paper does not allow us to deepen this discussion, please refer to [3] for more details.

Evolutionary views. The most prominent view is the *evolution matrix* view, a visualization of the evolution of complete software systems [5]. In Figure 5 we see an example of such a visualization, which again allows us to develop a pattern language applicable in the context of software evolution.

4 Features of CodeCrawler

Moreover, CodeCrawler features grouping support, customizable views, has been industrially validated, and is being used if software industry mainly by consultants. CodeCrawler is freeware and can be obtained at http://www.iam.unibe.ch/∼lanza/

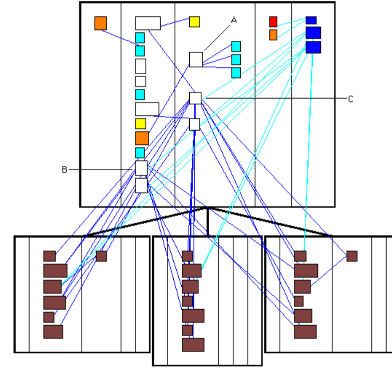


Figure 4. A ***Class Blueprint*** **view on a small hierarchy of 4 classes written in Smalltalk.**

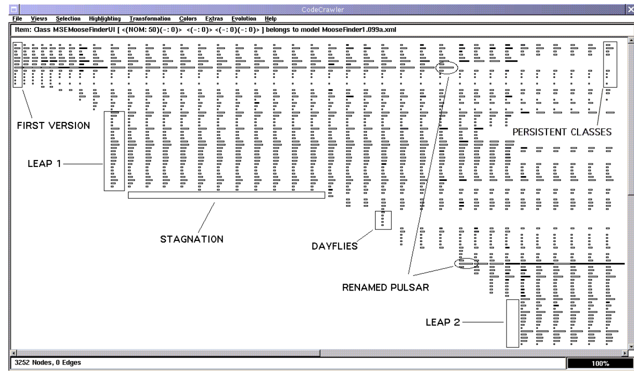


Figure 5. An ***Evolution Matrix*** **view on 38 versions of an application written in Smalltalk.**

References

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[2] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[3] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003.

[4] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[5] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002*, pages 135–149, 2002.

AutoCode: Using Memex-like Trails to Improve Program Comprehension

Richard Wheeldon, Steve Counsell and Kevin Keenoy
Department of Computer Science
Birkbeck College, University of London
London WC1E 7HX, U.K.
{richard,steve,kevin}@dcs.bbk.ac.uk

1 Introduction

In his seminal paper "As We May Think" [1], Vannevar Bush suggested a future machine called a "memex". In doing so, he introduced the world to the concept of linked documents and of the *trail* - a sequence of linked pages. The concept of trails is well established in the hypertext community and many systems have been built which support their construction [2].

Previous work has described a *navigation engine* for automatically constructing trails as a means of assisting users browsing Web sites [4]. This navigation engine was further used to provide search and navigation facilities for Javadoc program documentation. If JavaDoc-style program documention, which is derived from source code, can be indexed, it seems logical that the source code itself can be indexed.

We have developed a new tool called AutoCode based upon the navigation engine design. AutoCode provides full-text indexing of the java source code and uses a probabilistic best-first algorithm to identify trails in graphs of coupling-type relationships.

2 Trails on Java Code

Classes and objects in OO systems to not work in isolation. The classes are connected to each other by various dependencies. The Java language connects classes together via five coupling relationships - Aggregation, Inheritance, Interface, Parameter and Return Type [3]. Each of these coupling relations can be used to construct a graph of dependencies. AutoCode constructs trails on each of these five graphs and presents them in a Web-based interface.

The *NavSearch* user interface used to present the trails (figure 1) has three main elements. At the top is a *navigation tool bar* comprising of a trail of classes considered most relevant (the "best trail"). On the left is a *navigation tree window* showing all the trails. Whenever the mouse pointer moves over these trails, a small pop-up appears which shows metadata and an extract. The rest of the display is dedicated to showing the source code of the selected class. A demonstration of this interface showing the 6000 classes of the JDK libraries is available at http://nzone.dcs.bbk.ac.uk/.

Each trail is colour-coded according to the type of coupling involved. This coupling type is also shown in the pop-up for each class. green trails denote parameter type references, cyan trails denote return-type references, gold trails show interface extensions, purple trails shows chains of aggregation links and orange trails show inheritance relationships from subclass to superclass.

Figure 1 shows how the trails are presented for the results to the query "zip" on the JDK 1.4 source code. Figure 2 shows the trails more clearly. It can be easily seen from the first trail that there is a member variable of type `ZipFile` in the class `ZipFileInputStream`. The second and third trails start with the common root, `ZipFile`. These show that one or more methods in the `ZipFile` class must take `ZipEntry` as a parameter and that `ZipFile` has a subclass called `JarFile`. The fourth trail shows that `ZipFile` implements the interface `ZipConstants`. The fifth shows that `ZipOutputStream` has a member variable of type `ZipEntry`. The sixth and seventh trails show that both `ZipInputStream` and `JarFile` have methods which take `ZipEntrys` as parameters. The eighth trail shows that `JarInputStream` has at least one method which returns a `ZipEntry` and the ninth shows that `ZipEntry` is the superclass of `JarEntry` which is, in turn, the superclass for `JarFile.JarFileEntry`.

AutoCode indexes the Java code using a custom doclet.

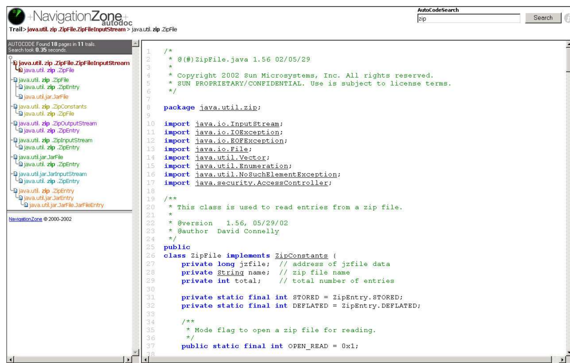


Figure 1. Results for the query “zip” on the JDK 1.4 source code.

Figure 2. Trails returned for the query “zip” on the JDK 1.4 source code.

This communicates with an external parser and constructs the five coupling graphs. Given the graphs of related classes, the navigation engine can be used to construct trails. This works in 4 stages. The first stage is to calculate scores (using $tf.idf$) for each of the classes matching one or more of the keywords in the query, and isolate a small number of these for future expansion, by combining these score with a metric called *potential gain* [4, 3]. The second stage is to construct the trails using the *Best Trail* algorithm [4]. This builds trails using a probabilistic best-first traversal. The third stage involves filtering the trails to remove redundant information. In the fourth and final stage, the navigation engine computes small summaries of each class and formats the results for display in a web browser. Jason Shattu’s Java2HTML[1] is used to present the source code, as it provides effective syntax highlighting, has a public API and makes links to both Javadocs and between classes in source code.

3 Future Work

Object Oriented languages gain particular benefit from the mapping between classes and Web pages. It is intended that AutoCode be extended to support both C++ and C#. It is also hoped that the system can be extended to allow personalized results so that programmers working on a particular field have query results tailored to their needs.

References

[1] Vannevar Bush. As we may think. *Atlantic Monthly*, 76:101–108, 1945.

[2] Siegfried Reich, Leslie Carr, David De Roure, and Wendy Hall. Where have you been from here? : Trails in hypertext systems. *ACM Computing Surveys*, 31(4), December 1999.

[3] Richard Wheeldon and Steve Counsell. Making refactoring decisions in large-scale java systems: an empirical stance. *Computing Research Repository*, cs.SE/0306098, June 2003.

[4] Richard Wheeldon and Mark Levene. The best trail algorithm for adaptive navigation in the world-wide-web. *Computing Research Repository*, cs.DS/0306122, June 2003.

[1] http://java2html.com/

Demonstration of Advanced Layout of UML Class Diagrams by SugiBib

Holger Eichelberger
chair of computer science II
Würzburg University
Am Hubland, 97074 Würzburg, Germany
eichelberger@informatik.uni-wuerzburg.de

Jürgen Wolff von Gudenberg
wolff@informatik.uni-wuerzburg.de

1 Introduction

The Unified Modeling Language (UML) [7] has become the standard for specifying object-oriented software systems. Some of the tools are primarily designed to work on a direct mapping between the design diagrams and the software and vice versa. Since understanding the software is usually using more abstract concepts than those defined in programming languages, restriction of the UML used in these tools is not permissible. On the other side, visualization of changes to the software implementation and design documents require sophisticated layout algorithms. In [2] we have shown that most of the tools are not sufficient in that field.

2 Layout of UML class diagrams

Obviously a UML class diagram can be described as graph $G = (V, E)$ with nodes V and edges E. V can then be partitioned into nodes of different types: packages and classes may be nested (nested nodes might be structured according to different criteria like coupling [3], subsystems [7] or component classifiers in the new UML 2.0), annotations can be attached to all model elements, association classes (classes attached to an association can be part of other associations or generalization relations) and natural or arbitrary clusters (like design patterns or higher associations). According to [1, 3] E should be partitioned into a set of hierarchical edges E_H and a set of non-hierarchical edges E_N using heuristics or user defined preferences. Different types of edges have to be respected: Generalizations, associations, aggregations, compositions and dependencies with different textual and symbolic adornments. Adornments of edges may not overlap adornments of other edges or node boxes. Constraints concerning two or more associations (denoted by a dashed line connecting the associations) have to be laid out.
The layout calculated by the algorithm should be optimized for a clear representation of a software design diagram, easy to read, understandable and therefore a large set of criteria for optimal readability according to semantical reasons must hold beside usual graph drawing criteria like overall number of edge crossings and bends [1, 3].
The layout algorithm is clearly explained in [1, 5].

3 Architecture of the framework

SugiBib is a pure Java framework which primarily was designed to implement a general, highly configurable, component-based version of the Sugiyama algorithm [10]. The components can be combined in different sequences to implement other layout algorithms. Because of the component architecture information hiding is preserved between two consecutive steps. Nodes and edges of the framework are parametrized by their individual graphical information.
SugiBib was instantiated to represent UML class diagrams and provides interactive frontends in AWT and Swing as well as online and batch rendering servers. Advanced features of UML class diagrams like association classes and annotations are treated by an extension of the Seemann algorithm [9].
Currently SugiBib accepts input in UMLscript [4], a programming language for object oriented design. The standard XML Metadata Interchange format XMI [8] is extended by different vendors with their proprietary notation for layout information. Therefore a general import of XMI into SugiBib can be realized only by extensive XSLT preprocessing. The current implementation is prepared for the XMI version produced by MagicDrawUML (`www.nomagic.com`). As an intermediary format while processing XMI, the XML version of UMLscript called XUMLscript is produced. Therefore SugiBib is able to accept plain XUMLscript as input, too. Additionally we are working to read diagrams in the new UML 2.0 Diagram Interchange format [6] XMI[DI] or XMI[UML+DI], respectively.
The output format we produce is the laid out graph and as a proprietary textual XML representation of the internal

graph to be postprocessed by diff, e.g., currently for debugging purpose only. Output of standardized XML (XMI[DI] or XMI[UML+DI]) is planned for the near future.

Current information on SugiBib can be obtained from `www.sugibib.de`.

References

[1] H. Eichelberger. Aesthetics of class diagrams. In *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 23–31. IEEE, 2002.

[2] H. Eichelberger. Evaluation-report on the layout facilities of UML tools. TR 298, Institut für Informatik, Universität Würzburg, jul 2002. Institut für Informatik, Universität Würzburg.

[3] H. Eichelberger. Nice class diagrams admit good design? In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 159–ff. ACM Press, 2003.

[4] H. Eichelberger and J. W. von Gudenberg. UMLscript sprachspezifikation. TR 272, Institut für Informatik, Universität Würzburg, feb 2001. Institut für Informatik, Universität Würzburg.

[5] H. Eichelberger and J. W. von Gudenberg. On the visualization of Java programs. In S. Diehl, editor, *Software Visualization, State-of-the-Art Survey*, volume 2269 of *Lecture Notes in Computer Science*, pages 295–306. Springer, Springer, 2002.

[6] OMG. UML 2.0 Diagram Interchange. Second (Final) Revised Submission, OMG document number ad/2002-12-20, Version 1.0, January 6, 2003, via `http://www.xml-strategie.de/files/UML2DIRevSub.pdf`

[7] OMG. Unified Modeling Language Specification. Version 1.5, March 2003 via `http://www.omg.org`.

[8] OMG. XML Metadata Interchange (XMI). Version 1.1, via `http://cgi.omg.org/docs/ad/99-10-02.pdf`.

[9] J. Seemann. Extending the sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. *Lecture Notes in Computer Science*, 1353:415–423, 1997.

[10] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, Feb. 1981.

Figure 1. A part of the static structure of a GUI application drawn by SugiBib.

GENISOM

James Brittle and Cornelia Boldyreff
Department Of Computer Science
University Of Durham
{j.g.brittle,cornelia.boldyreff}@durham.ac.uk

Abstract

GENISOM, is an offspring component of the GENESIS software engineering platform, incorporating the generation, maintenance and viewing of self-organizing maps. The self-organizing map's unsupervised clustering method has been used to visualise a large software collection.

Key Words: Self Organizing Maps, GENISOM, software visualisation

1 Introduction

The Self-Organizing Map (SOM),invented by Prof. Teuvo Kohonen in the early 1980s [4], is an unsupervised, clustering algorithm. It has been demonstrated to aid programmers in the process of reverse engineering by discovering common features within legacy code [2] and to assist in object recovery[1], although visualisation of the associated maps is not explicitly considered in reports of this research.

A prominent problem within the field of Software Engineering concerns reuse. Reusable assets are in abundance, over the web and in libraries but it is extremely difficult to locate reusable software artefacts that are relevant to a particular application. The necessary organisation is often lacking and difficult to achieve given the dynamic nature of such software collections. This problem can also be found where a large evolving software system consists of an ever growing number of components and the managment and hence the comprehension of the associated software artefacts tends to be increasingly difficult.

Having suitable visualisations of such software collections can mitigate the problem identified above.

2 GENISOM

GENISOM is a client/server application designed to manage and enable viewing of SOMs. Figure 1 illustrates a simplified architecture diagram for the GENISOM system.

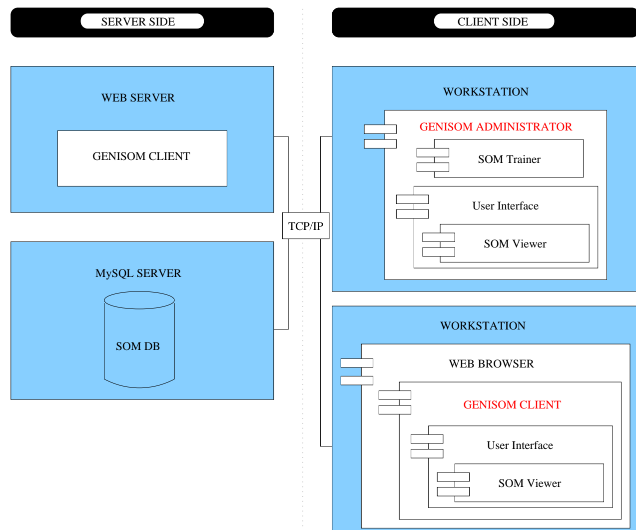


Figure 1. GENISOM architecture

The GENISOM Administrator is used to manage SOMs; it enables their creation and maintenance. The generated SOMs are then stored within a MySQL database, from which the GENISOM Client can then retrieve the data and display the generated maps. The Client is web based using Java Web Start[1] to aid its accessibility to users. The architecture of the system enables distributed software engineering teams to work together.

There are two main use case scenarios for the system; firstly the Administrator could be used by the librarian of a reuse library and the Client by the system developers (see Figure 2), therefore aiding them in the location of reuse candidates. Secondly both tools could be used by members of a software development team to aid program comprehension, or help in decisions regarding restructuring and reengineering of the system.

[1] http://www.java.sun.com/products/javawebstart/

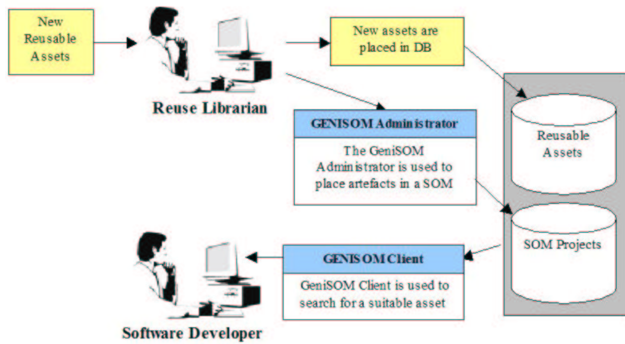


Figure 2. Reuse infrastructure with GENISOM system in place

3 Visualisations of Maps

The GUI for the SOM evolved through a number of stages which resulted in the final 2D map as illustrated in Figure 3.

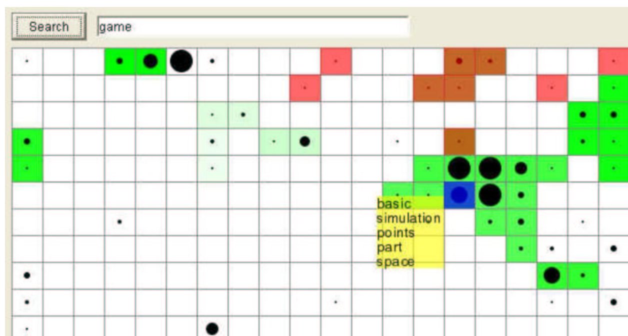


Figure 3. Screenshot of 2D Map

The neural net is arranged as a grid with the inputs (e.g. reusable artefacts) being attached to the neurons. The black dots on grid cells (i.e. neurons) indicate that inputs have been matched to them, with the size of the dot representing the number of them. The green shading of the grid cells indicates the boundaries of similar clusters of neurons.

The map is interactive allowing the user to select a particular neuron (the blue cursor indicating the selection), this displays the neuron's labels, five or less words that best describe the inputs matched to it. As well as displaying the labels for a selected neuron, details of the inputs matched to it are also displayed in a side bar though this is not depicted in the figures.

Browsing is aided by a search system coupled to the map which highlights the results in red for a certain search string.

Using 2D limits the amount of information that can be visualised. Improvements to the GENISOM GUI therefore naturally led on to the development of a 3D map using the Cityscapes technique which has already found application in software visualisation [3]. Using this technique in GENISOM each value is plotted as a column (or 'building'). The 'buildings' are plotted on the same horizontal plane, allowing differences in height and position to be analysed. In relation to a SOM, each building represents a neuron and the height of the building relates to the number of inputs that are matched to it. The implementation of this used Java3D[2] and in the final system the option was made available to switch between using 2D and 3D interfaces.

Figure 4 illustrates the 3D map; the user's view can be rotated and zoomed in and out. Furthermore, the user can also interact with the 3D map in the same manner as the 2D map following the same colour scheme.

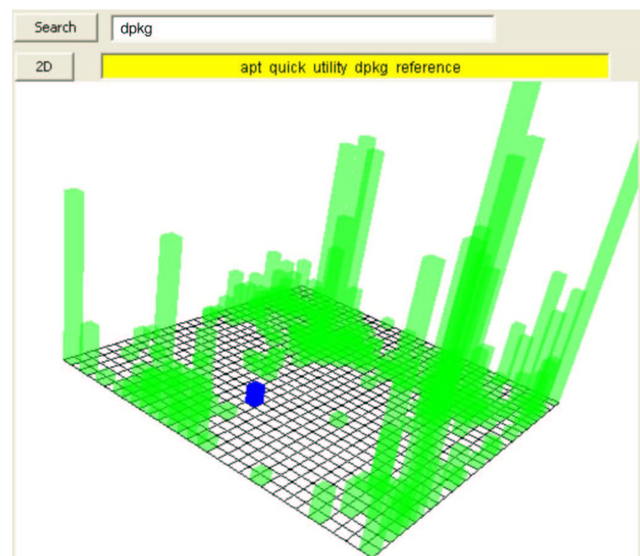


Figure 4. Screenshot of 3D Map

References

[1] A. Chan and T. Spracken. Discovering common features in software code using self-organizing maps. In *Proceedings of the International Symposium on Computational Intelligence*, Kosice Slovakia, August 2000.

[2] A. Chan and T. Spracklen. Object recovery using hierarchical self-organizing maps. In *Proceedings of the International Conference on Engineering Applications of Neural Networks*, Kingston Upon Thames UK, July 2000.

[3] C. Knight. *Virtual Software In Reality*. PhD thesis, Durham University, 2000.

[4] T. Kohonen. *Self-Organizing Maps*. Information Sciences. Springer, second edition, 1997.

[2] http://java.sun.com/products/java-media/3D/

Source Viewer 3D (sv3D)
A System for Visualizing Multi Dimensional Software Analysis Data

Andrian Marcus, Louis Feng, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
amarcus@cs.kent.edu, lfeng@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

Source Viewer 3D is a software visualization framework that uses a 3D metaphor to represent software system and analysis data. The 3D representation is based on the SeeSoft pixel metaphor. It extends the original metaphor by rendering the visualization in a 3D space. New, object-based manipulation methods and simultaneous alternative mappings are available to the user.

1. Description

Source Viewer 3D (sv3D) is a software visualization framework that builds on the SeeSoft [1, 2] metaphor. It brings a number of enhancements and extensions over SeeSoft-type representations. In particular it creates 3D renderings of the raw data and various artifacts of the software system and their attributes can be mapped to the 3D metaphors at different abstraction levels. It implements improved, object-based user interactions, is independent of the analysis tool, and it accepts a simple and flexible input in XML format. The output of numerous analysis tools can be easily translated to sv3D input format and the design and implementation of our system is extensible.

SeeSoft-like tools have a variety of uses in assisting the user solving software engineering and comprehension tasks. sv3D can be used for all these tasks such as: fault localization [4], visualization of execution traces [6], source code browsing [3], impact analysis, evolution, complexity, and slicing [1], etc. In addition, by allowing visualization of additional information (via 3D), sv3D can be used for solving other more complex tasks. For example, in the case of Tarantula [4], using height instead of brightness would improve the visualization and make the user's task easier.

Most software engineering tasks during maintenance and evolution require understanding of various elements of the software system and also of data resulted from analysis. The main features of sv3D, namely, advanced user interactions and usage of the 3D space for visualization directly support the user in achieving a better understanding of analysis data. This process, in turn, directly supports a variety of tasks.

2. Support for User Interaction

We focus here on the types of user tasks and interactions that are supported by sv3D. While this is not directly related to solving/visualizing specific software engineering tasks it is prerequisite for a software visualization tool.

One of the strongest features of sv3D is its *overview* features. The underlying 2D visualization construct used in designing the poly cylinder containers is the pixel bar chart [5], which generalizes the concept used by SeeSoft. Thus sv3D can show large amounts of source code in one view just as the SeeSoft metaphor. Figure 1 shows a 3D overview of a small system with 30 C++ source code files and approximately 4000 lines of code. Each file is mapped to one container. Each container is made up of a number of poly cylinders. Each poly cylinder represents a line of source code. In this simple example shading (color) is used to represent the type of control structure a statement is in and the height is used to represent the nesting level. On top of each container the name of the associated file is visible. When manipulating a container in the 3D space, the name of the file always faces the camera.

sv3D supports *zooming* and *panning* at variable speeds. This is especially important because the visualization space can be quite large. Each container in the visualization can be manipulated individually (rotate, scale, translate). The user can also zoom in and out on the entire space. Files can be brought into a closer view and manipulated for a better camera angle.

At this point sv3D directly supports a number of *filtering* methods. Un-interesting units can be filtered through their attributes or by direct manipulation. Transparency is used to deal with both occlusion and filtering. The user can chose various degrees of transparency on each class of cylinder, based on their attributes (color, shape, or texture). With semi-transparency the global context is preserved and heuristic information is retained. Elevation can also be used to filter out un-interesting units by lifting them into separate levels.

Currently our emphasis with regard to *details-on-demand* is for simplicity. It is important to be able to

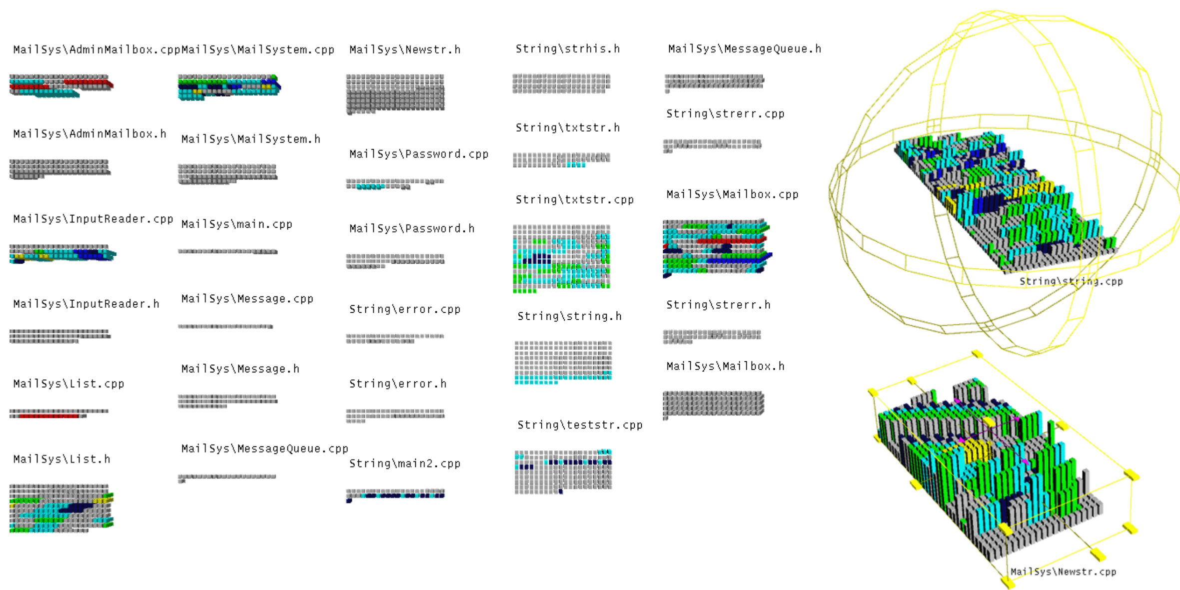


Figure 1. Overview in the 3D space of the mailing system. Color represents control structure and height represents nesting level. Two files have active manipulators (handle box for scaling on the left and track ball for rotating on the right). For a color view see www.sdml.cs.kent.edu

support user interaction, therefore performance is important. Two types of 3D manipulators (i.e., track ball and handle box) are available to the user to interact with the visualization. An information panel displays the data values on selected items.

The *relationships* between items are shown through the elements of the visualization that do not directly support representation of quantitative data (such as shape, texture, and position). The other elements (such as color and height) can also be used to show relationships. The 3D space allows arranging the containers in any place. We are investigating ways to use links between the 3D containers and arrange them in a graph layout.

The user can take snapshots of the current view to track a *history*. The current view is described by a scene graph, which is composed by the attributes of the camera and all 3D objects. These snapshots of the scene graph can be saved and reviewed. A sequence of such snapshots can be played, thus representing a path within the visualization. More than that, we intend to build into sv3D change tracking based on individual users.

3. Current and Future Work

sv3D is implemented in C++ and uses Qt for the user interfaces and Open Inventor for 3D rendering. See www.sdml.cs.kent.edu for additional information and downloads.

In the future versions of sv3D, position of the cylinder within a container can represent another type of information (or dimension). We need to define these visual attributes very carefully to ensure their usefulness. Containers in the 3D space can possibly be connected by edges to form a 3D graph. This will allow representation of hierarchical data and also diagrammatic visualizations (e.g., UML class diagrams). A number of user experiments to evaluate this system are being planned.

This work was supported in part by grants from the Office of Naval Research N00014-00-1-0769 and the National Science Foundation CCR-02-04175.

4. References

[1] Ball, T. and Eick, S., "Software Visualization in the Large", *Computer*, vol. 29, no. 4, April 1996, pp. 33-43.

[2] Eick, S., Steffen, J. L., and Summer, E. E., "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", *IEEE TSE*, vol. 18, no. 11, November 1992, pp. 957-968.

[3] Griswold, W. G., Yuan, J. J., and Kato, Y., "Exploiting the Map Metaphor in a Tool for Software Evolution", in Proceedings of ICSE'01, Toronto, 2001, pp. 265-274.

[4] Jones, J. A., Harrold, M. J., and Stasko, J. T., "Visualization for Fault Localization", in Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, 2001, pp. 71-75.

[5] Keim, D. A., Hao, M. C., Dayal, U., and Hsu, M., "Pixel bar charts: a visualization technique for very large multi-attribute data sets", *Information Visualization*, vol. 1, no. 1, March 2002, pp. 20-34.

[6] Reiss, S. P., "Bee/Hive: A Software Visualization Back End", in Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, 2001, pp. 44-48.

Plugging-in Visualization:

Experiences Integrating a Visualization Tool with Eclipse

Rob Lintern Jeff Michaud Margaret-Anne Storey Xiaomin Wu

Dept. of Computer Science
University of Victoria
Victoria, BC Canada
{rlintern, jmichaud, mstorey, xwu }@uvic.ca

Abstract

The Eclipse platform presents an opportunity to openly collaborate and share visualization tools amongst the research community and with developers. In this paper, we present our own experiences of "plugging-in" our visualization tool, SHriMP Views, into this environment. The Eclipse platform's Java Development Tools (JDT) and CVS plug-ins provide us with invaluable information on software artifacts relieving us from the burden of creating this functionality from scratch. This allows us to focus our efforts on the quality of our visualizations and, as our tool is now part of a full-featured Java IDE, gives us greater opportunities to evaluate our visualizations. The integration process required us to re-think some of our tool's architecture, strengthening its ability to be plugged into other environments. We step through a real-life scenario, using our newly integrated tool to aid us in merging of two branches of source code. Finally we detail some of the issues we have encountered in this integration and provide recommendations for other developers of visualization tools considering integration with the Eclipse platform.

Introduction

Many visualization tools that are developed in the research community are customized applications that are built from scratch. These visualization tools are dependent on having access to information sources about the software that are both rich and accurate. Research groups often have to write their own tools or even beg, borrow and steal parsers, and other information extractors to provide data for the visualization technique. These efforts are usually disjointed and many research groups have experienced frustration from reinvention of the wheel. Furthermore, since the visualization tools are stand-alone applications and do not integrate easily with the existing tools that developers use, it is difficult to evaluate their usefulness in real world contexts. Moreover, it is often impossible to combine features and tools from these stand-alone applications, or to compare them as each will offer many different features.

Over the past few years, we too initially focused on developing a stand-alone software visualization tool to assist in program understanding. Our tool is called SHriMP Views, which stands for Simple Hierarchical Multi-Perspective Views. SHriMP uses a nested graph view to display hierarchical structures in a Java program (see Fig. 1). Composite nodes in the graph represent key structures (for example, packages and classes) in the software. Leaf nodes correspond to entities in the software such as methods, and data types. Arcs in the graph show dependencies between these artifacts and may show inheritance, composition and association relationships. The *nested interchangeable view* feature in SHriMP allows a user to look at different presentations of information at any level of detail. A programmer can browse source code or documentation by following hyperlinks that result in animated panning and zooming motions over the nested graph.

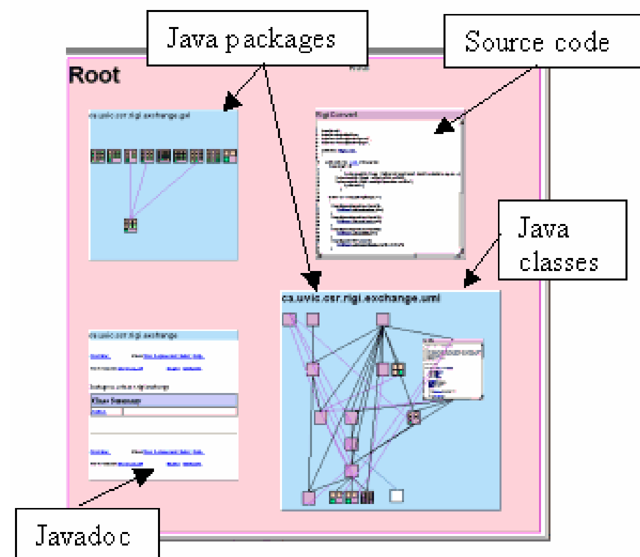


Figure 1. A SHriMP View of a Java program.

Integration

Over the past year we successfully integrated SHriMP with the open source Eclipse project (see Fig. 2). Eclipse (ww.eclipse.org) is a general purpose platform upon which other tools can be built as *plug-ins*. The JDT (Java Development Tools) are a suite of such plug-ins, comprising a full featured Java IDE, which comes bundled with the free download of the Eclipse platform. Many other commercial and research groups have developed further plug-ins for the Eclipse platform and the JDT– such as UML tools, version control tools, team support etc.

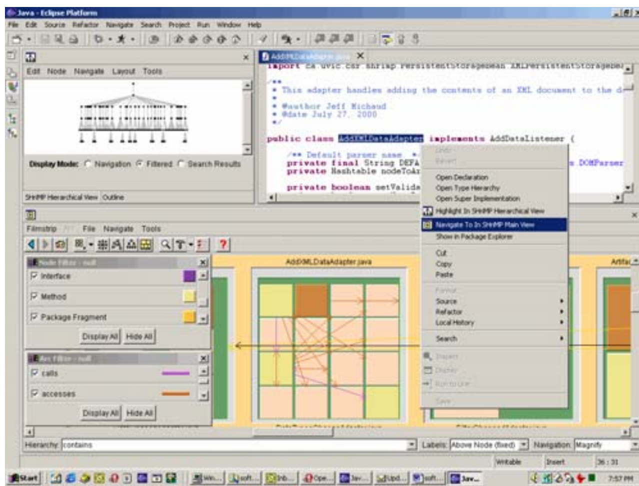


Figure 2: SHriMP plugged into the Eclipse platform. *SHriMP Hierarchical View* is shown in top left pane, *SHriMP Main View* shown is shown in the bottom pane and source code shown in top left pane, all of which are synchronized.

We refer to the integration of SHriMP with the Eclipse JDT as "**Creole**". Since Eclipse provides access to the program repository, we now can instead focus on visualization and how it can be further developed to provide support to the existing features in Eclipse.

Integrating SHriMP with Eclipse has also provided access to new information sources via existing plug-ins. Of particular interest is the CVS plug-in which is an integrated GUI front-end for the CVS version control system. We conjecture that visualization of team relevant information such as CVS histories could be of significant assistance in collaborative tasks. To explore this topic, we integrated SHriMP with the CVS plug-in giving us the ability to visualize information stored in the underlying CVS repositories. We refer to this integration as **"Xia"**.

Our most recent work has been spent creating a composite visualization of information from both the JDT repository and the CVS repository. We believe that such views could be used to reveal:

- Who is responsible for which parts of the system?
- Which parts of the system tend to change frequently?
- Which parts have been changed since a particular date?
- What are the relationships between these parts of interest and the rest of the system?
- … and any combination of the above

Discussion

We have found the integration of SHriMP with the JDT (i.e. Creole) to be of benefit when trying to **navigate** and **understand** code written by other groups. It is especially powerful when we are first exploring code and trying to get an overview of the scope and design of a program. With respect to providing support for **collaboration** and project **management**, we have found the visualizations of the CVS information to be very useful despite the fact that our tool is still at a prototype stage and is not very robust.

There are still, however, many issues that remain from this trial and many questions that have been raised. We are faced with much to explore. Much more empirical work is required before any conclusions can be drawn. We need to discover the specific tasks that our visualizations could help with. This leads us to the underlying question: *who* exactly is our user? Is it the team lead or software designer who needs a tool to support high-level decisions, or is it the programmer doing day-to-day programming tasks, or is it both? We need to empirically study Creole to determine whether or not it actually decreases cognitive load and increases performance on specified tasks.

Another issue we have come across is one that arises with any visualization. It is difficult to decide which view of the information is the most useful. Our visualization depends on the information we have at hand. In our case we have two sources of information: the JDT and the CVS plug-ins. Through these two plug-ins we now have easy access to reliable information, but, is it the right information for producing visualizations that help with the software task at hand?

Other future work will include using more animation in our visualizations to aid in refactoring code, comparing code, and synchronizing code with a repository, CVS repository we could animate the evolution of a project over time.

One major issue faced in our integration with Eclipse is that its GUI is built from a toolkit called SWT instead of using the more widely used AWT and Swing toolkits. The major advantage of SWT is that it uses native widgets wherever possible, increasing speed and guaranteeing the native platform's look and feel. This departure has been a major hurdle for our integration; SHriMP relies heavily on a zooming library based on AWT and Swing, making it difficult for us to create an SWT only version of our software. The approach taken to embed Swing and AWT widgets inside of SWT widgets is still problematic and results in some screen flickering, missing popup menus, and other GUI glitches. Furthermore, this UI integration currently only works on the Windows platform, and is not encouraged or officially supported by OTI (OTI are the primary developers of Eclipse).

Despite the work required to redesign aspects of our architecture, and issues integrating Swing and SWT widgets, the effort required to do the integration was not that arduous, especially when we consider what we have gained as a tool developer. The biggest issue that we have faced doing research in the largely non-validated area of software visualization, is trying to evaluate our own work. This has in part been hampered by not being able to evaluate how the visualization techniques work when they are used as part of the normal tools used by developers. By integrating with Eclipse, we can now continue with these evaluations, and furthermore, combine features from our tool with other visualization tools for further feedback and comparison.

More Information

http://shrimp.cs.uvic.ca/

Workshop Part II

Program Visualization Support for Highly Iterative Development Environments

Michele Lanza
lanza@iam.unibe.ch
Software Composition Group
University of Bern, Switzerland

Abstract

Software Visualization is, despite the many publications and advances in this research field, still not being considered by mainstream software industry: currently very few integrated development environments offer (if at all) only limited visualization support, and in general it can be said that software visualization is being ignored at a professional level by the average software developer. Moreover, even relatively successful software visualization tools (such as Rigi, Shrimp, JInsight, etc.) are seldom being used except by their developers themselves. In this position paper, based on our own experience and an analysis of the current state and possible future trends of integrated development environments, we put up a non-exhaustive list of features that software visualization tools should possess in the future to have more consideration by mainstream development.

1 Introduction

Software visualization is a fairly recent research field dating back to the 1960's, and started to become an established research field in the 1980's. The main benefit that software visualization (as a specialization of the more general field of information visualization) brings, is that it "provides an ability to comprehend huge amounts of data" and "allows the perception of emergent properties [of the data] that were not anticipated" [28]. Despite these and other benefits of software visualization, the contributions that this field has made to mainstream software industry are barely noticeable and largely ignored.

In this position paper we want to analyze this research field from different points of view, investigate and discuss some of the reasons that make software visualization still a "secondary" research domain, and put up a non-exhaustive list of features that need to be implemented by the developers of software visualization tools, should they want to propagate their research into an industrial context.

2 Software Visualization

> "Software visualization is [..] the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software." [20]

Software visualization is a specialization of *information visualization*, whose goal is to visualize any kind of data, while in software visualization the sole focus lies on visualizing software. Information visualization is defined as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition." [3]. It derives from several communities. Starting with Playfair (1786), the classical methods of plotting data were developed. In 1967, Jacques Bertin, a French cartographer, published his theory in *the semiology of graphics* [2]. This theory identifies the basic elements of diagrams and describes a framework for their design. A few decades later Edward Tufte published a theory of data graphics that emphasized maximizing the density of useful information and minimized recurrent errors in data visualization [25, 26, 27]. Both Bertin's and Tufte's theories have influenced the various communities that led to the development of information visualization.

The goal of information visualization is to *visualize any kind of data*. Note that the above definition by Card *et al.* of information visualization does not necessarily imply the use of vision for perception: visualizing does not only involve *visual* approaches, but any kind of *perceptive* approach. Data can be perceived by a person by using the senses at his/her disposition, *e.g.,* apart from seeing the data, a person can also hear it (information auralization) and/or touch it (by using virtual reality technology). However, most information visualization systems currently use computer graphics which render the data using 2D- and/or 3D-views of the data. Applications in information visualization are so frequent and common, that most people do not notice them:

examples include metereology (weather maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc.

In short, information visualization is about visualizing any kind of data, while software visualization is about visualizing software.

According to Stasko *et al.* the field of software visualization can be divided into two separate areas [20]:

1. *Program visualization* is the visualization of actual program code or data structures in either static or dynamic form. Most of the present approaches deal with *static code visualization*, because the source source code is visualized by using only information which can be *statically* extracted without the need to actually run the system.

2. *Algorithm visualization* is the visualization of higher-level abstractions which describe software. A good example is *algorithm animation*, which is the dynamic visualization of an algorithm and its execution. It is used to explain the inner working of algorithms like sort-algorithms. In the meantime this discipline has lost importance, mainly because the advancement in computer hardware and the possibility to use standard libraries containing such algorithms have shifted the focus away from the implementation of such algorithms.

In this paper we concentrate ourselves on program visualization, because most software visualization tools belong to this category, and because algorithm visualization has greatly lost importance in the past two decades, except for educational contexts (*e.g.,* teaching algorithms to students).

3 The Mission

The overall mission of program visualization is to visualize the static structure or the dynamic behavior of a software system.

In that sense software visualization researchers are trying to visualize an immaterial construct (software has no physical limits, no notion of proximity or distance) like software *the way it is*, although this is (by definition) not feasible: there is no unique and correct way of visualizing software. Taking an ambitious stance, we claim that the ultimate goal of a good visualization is to become the preferred way of developers of looking at software. We do not claim that software visualization could replace the most important and still most used way of perceiving software: code reading. We rather suggest that software visualization should have a symbiotic relationship with the practice of code reading by pointing the viewer to the location in the system where he should read and/or modify the code. According to the program cognition model vocabulary proposed by Littman *et al.* [15] we propagate an approach of software understanding that is *opportunistic* in the sense that it is not based on a *systematic* line-by-line understanding but *as needed*.

Moreover, software visualization has become relevant in the reverse engineering research community. Software reverse engineering is defined by Chikosfky and Cross as "the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction" [4]. The goal is thus to *construct a mental model* of a software system. Storey *et al.* have highlighted in various papers that a good software visualization is a powerful asset in the building of such a mental model [23, 21, 22].

Although software visualization is at least in a reverse engineering context of great importance (as shown by the number of publications on software visualization in the reverse engineering community), this could lead to a detrimental distinction between a forward and a reverse engineering phase. This view is not up-to-date anymore: an evolutionary view of software is taking its place, putting forward a notion of continuous iterative development including tasks such as code editing, refactoring, reverse engineering, and (in a larger context) reengineering.

Of course software visualization tools cannot ignore the current evolutionary/highly iterative view of software, even less so because they could be the key to propagate this view by combining and compressing large amounts of information into simple, yet expressive, visualizations.

Based on the assumption that such an evolutionary view of software will be predominant in the next years or decades, we want to briefly highlight some characteristics and features that software visualization tools of the future should possess to propagate such a view:

Symbiotic relationship with the development environment. The best way to propagate software visualization is to infiltrate existing development environments and complement the existing functionalities. We do not think that standalone software visualization tools would be used extensively, mainly because working people dislike changing their habits: a separate visualization tool introduces a disruptive latency between what one is seeing and what one is editing and/or manipulating. The market, represented by a few million programmers on this planet, will only adapt itself if there are evident technical, cognitive, and ultimately financial benefits provided by the software visualization facilities.

Refactoring support. Code refactoring, originally intro-

duced by Opdyke at the beginning of the 1990s [16], has become an issue in software development since its first mainstream appearance in the book of Fowler *et al.* which comes up with a list of dozens of ways to manipulate object-oriented software [9], most of which can preserve the behavioral semantics of the manipulated software, *i.e.,* it is certain that the system will still work after the manipulations. Such manipulations include renaming a class/method/attribute, pushing up and down methods/attributes from/to a subclass/superclass, transforming temporary variables into instance variables, etc. A few years ago, elegant and powerful implementations of software refactoring engines have made their way into existing development environments such as the Visualworks Smalltalk Refactoring Browser [19, 18] and the refactoring engine plugin of the IDE developed by the OpenSource Eclipse project[1]. It is clear that if a software visualization tool is to become a preferred way of looking at software, the manipulations must be possible as part of the software visualization tool and should be rendered visually, if only by updating the view given by the visualization tool. However, a technical problem is given if the source code of the development environment or the refactoring engine cannot be obtained and understood by the visualization tool makers.

Multi-user support. Complex software systems are being designed and developed by many people concurrently. To support a cooperative view as can be done with collaborative/versioning tools (such as the concurrent versions system CVS[2], Microsoft SourceSafe[3], and VisualWorks Store[4]), a good visualization tool would certainly need to visually render the current point of interest (*i.e.,* the position) of the developers and their most recently changed software artifacts.

Evolution analysis support. Software systems are constantly being evolved (at whatever pace) to cope with new requirements and to integrate bug fixes. The study of the past, present, and future of software systems is the research focus of the expanding and increasingly interesting field of software evolution research [14]. It would be useful for the developers to be able to replay the past lifetime of a class or a group of classes. This could provide an important source of information for decision making. Moreover, it could also help to identify costly parts (*e.g.,* if a class is changed over and over again, it is costing more than other parts in the system) or obsolete parts (*e.g.,* if a class is never changed, it is either dead code or good code). A thorough knowledge of the history of a system represents important information about that system.

Unification of information sources. There is a great spectrum of different sources of information about a software system. Apart from the primary one, the source code itself, one can also take into account documentation, bug reports, comments in the source code, UML diagrams, CRC Cards, user stories, unit tests, etc. Software visualization is an ideal vehicle to unify all these sources into one data pool which can then be visualized. Of course most of these data sources come without a formal definition and must be formalized before they can be integrated into any visualization. A simple example are comments in the source code: in the Java programming language the tool JavaDoc parses the declarations and documentation comments in a set of source files and produces a group of cross-linked HTML pages describing the software artifacts. An attempt to formalize these comments to use them for reverse engineering purposes has for example been proposed by Torchiano [24]. A visualization tool could display the comments for example as tool tips when the point of attention of the viewer is moving around. The benefits of these formalizations is that several of these informal sources of information could enrich the already present visualizations, thus augmenting the amount of information transmitted by them. The technical problems involved with such a unification are not to be underestimated, and even a standardization of the information sources (*e.g.,* with XML [7, 8]) will only solve part of the problems. An example of an open problem is keeping the various sources synchronized.

Spectrum of views. Various software visualization tools visualize software in different ways. Some of the tools propose views residing at different/complementary levels of granularity and visualize also different kinds of information (classes, applications, collaboration, subsystems, etc.). A good software visualization tool should not only propose good/complementary views, but also keep the views synchronized between them and allow the user to easily define new views. This is important in a reverse engineering context where a software visualization tool must be specialized to take case study-specific aspects into account.

Real-world validation. A crucial test that software visualization tools must undergo is certainly the industrial validation: the real world has many challenges, such as scaling up, being able to even parse the system or extract whatever information which can be fed into

[1] See http://www.eclipse.org/ for more information.
[2] See http://www.cvshome.org/ for more information.
[3] See http://msdn.microsoft.com/ssafe/ for more information.
[4] See http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/ for more information.

the visualization tool. Through repeated confrontation with real case studies, one will also remark where the tool still needs general improvements or where there is need for specialization. A simple example of such a specialization are the acronyms often present within class names, which convey hidden semantic information about which subsystem or subarea of the system the class belongs to from the developer's point of view. After repeatedly encountering this kind of information one quickly wants to get an elegant way of modeling and handling it, for example by encoding it into nominal colors.

4 An Example

In this section we want to give a simple example of how some of the previous features can be achieved without a huge effort, although we do not want to minimize the technical difficulties that some of these points involve and which we still did not solve yet.

In Figure 1 we see a screenshot of the VisualWorks Refactoring Browser which we extended to accomodate visualizations provided by our software visualization tool CodeCrawler [11, 12]. In the figure we see a Class Blueprint view [13] of the currently selected class. Moreover we see that all methods selected in the browser (actually a complete method protocol 'private' has been selected) are also selected in the visualization. Furthermore a rename refactoring is being performed on one method of this class. Note that the visualization occupies the space normally used for displaying the method body. However, since during the browsing of the class (*e.g.,* looking for a certain method, method protocol, or attribute) the method body panel remains often empty anyway, this is not such a severe problem. Our implementation (re)uses the refactoring engine of the Refactoring Browser and thus allows us to perform even more complex refactorings like push-up and pull-down of methods or attributes. After the software has changed, our tool gets notified and automatically redisplays an updated view of the software.

The main drawback of our current integration is that between selecting a class or a group of classes and their visualization in the browser takes a few seconds[5]: this latency introduces wait times which disturb the viewer. We have already taken some countermeasures by implementing a cache which yields (the cached visualizations) in less than one second, but our personal experience shows that even small latencies disturb the viewer.

[5] from 2 to 20 seconds, depending on the number of classes and contained methods/attributes, measured on a PPC 500 MHz Apple G4.

5 The Goal?

The future of software visualization can hardly be predicted, we think however to be able to predict a possible and desirable goal that at least some program visualization tools will inevitably try to achieve: *quasi-real-time static software visualization*. Versioning systems like CVS have introduced a novel way of handling source code: they allow us to retrieve any version of any source file ever written by any person. If software visualization is to become the preferred way of developers of looking at source code, we cannot ignore the issue of this quasi-real-time: there must be a dependency mechanism between the versioning tool, the integrated development environment, and the software visualization tool: *e.g.,* as soon as someone changes a part of the system several people are editing and manipulating at once, they must be notified by the change at once. This is also a place where software visualization can fully exploit its potential: notifying the developers of system changes by means of text boxes, dialogs, log files, etc. are all clumsy approaches compared to literally *seeing* the changes happen and the system grow/shrink/change its shape in quasi-real-time.

6 Discussion

Should the goal described above be achieved, the result would be a real-time visual collaborative environment in which software engineers develop a system together, and in which they all have a common view of the system, namely the one proposed by the software visualization tool. This of course makes the achievement of the goal heavily dependent on the quality and the success of such a software visualization tool: should the users (*i.e.,* the software engineers) dislike or disagree with the proposed visualization, they will not accept it as part of their mental model of the system. Therefore, before software visualization tool providers can give it a try at such a long-term vision, they must first cope with the following (and many other) issues:

Human-Computer-Interaction issues. The field of Human-Computer-Interaction (HCI) deals with how humans interact with computer and with how to increase the quality of the interactions. User Interface design [6, 17] plays an important part in this context, and is also an issue in context of software visualization: the better a human viewer can interact with the visualizations, the more (s)he will think that the visualizations are useful. These are largely unsolved issues which can only increase in the context of real-time- or 3D-visualization. Another obstacle are the current input devices we are using to communicate with a computer: a computer mouse is for example

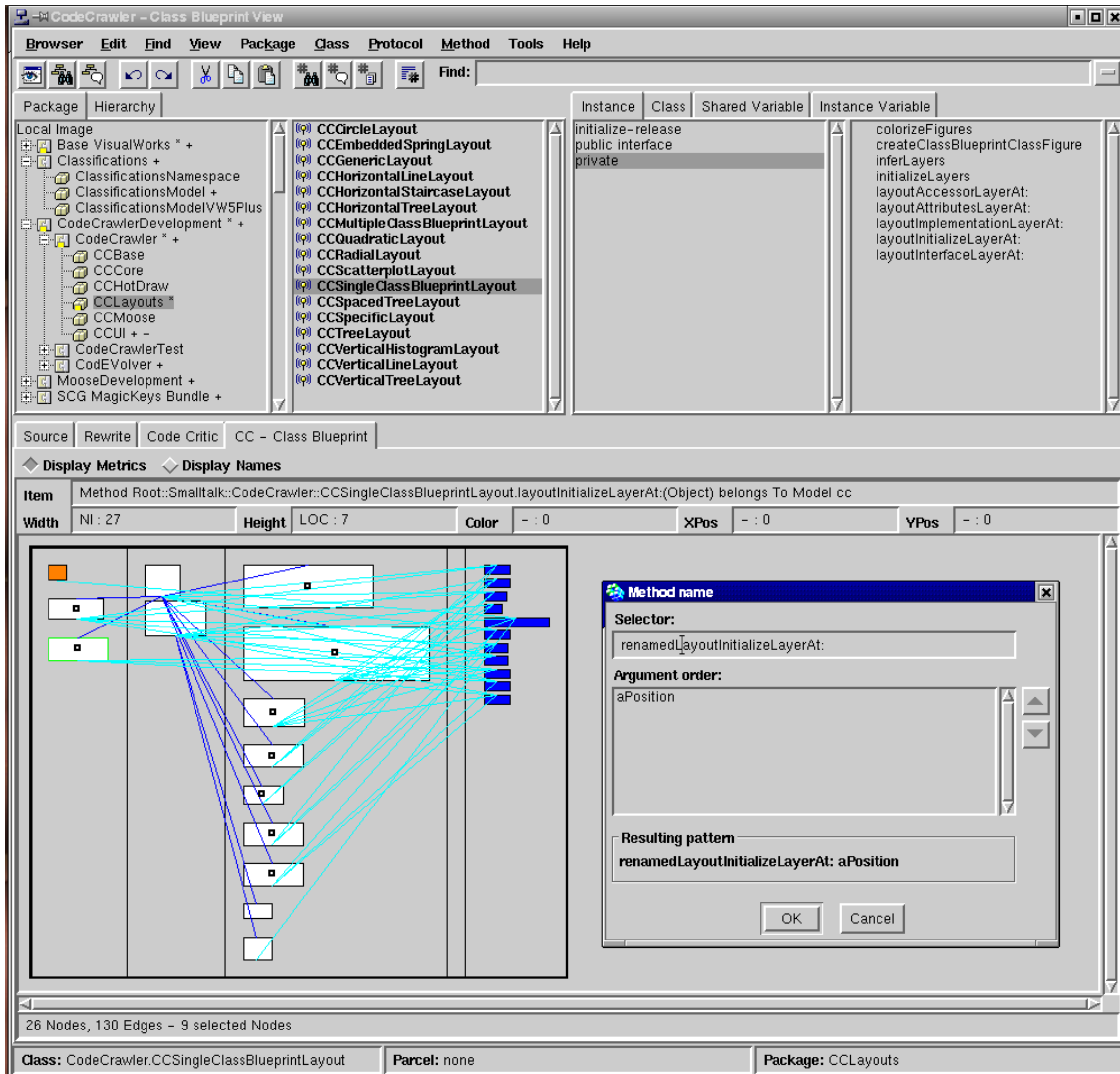


Figure 1. The integration of CodeCrawler with the VisualWorks Refactoring Browser.

not an ideal device to navigate three-dimensional information spaces, trackballs and other gadgets may be more appropriate, but are far less present at people's working place, and therefore not a recognized industry standard.

Scalability issues. Scalability has always been an issue in reverse engineering and reengineering, mainly because the examined subject systems are usually very large and complex. Although the ever-accelerating computer hardware can solve a part of this issue, our own experiments in the field of software evolution have shown that the massive amounts of data (hundreds of versions of systems which contain hundreds of classes and tens of thousands or software artifacts) put a heavy strain on even the fastest hardware. Therefore an important part of the scalability problem must be solved on the software side. Note also that a different kind of scalability problem, a purely visual one, came up during our experiments on software evolution: when a tool visualizes thousands of software artifacts the visualized items either take up too much space (generating navigation problems) or, in order to fit on a display, become too small to be interacted with.

7 Conclusion

New techniques like Design Patterns [10], Code Refactoring [9], and methodologies like eXtreme programming

[1] and Agile Development [5] have changed the way developers see software: we are more and more going towards an evolutionary view of a quasi-living software systems. This view is further amplified by the right tool support like refactoring engines, multi-way browsers, etc. We are convinced software visualization still does not exploit its full potential in such an evolutionary context, on the contrary it is rather being ignored so far. The future challenge of software visualization is thus to prove its value in the tough arena of mainstream (professional) software development.

Acknowledgments

We would like to thank Gabriela Arévalo and Stéphane Ducasse for commenting drafts of this paper.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] J. Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[3] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization - Using Vision to Think*. Morgan Kaufmann, 1999.

[4] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[5] A. Cockburn. *Agile Software Development*. Addison Wesley, 2001.

[6] A. Cooper. *About Face - The Essentials of User Interface Design*. Hungry Minds, 1995.

[7] S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) version 1.0 - w3c proposed recommendation 20 december 2000. Technical Report PR-xlink-20001220, World Wide Web Consortium, Dec. 2000.

[8] e. a. Didier Martin, Mark Birbeck. *Professional XML*. Wrox Press Ltd., 2000.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[11] M. Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409 – 418. IEEE Press, 2003.

[12] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, may 2003.

[13] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[14] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[15] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98, 1996.

[16] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[17] J. Raskin. *The Humane Interface*. Addison Wesley, 2000.

[18] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[19] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[20] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.

[21] M.-A. D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, dec 1998.

[22] 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. *Journal of Software Systems*, 44:171–185, 1999.

[23] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

[24] M. Torchiano. Documenting pattern use in java programs. In *Proceedings of ICSM 2002 (International Conference on Software Maintenance)*, pages 230–233. IEEE Computer Society, IEEE Press, 2002.

[25] E. R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[26] E. R. Tufte. *Visual Explanations*. Graphics Press, 1997.

[27] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[28] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

Position Paper: Challenges in Visualizing and Reconstructing Architectural Views

Juergen Rilling
Concordia University,
QC, Canada, H3G1M8
rilling@cs.concordia.ca

Michel Lizotte
Defence R&D Canada
QC, Canada, G3J 1X5
Michel.Lizotte@drdc-rddc.gc.ca

Abstract

A common approach to cope with software architecture comprehension is to provide higher levels of abstraction of lower level system information. Architectural recovery tools provide such high-level views by extracting and abstracting a subset of the software entities. In this research we are focusing on challenges in visualizing and reconstructing architectural views. In particular we are looking into issues related to the applicability of current visualization representations generated by architectural recovery tools to support views and products specified by the C4ISR architecture framework.

1. Introduction

One aid to improve the understanding of large programs is to reduce the amount of detail a programmer sees by using a higher level of abstraction to represent a program. Over the last decade, programs became larger and more complex, causing new challenges to the programmer in visualizing these complex and large source code structures. Different techniques and approaches have been developed and validated with users. However, providing different levels of abstraction might not be sufficient since users might be still dealing with a large amount of information and data. Not every visualization technique is equally usable in displaying a particular dataset. The visualization technique might lack an appropriate navigation support or may not allow the effective reduction of the amount of information displayed through a choice of distinct views.

Software visualization can be described as analyzing a subject system (a) to identify the system's components and their interrelationships, (b) to create representations of a system in another form at a higher level of abstraction and (c) to understand the program execution and the sequence in which it occurred. It would be ideal to be able to simultaneously view and understand detailed information about a specific activity in a global context at all times for any size of program. As Ben Shneiderman explains in [12], the main goal of every visualization technique is: "Overview first, zoom and filter, then details on demand". This means that visualization should first provide an overview of the whole data set then let the user restrict the set of data on which the visualization is applied, and finally provide more details on the part the user is interested in. Software visualization of source code can be further categorized in static views and dynamic views. The static views are based on a static analysis of the source code and its associated information and provide a more generic high-level view of the system and its source code. The dynamic view is based on information from the analysis of recorded or monitored program execution. Based on their available run-time information, dynamic views can provide a more detailed and insightful view of the system with respect to a particular program execution. As Mayhauser [9] illustrated, dynamic and static views should be regarded as complementary views rather than being mutually exclusive. Users tend to apply an opportunistic approach, using both static and dynamic views to achieve a specific task. The software visualization techniques used by recovery tools are in most cases a carry over from the more traditional reverse engineering tool domain. With the majority of tools providing support for UML visualization based techniques or procedural orientated visuals, like call-graphs, tree structures. Ideally, the high-level views provided by these tools should be organized in a hierarchical/layered fashion, allowing users to navigate through different layers of abstraction.

Software Architecture

Software architecture has been defined as a structure composed of components and rules characterizing the interaction of these components [13]. In [11] it has been defined as elements, form, and rationale. Another definition is presented in [6] where it was defined as components, connectors, and configurations [6]. C4ISR AF is using a definition, not limited to software, based on the IEEE STD 610.12 and established by the DoD

Integrated Architecture Panel in 1995 [7]. They define *"architecture"* as *"the structure of components, their relationships, and the principles and guidelines governing their design and evolution over tim*e." One of the earliest definitions of software architectures, by Perry and Wolf [6], has remained one of the most insightful.

Architecture Recovery

Architecture recovery can be seen as a discipline within the reverse engineering domain that is aimed at recovering the software architecture of a system [2]. It can be described as the process of recovering up-to-date architectural information from existing software artefacts [2, 16]. The rational of system architectural recovery and comprehension is to provide reasoning behind the software architecture or high-level system organization of a system. There may be little or no documentation available and the documentation that does exist probably does not resemble the current system due to drift and erosion [3]. The application of system understanding tools goes beyond mere object identification - it includes a generation of (interactive) documentation, quality assessment, and introducing novice programmers to a legacy application. Architectural recovery is motivated by (re)generate coherent abstractions of existing systems to guide analysts during the comprehension of large existing systems and to provide some reasoning about the system architecture.

Motivation

The presented research is conducted under a project of the Defense Research and Development Canada (DRDC) at Valcartier. The focus of this project is the visualization support for the various products described in the US Department of Defense (DoD) Architectural Framework (AF), better known as the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) Architecture Framework (AF) [10]. As part of this research, we extended a previously performed survey of current reverse and architectural recovery tools, with a focus on visualization support for C4ISR AF, its views and products. Tools should provide adequate visualization support, by providing on the one hand users with views and information abstraction that are beneficial for the recovery process, as well as visualization techniques that are required by architectural frameworks to document the architecture.

The remainder of this article is organized as follows. Section 2 introduces provides a brief overview and background C4ISR architectural framework. Section 3 maps and discusses the applicability of the surveyed tools to the C4ISR AF. Section 4 provides a discussion about challenges and pitfalls of current visualization techniques in supporting architectural views.

2 The DoD Architecture Framework

The purpose of the DoD AF is to improve capabilities by enabling the synthesis of requirements with sound investments leading to the rapid employment of improved operational capabilities, and enabling the efficient engineering of warrior systems. This framework formerly called the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) *Architecture Framework* [10] is intended to ensure that the architecture descriptions developed by the Commands, Services, and Agencies are inter-relatable between and among each organization's operational, systems, and technical architecture views, and are comparable and able to integrate across Joint and combined organizational boundaries. It provides the rules, guidance, and product descriptions for developing and presenting architecture descriptions that ensure a common denominator for understanding, comparing, and integrating architectures. This section is based on the C4ISR Architecture Framework (Version 2.0 as published by the AWG)

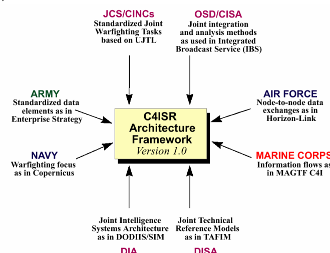


Figure 1: C4ISR Architecture Framework

The *operational architecture view* is a description of the tasks and activities, operational elements, and information flows required to accomplish or support a military operation. It contains descriptions (often graphical) of the operational elements, assigned tasks and activities, and information flows

The *systems architecture view* is a description, including graphics, of systems and interconnections. For a domain, the systems architecture view shows how multiple systems link and interoperate, and may describe the internal construction and operations of particular systems within the architecture.

For an individual system, the systems architecture view includes the physical connection, location, and identification of key nodes (including materiel item nodes), circuits, networks, warfighting platforms, etc., and specifies system and component performance parameters (e.g., mean time between failure, maintainability, availability). The systems architecture view associates physical resources and their performance attributes to the operational view and its requirements per standards defined in the technical architecture.

The *technical architecture view* is the minimal set of rules governing the arrangement, interaction, and interdependence of system parts or elements, whose purpose is to ensure that a conformant system satisfies a specified set of requirements.

In what follows, we present a case study based on a survey of 23 architectural recovery and reverse engineering tools (see appendix) that was performed as part of this project and map their capabilities in supporting the visualization products described in the C4ISR system view. The other two views described in the C4ISR, the operational and technical view were not considered in this survey, since these views are mostly based on domain knowledge, rather than information that can be recovered by analyzing program artifacts.

3. Case study – C4ISR Capability matrix

The motivation for the presented case study and the resulting C4ISR visualization support capability matrix are two-fold. The first objective was to analyze the current state of the art support of architectural views and visualization techniques provided by recovery tools and their applicability in support for the different visualization products described in the system view of the C4ISR architectural framework. Secondly, the resulting capability matrix can serve as guidance for directing future research, by addressing shortcomings of current tools.

Visualization techniques supporting system view products

The system view products described within the C4ISR architecture framework suggest certain visualization and diagrammatic techniques that should be provided to document an existing architecture. One intend of the C4ISR AF was to guide tool developers by providing templates for suitable/expected visualization and representation techniques, to support the various system view products. The suggested templates are not compulsory and can be replaced by other visualization techniques. There is a currently a tendency in applying the standard UML notations to document software architectures within the C4ISR framework. This approach has both advantages and disadvantages.

Advantages can be found in using a well-known standard notation, in reducing the learning overhead that might be caused by introducing new visualization techniques and their notations. Furthermore, over the last several years, UML established itself as a viable approach for documenting various aspects of the requirement, specification and design phase

One of the major disadvantages of the UML standard notation is its limited expressiveness with respect to architectural aspects. Firstly, its notation does not provide enough expressive power to describe the specific requirements of architectural artifacts. Secondly, the levels of abstraction provided by UML might not be sufficient to provide some of the required views.

The open framework approach of the C4ISR AF with respect to visual representations encourages tool developers to explore new avenues and derive new visualization techniques that might lead to more intuitive and architectural specific representations. In particular tool developers are facing during architectural recovery additional challenges having no or only limited domain knowledge available to derive the visual abstractions.

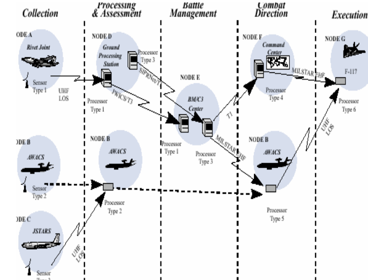


Figure 2: System interface description

Figure 2 and 3 illustrate this situation, with figure 2 abstracting the system interfaces in a high-level view (using a non UML notation), which can easily be understood by both novice and experts. Comparing this with the UML view of the system interface description (Figure 3), the differences in both the capabilities, abstractions and applicability of the visualization becomes evident.

The following are some of the visualization techniques templates described in the C4ISR standard document that should be created to document system view specific products.

With current recovery tools focusing on the structural analysis of existing system artifacts, one of the challenges can be found in the reconstruction of visual abstractions is their lack of domain knowledge. Figure 2 is an example for domain knowledge based visualization. The graphic requires not only specific annotations, but also domain specific representations of the objects (e.g. different types of airplanes) involved in the system and their intercommunication.

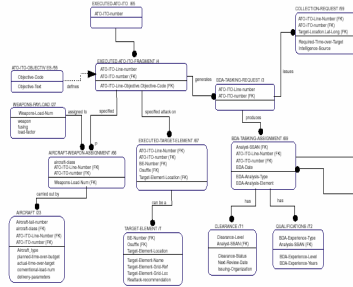


Figure 3: System interface description (UML based)

Figure 3 on the other hand is based solely on structural analysis through lexical and semantic parsing of existing system source code. This information can almost completely automatically be extracted, without any prior domain knowledge.

Some other visualization challenges include the support for building traceability matrixes. These traceability matrixes are an essential part of architectural documentation and re-documentation not only within the C4ISR architectural framework but also within other frameworks (e.g. Zachmann). Matrixes are used widely by the following products within the system view (C4ISR):

System Performance Matrix: Depict current performance of each system, and the expected or required performance characteristics at specified times in the future (soft and hardware).

Operational Activity to System Function Traceability Matrix: Maps operational activities to system functions in the form of a matrix (Figure 4)

	GCCS	MCS/P	FBCB2	M1A2 SEP	M2A3	ASAS	CGS	GBCS	IMETS	IREMBAS
GCCS		●								
MCS/P	●		●	●	●	●				
FBCB2		●		●	●	●				
M1A2 SEP		●	●		●	●				
M2A3		●	●	●		●				
ASAS	●	●	●	●	●		●	●	●	●
CGS						●				
GBCS						●				
IMETS						●				
IREMBAS						●				
AFATDS	●	●	●	●	●	●				
BFIST			●	●	●					
Paladin										
FAAVS										
MLRS										
FAADC3I	●	●	●	●	●	●				
Avenger										
BSFV-E										
GBS										
CSSCS		●	●			●				
SAMS										
SAAS										
SPDS-R										
DAMMSR										
ULLS										
....										
....										

Figure 4: *Operational Activity to System Function Traceability Matrix*

System Information Exchange Matrix: Shows the data exchange among nodes in different systems in the form of a matrix.

Systems Matrix: The product focuses on the flow of data among system functions, and on the relationships between systems or system functions and activities at nodes.

Behavioral modeling

Within the C4ISR architectural framework the importance of documenting and being able to trace the dynamic and behavioral system aspects is reflected by the following system behavior modeling products.

? *Systems Rules Model:* A rule base for actions occurring as part of the trace. The rule base applies for the different visualization techniques within the system activity product.

? *Systems State Transition Description:* State transition descriptions describe system responses to sequences of events. Events may also be referred to as inputs, transactions, or triggers (Figure 5).

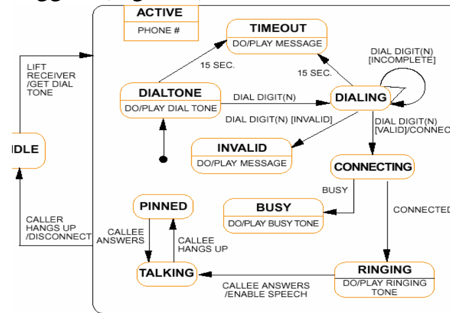


Figure 5: *State transition diagram*

? *Systems Event/Trace Description:* The system event trace describes the timing and behavior (based on the rule model) between nodes, as well as the interaction among these nodes. The standard UML sequence diagram notation can be applied to capture the behavior (Figure 6).

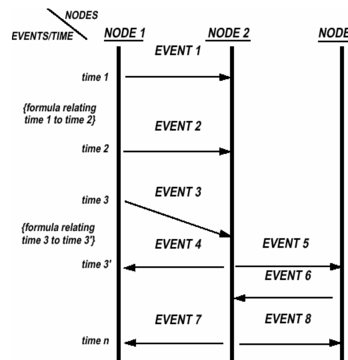


Figure 6. Modeling dynamic behavior within the C4ISR AF

Physical Data Model: Describes the physical implementation of the logical data model from an operational view point. The product is supported in the form of standard E/R diagrams, etc.

Capability matrix

Table 1 shows a capability matrix (based on the survey of 23 architectural recovery and reverse engineering tools) and maps their overall capability to the products and their visual representations as described by the C4ISR architecture framework. The matrix provides a general summary of the overall tool capabilities rather than focusing on the specifics of a particular tool.

System view product	*Visualization Support*
System Performance Parameters Matrix	Partially
Systems Functionality Description	Partially
Operational Activity to System Function Traceability Matrix	No
System Information Exchange Matrix	No
System Interface Description	Partially
Systems Communications Description	Partially
Systems Matrix	Partially
System Evolution Description	No
System Technology Forecasts	No
Systems Rules Model	No
Systems State Transition Description	Partially
Systems Event/Trace Description	No
Physical Data Model	Fully

Table 1: Visualization Capability

Partial visualization support is achieved if at least one or more tools provide capabilities required by the particular system view product. The capabilities are often limited and do not exist, because of a lack of domain knowledge, that is necessary to re-create these views and products.

4. *Discussion: Challenges and Pitfalls*

Larger software systems place an enormous cognitive load on users and humans are limited in the density of information they can resolve and comprehend [5,8]. Visualization facilitates the discovery of new science by revealing hidden structures and behaviours in model output. It is in the areas of insight and understanding that visualization plays a central role [8]. Many reverse engineering tools have been built to help the comprehension of large software systems. Software visualizations are one approach being investigated worldwide to provide some assistance in program understanding. It should be recognized that visualization is a complementary technique and is to be used in conjunction with other program understanding techniques such as software inspection, metrics, static and dynamic source code analysis, etc.

Throughout a software product's life cycle, many different people are responsible for understanding the design details of the software code. Learning the structure of code developed by others is especially time consuming and effort intensive during the software maintenance phase. From an architectural recovery perspective the challenges becomes even more aggravating, because the maintainer has to create a mental model of a larger system that might include several subsystem and the interaction among these subsystems.

One of the shortcomings of current architectural recovery tools is their lack of supporting architectural views and abstractions. Architectural views require often notations other than the ones provided by current reverse engineering tools In particular traditional visualization techniques are limited by their available notations and their ability to map between visualization elements and architecture components (e.g. throughput, dynamic linked information, etc.). Other factors are the lacking support for architectural views that match the more traditional architectural views (e.g. 4+1 or C4ISR AF). The creation of architectural views requires often additional user domain knowledge, architectural design decisions and analysis support in form of grouping/clustering have also to be considered.

The majority of the surveyed tools focus on the visualization of static system structures rather than dynamic interaction aspects. System architectures are often based on distributed and dynamic systems that take run-time behaviour into consideration. In particular the mapping of these dynamic architectural aspects to the static visualization techniques is often difficult, because these techniques do not support natively graphical notations for representing these dynamic aspects. Examples are their lack of support for e.g. remote connectors, throughput, performance, resource requirements, etc. Furthermore, in visualizing architectures there exists an explicit need for views and visualization techniques that are based on dynamic tracing and profiling aspects. This aspect are addressed and acknowledged for example by the System Activity Sequence and Timing product in the C4ISR Architecture Framework. For

architectural recovery tools to be able to manage and display dynamic behaviour, often a large amount of data has to be processed. Additionally, the tools have to facilitate notations that support the visualization of these dynamic aspects.

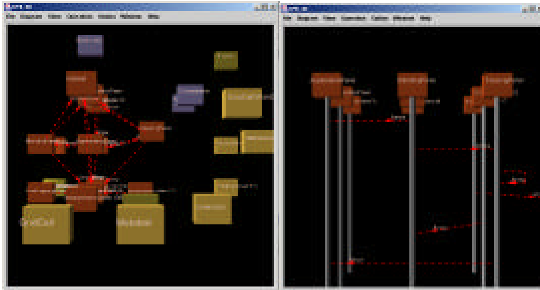


Figure 7 Moving to 3D worlds

Furthermore visualization techniques should take advantage of 3D [5], virtual reality [8], multimedia to provide intuitive and meaningful representations of the underlying architectural structure, its behaviour and relationships. In the context of the C4ISR framework there are further needs to provide views that combine system views with operational views, as well as the technical with the system view. Feature extraction and concept analysis techniques have to be integrated to facilitate this. Clustering and grouping requires application–specific data and domain knowledge, as well as source code analysis techniques. It is important to note that clustering can be used for functions such as filtering and search. Scripting support is also essential to create abstract views on the underlying repository

Different levels of granularity are required, often not facilitate in the current tools, e.g. UML does not provide enough meaningful levels of abstraction. Navigation and context switching has to be further improved to help the architects and maintainers to navigate through the recovered information.

6. References

1. Ball T., Eick Stephen G., “Software Visualization in the Large”. *IEEE Computer* 29(4): 33-43 (1996).
2. O'Brien, L., Stoermer, C., Verhoef, C. 2002. Software Architecture Reconstruction: Practice Needs and Current Approaches ; CMU/SEI-2002-TR-024 ADA407795
3. Clements, P., Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. 2002. Documenting Software Architectures: Views and Beyond. Boston, MA: Addison-Wesley.
4. Deursen van A. 2002.Software Architecture Recovery and Modelling. ACM Applied Computing Review 10(1):4-7.
5. Feijs, L.M.G. & de Jong, R.P. 1998. 3D Visualization of Software Architectures. Communications of the ACM 41, 12 (December 1998): 73-78.
6. Garlan, D. and M. Shaw, An introduction to software architecture, in: V. Ambriola and G. Tortora, 1993, Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, 1993 pp. 1--39.
7. Institute of Electrical and Electronics Engineers. IEEE Std 1471-2000. Piscataway, NJ: IEEE Computer Press.
8. Knight C., Munro M., 2001. Visualising the non-existing”, IASTED International Conference: Computer Graphics and Imaging, Hawaii, USA..
9. Mayrhauser A., A. M. Vans, “Program Understanding Behavior During Adaptation of Large Scale Software”, *Proceedings of the 6th Intl. Workshop on Program Comprehension*., IWPC ‘98, pp. 164-172, Italy, June 1998.
10. Office of the Secretary of Defense Working Group. 1997 C4ISR Architecture Framework, Version 2.0. Washington, DC.
11. Perry D. E. and Wolf A. L. 1992, Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17:40--52, October 1992.
12. Shneiderman, Ben, “Tree Visualization with Tree-Maps: A 2-D Space-Filling Approach”. In *ACM Trans. of Computer-Human Interaction*, vol. 11, no. 1, 1992, pp. 92-99.
13. Shaw M. and Garlan D. 1996. Software architecture: Perspectives on an emerging discipline, Prentice-Hall.
14. Sneed, H. M. 1998.Architecture and Functions of a Commercial Software Reengineering Workbench. 2-10. Proceedings of the Second Euromicro Conference on Maintenance and Reengineering. Florence, Italy, March 8-11. Los Alamitos, CA: IEEE Computer.
15. Storey M.-A., Fracchia F. and Müller H.., “Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration, *Journal of Software Systems*, special issue on Program Comprehension, v 44, pp.171-185, 1999
16. Trevors A. and Godfrey M.W., 2002. Architectural Reconstruction in the Dark, Position paper, Workshop on Software Architecture Reconstruction collocated with WCRE '02, Richmond, VA, October 2002

Appendix A: Tool survey

1. Argo/UML : http://argouml.tigris.org/servlets/ProjectSource
2. Bauhaus: http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/
3. CIAO http://www.research.att.com/~ciao/
4. CodeCrawler: http://www.iam.unibe.ch/~lanza/CodeCrawler/codecrawler
5. CodeSurfer:http://www.grammatech.com/home/index.htm
6. Columbus/CAN : http://www.frontendart.com/
7. CONCEPTwww.cs.concordia.ca/CONCEPT
8. The Dali Architecture Reconstruction Workbench. http://www.sei.cmu.edu/ata/products_services/dali
9. Fujaba: http://www.uni-paderborn.de/cs/fujaba/
10. G^{SEE}http://www-adele.imag.fr/~jmfavre/GSEE/
11. Headway: http://www.headwaysoft.com/index.htm
12. Imagix4Dhttp://www.imagix.com/index.html
13. KLOCworkinSight. www.klocwork.com/products/inSight.
14. ManSARThttp://www.mitre.org/pubs/edge/january_98/first
15. Rational http://www.rational.com/index.jsp
16. Red Hat Source-Navigatorhttp://sourcenav.sourceforge.net/
17. Refine/C Illuma: http://www.frontendart.com/
18. SniFF++: http://www.takefive.com/
19. SoftArch: http://www.cs.auckland.ac.nz/~john-g/projects.html#softarch
20. Soloway E. and Ehrlich K.,1994. Empirical studies of programming knowledge, IEEE Transactions on Software Engineering, SE-10, 595--609 (1984).
21. SWAG tool kit: http://swag.uwaterloo.ca/pbs/
22. Understand for C++: http://www.scitools.com/ucpp.html
23. Visual Paradigm :http://www.visual-paradigm.com/index.php

Visualization to Support Version Control Software: Suggested Requirements

Xiaomin Wu
University of Victoria
xwu@uvic.ca

Margaret-Anne Storey
University of Victoria
mstorey@uvic.ca

Adam Murray
University of Ottawa
amurray@site.uottawa.ca

Rob Lintern
University of Victoria
rlintern@uvic.ca

Abstract

Many version control systems have been developed to manage both software version history and associated human activities with the intent of producing higher quality software. To better understand and explore the vast information these version control systems portray, several approaches have been conducted to apply visualization techniques in this domain, resulting in a variety of tools. However, these tools have rarely been evaluated and hence we are unable to tell how successful these information visualization techniques are for understanding and exploring version control information. Moreover, there is lack of requirements for how such a visualization tool can support version control activities. This paper describes a set of requirements for visualization support in a version control tool. We also present a tool called Xia, which was developed for the navigation and exploration of software version history and associated human activities. Moreover, we conducted an exploratory user study to test if the functionality of the Xia tool meets these requirements and if there are requirements we missed -- the results are documented. This position paper ends with a question – how should we proceed next with our research? We intend to refine the requirements and seek directions for future exploration.

1. Introduction

Version control systems are becoming an increasingly important tool for software development projects, especially when the development tasks are performed in a team environment. Presently, most medium to large-scale software projects are developed in association with a version control tool. A large amount of information is generated and stored in the repositories of these version control tools. What does this information mean to the software development process? Can this information be used in a meaningful way to help with team work? If so, how does the presentation of this information assist software development? To answer these questions, we conducted a preliminary survey of five version control systems in the spring of 2002. In this survey, we posed questions related to the functionality and ease of use of version control systems, as well as what data is important to support team collaboration. The results of this survey highlighted that although the features of version control systems are considered adequate; the interfaces of these systems are not satisfactory for users to understand and explore version control information. Also, our survey demonstrated that the most prominent concerns related to a team development task include:

- What happened since I last worked on the project (types of events, such as new file added, file modified, etc.)?
- Who made this happen?
- Where did this take place (location of the new file, change, deletion, etc.)?
- When did this happen?
- Why were these changes made (what is the rationale of the designer(s) who made the change)?
- How has a file changed (exact details of the change, as well as relationship to other files)?
- What is the history of a particular file?

We name this problem the "5W+2H" problem for brevity, referring to the 5W's, what, who, where, when, why, and 2H, how, history, above. When many of the 5W+2H questions remain unresolved, developers may feel like they are working in a void, and progress will be greatly hindered. More importantly, if the 5W+2H problem is not properly addressed, we cannot explore how people work in teams on software projects. When this problem is related to the entire software project, participants in our survey stated that they would like to have an overall view of the entire dataset. They believe that an overall view showing data related to all people's work would enable them to collaborate even better. This is because they will have more awareness of each other's activities and how other people's work may be relevant with their own work.

On the basis of these findings, we conjectured that applying information visualization techniques to a version control system might resolve these problems. Consequently, we investigated related research and noticed that some approaches have been deployed as in the following projects: Seesoft [1, 7], Beagle [17], CVS Activity Viewer [4], and others. These tools used some kind of visualization and query mechanisms to display and explore data from version control systems. However, these tools don't provide requirement analysis and have rarely been evaluated and hence we are unable to tell how

successful these visualization techniques can be for assisting people in understanding and exploring version control information.

In this position paper, we present Xia, a version control visualization tool, which is tightly integrated with a full-featured IDE, Eclipse [5]. In Xia, advanced visualization techniques can be used for browsing and interactively exploring the data in a CVS repository. A preliminary user study was also conducted to evaluate both the requirements we identified through the survey and to test if the tool satisfactorily meets these requirements

Section 2 introduces our approach to the design and implementation of Xia. The details and results of our user study are described in Section 3. In Section 4, we outline future work and pose questions about how to improve our requirements and further evaluate our tool.

2. Approach

In our approach, we elected to focus our tool on the version control system known as CVS [3]. CVS is freely available open-source software that is widely used. We believe the widespread user base will make it easier to evaluate the effectiveness of our tool, as users will be easier to find. Our previous experience [8, 10] of plugging a visualization tool, SHriMP [14], into the Eclipse platform [6], encouraged us towards an approach of using the Eclipse platform as a framework for the integration of:

(1) The Eclipse CVS plug-in, a CVS interface plugged into the Eclipse platform, through which the CVS repository information could be accessed and retrieved;

(2) The Eclipse JDT (Java Development Tools) plug-in, which provides the workspace information for a particular Java project and;

(3) The SHriMP visualization engine, a domain-independent information visualization tool developed at the University of Victoria.

Xia is the result of an integration and customization of these components. The Eclipse CVS plug-in [9] and the JDT plug-in serve as data backends for Xia. The SHriMP visualization tool is customized and used by Xia as a visual front-end for the back-end data. Figure 1 illustrates the architecture of Xia.

In the following subsections, we look into the data we obtained from a CVS repository, and describe how we design the visualization in our tool to help answer the questions we raised before.

2.1. Data acquisition and analysis

The CVS repository is a good resource of information for helping to resolve the 5W+2H questions. We believe that pertinent information can be obtained and visualized. For instance, the log message in CVS contains the record of each commitment, including:

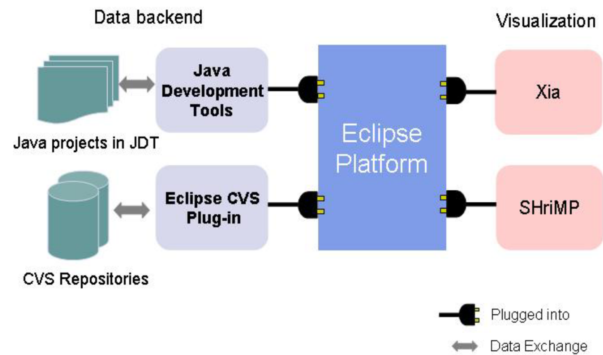


Figure 1. The architecture of Xia

- The author *who* made the commitment;
- The comments made by the author of *what* was changed and hopefully *why* it was changed; and
- The time and date *when* the file revision was created.

To understand *how* a change occurs, we propose the diff function of CVS helps. The location of the changed file in the repository hierarchy helps determine *where* the change takes place. In our tool, the information we required was retrieved directly from both the CVS repository via the Eclipse CVS plug-in, and the JDT in Eclipse, or the information was calculated from the retrieved data.

By analyzing data from the CVS repositories, we classified data into two categories: the software artifacts, including files, folders, and other code-level entities, and associated revision attributes. We attached the following attributes (see Table 1) to each of the file revision.

Table 1. File revision attributes

Attribute Name	Data Resource	Data type
File revision number	Retrieved	Ordinal
File revision tags	Retrieved	Nominal
Date of last commitment of a file revision	Retrieved	Ordinal
Author who changed the file most recently	Retrieved	Nominal
Author who changes the file most times in a particular time period	Calculated	Nominal
Comments associated with each commitment	Retrieved	Nominal
Number of changes associated with a file revision	Calculated	Ordinal
History of a file	Retrieved	Nominal

These attributes reflect human activities that concern people in answering the 5W+2H questions. They have

been classified into two categories according to their data types, nominal and ordinal. Nominal attributes are strings whereas ordinal attributes have numeric or ordinal values.

2.2. Visualization

In this section, we describe the visualization of software artifacts and associated version attributes. Then we outline our method of interactively exploring this information. Finally, we summarize how our visualization techniques were designed to answer the 5W+2H questions.

2.2.1. Visual representation of CVS artifacts. A single file revision in the CVS repository is mapped to a single node in SHriMP. Likewise, a folder containing file revisions is mapped to a parent node of file revision nodes.

In the Eclipse CVS plug-in, software in the repository is displayed in a tree-like hierarchical structure of folders and file revisions. This structure corresponds well to nested graphs in SHriMP, as illustrated in a screen shot of the CVS data in Fig. 2. In Fig. 2, parent folder nodes (shown in purple) encompass file revision nodes (shown in yellow). The outmost blue nodes represent two versions of the same project. Node size relates to the size of the content within the node, so the version on the left (which is graphically larger) contains more sub-nodes than the version on the right..

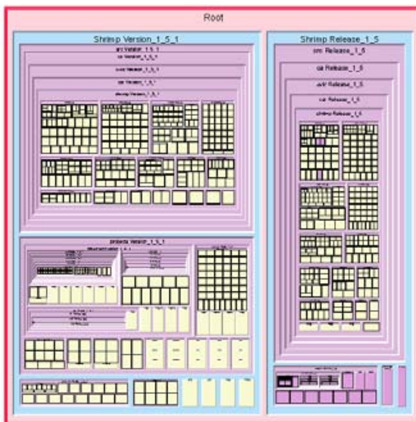


Figure 2. A nested graph showing two versions (in blue) of a software project.

2.2.2. Visual representation of attributes. In addition to the pre-existing SHriMP visualization techniques, we developed an **Attribute Panel** for showing and querying attributes associated with file revisions. The attribute panel concept was originally developed at the University of Maryland, and combined with the Treemap visualization tool [11].

Appropriate visual variables are used to display the attributes [18]. For instance, using color, intensity, tool tips, size, and position to highlight nodes and accentuate their differences. Visual variable values are triggered through the **Attribute Panel**, as illustrated in Fig. 3.

Tool tips provide instant messages that are easily perceived during browsing. In Xia, all attributes in the CVS domain can be viewed with tool tips. The user selects which of the attributes to show in the tool tips.

Colors may be used for both nominal and ordinal attributes, though different color schemes are necessary for each type [2]. For nominal attributes, each of the values may be assigned a distinct color; whereas ordinal attributes may use color intensity instead of different colors (see Fig. 4). In Xia, the date of last commitment and number of changes are the two ordinal attributes that can be visualized using color intensity. These ordinal attributes are sorted in an old-to-new and few-to-more order respectively, and then each value is assigned an intensity of green (the default color). In our example, a more recent date is assigned a brighter green color. Figure 4 shows two screen shots of coloring nodes according to their nominal and ordinal attributes.

The arcs between nodes enable people to focus on a specific task and keep track of its relationship to other files in the project. This kind of awareness is very important for teams to collaborate on work effectively, especially if there are many dependencies between the different artifacts that are being worked on.

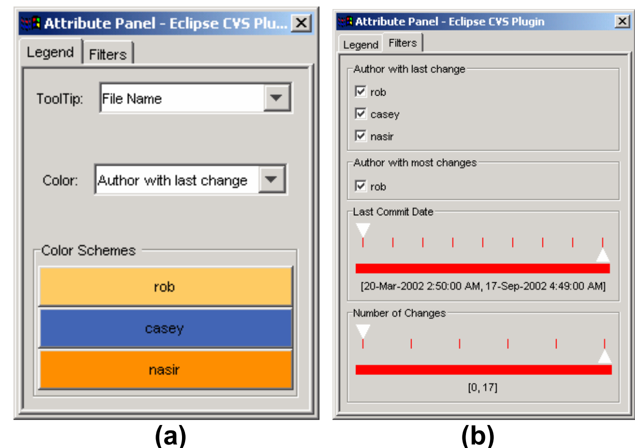


Figure 3. In (a), the user can change colors for the developers, and change how the tool tips appear; In (b), checkbox and double slider filters are used to filter nodes by their attribute values

2.2.3. Interactive exploration. The **Attribute Panel** also supports dynamic exploration using filters. Two kinds of filtering widgets have been developed for different attribute types. A checkbox filter, as illustrated in Fig. 3b is created for each of the nominal attributes. A checkbox filter consists of a set of checkboxes associated with each of the attribute values in the domain. An unchecked checkbox results in the corresponding nodes having equal

attribute values being filtered from the screen. The other filtering widget, a double slider, is designed for ordinal attributes and is especially useful for dynamic queries. A double slider allows the user to select a range of values for query by adjusting the minimum and maximum value of the slider, and can also be used to select a single value by setting the minimum and maximum value of the slider to the same value. We implemented the double slider in Xia to filter two ordinal attribute values: Date of last commitment and Number of Changes. These two sliders could be used together to perform a multi-variable query. For example, if a programmer wants to look at the file that changed most frequently in the past week, he/she would be able to get the result by setting the Date of Last Commitment slider to the corresponding range, and setting the Number of Change slider to its minimum and maximum value.

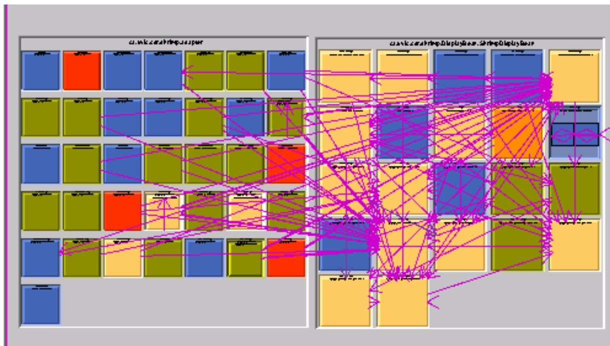


Figure 4(a). The color of each node represents the author who made the latest change

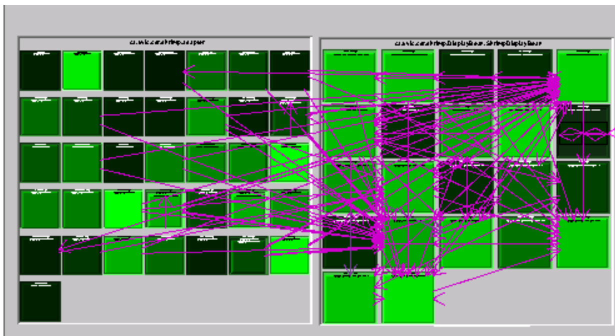


Figure 4(b). The color intensity of each node is determined by the date of the latest commitment.

2.2.4. Ordered Treemap layout. To provide a view that addresses the 5W+2H problem at the entire project level, we adopted the Ordered Treemap algorithm created by the HCI lab at the University of Maryland [12]. Two distinct features of the Treemap layout are the variation of the node size according to the associated numerical attribute and the repositioning of nodes according to their associated ordinal value. Figure 5 demonstrates a screen shot in which node size was adjusted according to the

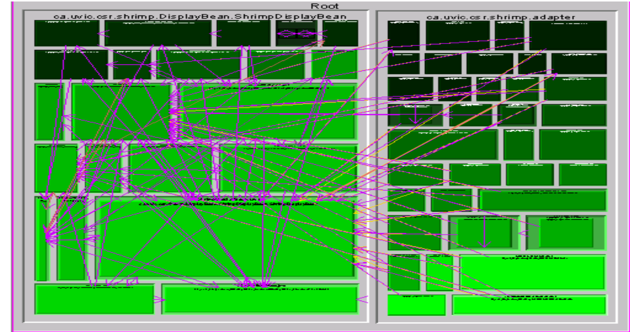


Figure 5. An ordered Treemap layout.

number of changes and the position of nodes are ordered by their last commit date. This ordering feature provides a comparable view for files in a project, hence answering the *When* question at the project level.

2.2.5. Summary of visualization features for CVS. Xia provides various ways to visualize data or derived data from the CVS repository. In addition, relationships between files can be determined using information extracted from the Eclipse JDT plug-in.

Table 2. Map of visualization techniques to questions of interest when working with CVS

Question	Visualization techniques
What	The name of the changed file can be shown using labels on the nodes (which are visible when you zoom in), or they can be shown using tool tips when the user brushes over nodes in the graph with a mouse.
Who	Can be distinguished using different node colors; filter by name using a checkbox. Tool tips could also be used to show the author's name.
Where	Nested within relevant folders in the layouts
When	Date can be shown using color intensity, tool tips. File revisions can also be filtered by date
Why	Rapid access to the code, CVS comments (in the attribute table) and documentation (by right clicking on the nodes, or by zooming in to an embedded view)
How	Access to the code, Javadoc and CVS comments by zooming on a file revision node. Tool tips, intensity, size and location could also be used to show number of changes to a file. Relationships between files are shown using arcs, which could be used to trace the impact of changes.
History	Access to a history panel by zooming on a file revision node and the use of right mouse menu. A table contains the revision history of the file is displayed.

We conjecture that the 5W+2H questions can be answered by interacting with the features in Xia which includes the double slider filters, the different layout algorithms (such as the Treemap layout) effecting size and order of the nodes, the checkbox filters, tool tips, color and intensity. In addition to these features, Xia provides easy access to the source code, documentation (Javadoc) and comments in the CVS repository. The user can zoom into a node representing a file revision and switch between these different views. In Table 2 we summarize how these different features can be used to answer the 5W+2H questions.

3. Evaluation

We conducted an exploratory user study to test both the initial requirements we discovered through the survey and the functionality and usability of Xia. As very few studies have been conducted in the field of version control visualization [19], we consider this study novel.

A Java project with four versions was chosen as the dataset for the study. Five graduate students from the Department of Computer Science at the University of Victoria participated in the study. Each participant had experience on a team software project, working with at least one version control tool.

Following the pre-study questionnaire (to determine their previous programming and version control experience etc.), a fifteen-minute orientation on Xia was provided to introduce the basic tool operations and the tool's core features to each participant. Then, a task list was administered to participants. Further inquiry into the user's opinion of the tool was gathered through a post-study questionnaire.

The following subsections describe in detail the tasks users performed and our general observations.

3.1. Tasks

Two sets of similar tasks were assigned to participants corresponding to two different data resources: the data in the CVS repository and the data in the programmer's own workspace. These two sets of data constitute a programmer's working data in the real world.

The tasks involved exploring the information space and answering questions related to team work and software history, including the 5W+2H questions. For example, one of the tasks asked the participant to name all programmers that have been working on the project. Another task asked the participant to find out who was the last person working on a particular file. These two tasks correspond to the "Who" question on the project and file levels. In regards to the "What" question, we asked the user to determine what kind of changes to a particular file have been made. As per the "When" question, we encouraged the user to establish which file was changed most recently. With respect to the "How" question, participants were asked to discover how a particular file was changed in the latest commitment. The "Why" question was explored by asking for the rationale behind a particular change, and the "History" of a file was explored by request too.

3.2. General observations

Participants successfully resolved most tasks. Some general observations were as follows:

- The visualization and exploration techniques provided by the **Attribute Panel** were used frequently to resolve the tasks. Also, participants pointed out in their post-study questionnaires that they would like to use features of the **Attribute Panel** in their everyday work.
- The tool appeared to be easy to learn and use. Although only fifteen minutes of orientation was provided, participants used the tool effectively to perform the tasks. They were aware of the possible ways to use the tool to solve problems and did not require additional assistance or note any significant difficulties.
- Participants considered the tool informative, from both the project manager and programmer perspective. Candidates indicated they believe the Xia tool could prove helpful in helping them solve many problems they encounter in a work environment.
- The tool was also used to answer more sophisticated questions by making use of a combination of features. For example, one of the tasks asked the participants to find out which file is most stable and which file is most active. The participants defined "stable" and "active" in a similar way: a file that has not been changed for a long time and to which very few changes were made was considered stable; the opposite held true for an active file. To answer this question, participants chose both the *last commit date* double slider and *number of change* double slider to narrow down the range of candidate nodes, and analyzed the candidate nodes.
- The visualization features in Xia helped the users gain more awareness of their teammates activities.
- The participants were impressed by the immediate feedback the visualizations provided when they posed a new query. Some of them had special interests in color schemes while others used filters more often.

Though the positive feedback is encouraging, we also noticed some deficiencies of the tool:

- Some participants were confused when working with different revisions of the same file. They suggested that some kind of mapping between different revisions of the same file would be helpful.
- The file revision organization requires a more elegant display. Currently, the file revisions are organized by software versions. However, revisions not belonging to a particular version will not be considered or displayed in the tool. This may lead to the loss of information.
- Some participants also suggested a time-line arrangement of project versions as time is a very important attribute in version control.
- Visualization of other attributes was also anticipated by some of the users. For example, one of the users was interested in who originally created a particular file.
- Participants considered the "diff" function – a comparison of two different file revisions very important in their everyday work. We considered displaying the CVS plug-in's diff view within Xia, however, Xia does not currently support this feature on account of difficulties embedding Xia's Java Swing [15] GUI inside of Eclipse's SWT [16] GUI (a problem discussed by Rayside *et al.* [10]). Further investigation is required for this technical issue.

3.3. Justification of the study design

The study we conducted is a very preliminary step to provide feedback on the use of a tool such as Xia and to help us in our requirements gathering process. Although the number of users was small, we were more interested in finding out if the requirements we had were correct, and if we were missing any. The preliminary study also allowed us to study how the tool can be used to help with version control activities. Our intention at this point is not to perform statistical analysis of results, as we believe research in this area is still too new. The size of the code studied was also small but the study required that the users gain some knowledge of the code in a relatively short time. We also did not compare our tool to others as firstly, there are no other prototypes available that use visualization for version control activities. Secondly, we considered comparing our tool to CVS, but this would be a biased study as our tool provides answers to questions which cannot be easily answered by CVS even in a textual way.

4. Future Work

Based on our observations from the user study we found more research questions that need to be explored. For example, we believe more version attributes beyond the 5W+2H questions (e.g. the creator of an artifact) and visualization of finer granularity of version control should be investigated. We are currently making improvements to our tool – the question we now face, is how should we proceed in the next evaluation phase? Our proposed approach is to do an introspective case study by applying the tool in our own research group's programming activities. We also believe the requirements are evolving with the availability of various tools. We are interested in feedback on our tool and on our requirements at the Vissoft workshop.

References

[1] Ball, T. A. and Eick, S. G. 1996. Software visualization in the large. *IEEE Computer*, vol. 29, no. 4, pp. 33-43.

[2] Card, S. K., Mackinlay, J. D., and Shneiderman, B. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann.

[3] CVS 2003. The CVS website: http://www.cvshome.org/

[4] Dourish, P. 2002. "Visualizing Software Development Activity". URL: http://www.ics.uci.edu/~jpd/research/seesoft.html

[5] Eclipse 2003. Eclipse Homepage: http://www.Eclipse.org

[6] Eclipse Platform, 2003. The Eclipse Platform Subproject Webpage: http://www.eclipse.org/platform/index.html

[7] Eick, S. G., Steffen, J. L., and Summer, E. E. 1992. Seesoft – A tool for visualizing line oriented software statistics. *IEEE Trans. Software Engineering*, vol. 18, no. 11, pp. 957-968.

[8] Lintern, R., Michaud, J., Storey, M.-A., and Wu, X. 2003. Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse. In *Proceedings of Software Visualization 2003*.

[9] McGuire, K. 2002. VCM 2.0 Story (article in Eclipse website: http://www.eclipse.org/platform/index.htm)

[10] Rayside, D., Litoiu, M., Storey, M.-A., Best, C. and Lintern, R. 2002. Visualizing Flow Diagrams in Websphere Studio Using SHriMP Views (Visualizing Flow Diagrams). *Information Systems Frontiers: A Journal of Research and Innovation,* vol. 4 (4)

[11] Shneiderman, B. 1992. Tree Visualization with Tree-maps: A 2-d space-filling approach. *ACM Trans. Graphics*, vol. 11, no. 1, pp. 92-99.

[12] Shneiderman, B. and Wattenberg, M. 2001. Ordered Treemap Layouts. In *Proc. IEEE Symposium on Information Visualization 2001*, 73-78. Los Alamitos, CA

[13] Storey, M.-A., Best, C., Michaud, J., Rayside, D., Litoiu, M. and Musen, M. 2002. SHriMP views: an interactive environment for information visualization and navigation. In *Proceedings of CHI 2002 Conference,* Minneapolis, Minnesota, USA, pp. 520-521.

[14] SHriMP 2003. SHriMP Website: www.shrimpviews.com

[15] Swing 2003. The Swing Connection, http://java.sun.com /products/jfc/tsc/

[16] SWT 2003. SWT: The Standard Widget Toolkit, http://www.Eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html

[17] Tu, Qiang and Godfrey, Michael 2002. An Integrated Approach for Studying Software Architectural Evolution. In *Proc. of 2002 Intl. Workshop on Program Comprehension (IWPC-02)*.

[18] Ware, C. 2000. *Information Visualization, perception for design.* Morgan Kaufmann

[19] Weinberg, Z. 2002. Novel Methods of Displaying Source History: A Preliminary User Study. http://www.panix.com/~zackw/cs260/novel-methods-of-displaying-source-history.pdf

Visualization for Software Risk Assessments

Jordi Vidal Rodríguez
jordi@software-improvers.com

Tobias Kuipers
tobias.kuipers@software-improvers.com

Software Improvement Group
www.software-improvers.com

1. Introduction

The Software Improvement Group performs so-called Software Risk Assessments (SRAs) [10]. An SRA is performed to identify the risks inherent in a software system. The types of risks that are identified during an SRA can be varied, depending on the system and the requirements of the customer.

Risks can be identified with respect to maintainability, performance, operational costs, and so on. The systems the SRAs are performed onvary widely in size, technology and complexity. They can be web applications consisting of 20 forms, or multimillion lines-of-code Cobol legacy systems.

An SRA consists roughly of two parts: A first part where the stakeholders in the system are interviewed and documentation about the system is analysed. In the second part of the assessment the source code of the system is analysed using various tools that we have developed at the Software Improvement Group.

The Software Analysis Toolkit (SAT) that has been developed at the Software Improvement Group routinely calculates a number of metrics for the system under assessment. Furthermore it calculates a graph structure that contains all the dependencies in the system. The various dependencies in the system are typed and can be data dependencies from a specific module to a specific view on a database, which in turn is dependent on a specific table in a database, which in turn triggers a stored procedure in the database (and so on). Effectively using the information in this graph depends largely on the ability to interactively view (parts of) the graph, and relating those parts to specific locations in the source code, and to metrics about those parts of the source code.

An example graph of all dependencies in a 150,000 line web application is given in figure 1.

1.1. Scenarios

Visualization needs during an assessment are twofold: first of all, visualization should facilitate the understanding of the system. This initially requires visualizing the overall structure of the system: control dependencies between modules, and data dependencies between modules and databases. Afterwards, when a general understanding of the system is reached, the visualization should provide detailed views to validate ideas that have occurred about the functioning of the system.

Interactively manipulating the view on the software system is key in both phases. As an example, consider the following scenarios.

Scenario 1

A system consisting of 3,000,000 lines of Cobol is analysed using the SAT. This results in a graph containing all the dependencies. (More about the data representation that results from the SAT analysis in section 3). The first visualization shows that there are 20 database tables in the system, and that only about 10 modules (out of 1200) access these tables. Closer inspection of the 10 modules shows that these are so-called utility modules, and that all modules that call these utilities can be considered to perform database access. The view should than be adjusted to remove the database utilities, and replace them with direct edges from the modules that called the utilities to the database tables.

Since these systems typically use different technologies to access persistent data, the view needs to be adjusted to accommodate for that fact: for gaining a general understanding of the system database access through DB2 should be visualized in the same way as, say, access through IMS. However, when looking at detailed IMS usage DB2 should obviously be removed from the view.

Scenario 2

As an example of a more detailed view consider the system displayed in figure 1. This is a web application built using Microsoft Active Server Pages. After a first inspection, and interviews with the developers of the system the consultant performing the assessment comes up with the following

hypothesis: All asp files in the system include a standard "library" file, and a single file that contains the non-dynamic portion of the page. In order to validate this hypothesis she looks at the graph in figure 1. Obviously there is nothing to see there, since there are far too many edges and nodes.

The graph needs to be interactively filtered to first show only the asp files, and their includes. If there are too many to immediately see whether the above hypothesis holds, than a threshold needs to be set to show only asp files with less than two includes: if they exist then the hypothesis does not hold.

1.2. Graph Visualization Requirements

The assessments described in the scenarios above could be performed using an interactive graph browser that supports a number of operations. The operations we currently are looking for in a graph visualization tool are: abstractions (leading to nested graphs), searching, filtering, undo facility, and automatic layout. For instance, the set of operations should enable us to replace a node by direct edges between all its sibling nodes. These operations will be applied on large graphs (up to 100,000 nodes). Finally, an scripting facility to enable automatization and an annotation facility to store comments on findings are seen as crucial for efficiently carrying out SRAs.

1.3. Position

Our position is that in spite of good tools solving partly SRA's requirements, there is still missing a general tool to handle all SRA aspects smoothly, embracing from data model, visualization to reporting.

In the next section, we give an overview of the various software visualization tools that we are currently using, or have tried in the past.

In section 3 we describe the data model that we use for our assessments, and how it relates (both in theory and in practice) to our visualization tooling. Section 4 discusses the various challenges we see when using existing visualization tooling.

We end the paper asking ourselves whether we should start to produce our own visualization software...

2. Related Tools

The most common techniques for software system comprehension are graph visualization and a combination of browsing/navigation/query. Specialized instances of these techniques are scattered among tools. The most relevant and inspiring tools for our activities are described next.

Rigi [7] uses a nested graph model. However the graph's levels are displayed using multiple windows. It has basic graph operations for name pattern search, selection and abstraction. Visualization and interaction can often be cumbersome.

SHriMP [9] is a nested graph navigation tool, lacking any graph manipulation feature. It complements Rigi. It maintains context and focus via a modified fish-eye algorithm.

Dalí [6] is a workbench that supports extraction and fusion of architectural views. It highlights the need to fuse views from different source extractions, leaving the data gathering to other tools for the purporse. It is an open approach to integrate tools and uses a common data repository.

Portable Bookshelf [5] is aimed at re-engineering and migration, mainly as a navigation tool by means of directed graphs. Software landscapes visualize the main part of the system and keep context with neighbouring subsystems.

Code Crawler [3] is a tool that combines object oriented software metrics into some predefined such graph models like tree, matrix correlations and histograms. The nodes (or entities) can distinguish up to 3 data dimensions, visualized as x, y size and colour.

CIAO [2] is a flexible navigator that can visualize graph models and query the system at source code level. It can be used in any project by specifying a new data model.

SPOOL [8] is a tool set to bridge to other comprehension tools. It allows browsing of high level constructs, query the design and structural searching, but lacks abstraction operations. Other helper tools are used for source code analysis. It aims to integrate several tools to allow flexibility in creating user-defined views of any system.

As the above descriptions reveal, these tools do not satisfy all requirements as stated in section 1.2. Consequently we looked at graph layout libraries such as Dot, aiSee, GVF, JViews and Tom Sawyer, which offer advanced features. Dot is not an interactive tool, although there are some libraries containing dot that alleviate this problem. aiSee was found to offer good layout algorithms but with an unfriendly user interface. The last two are powerful graph libraries allowing nested, multiple layouts.

3. Model Requirements

The tools above demonstrate the usefulness of the various features they were developed for, but fail to satisfy the complete list of features we need to perform Software Risk Assessments.

During assessments we need to contrast different views at different abstraction level of the system, subsystem or a slice of source code. Due to the variety of systems that can be analysed, no specialized tool fully suits the purposes. Instead an open, extensible tool should be created to cope with the business demands.

In general data is gathered during the assessment process into a generic data model (in similar fashion as FAMIX [4]). From it we generate three generic types of views: Directed graphs, charts and source code. The visualization tool should be able to display these three views, their relations, and navigate effortlessly between them.

The model we generate views from are described below.

3.1. Data Model

The are two main aspects the data model must meet. Fisrt, it have to allow navigation from one view to another, this is, regardless of the inspection starting point we can navigate to any related view forth and back even at different abstraction level.

Current tools attempt to provide such functionality (i.e. SHriMP, SPOOL) but limited to a pair of views with their own data model. On the other side, (commercial) source navigators (such as Eclipse [1]) provide excellent features but miss high level views.

Second, it holds data from three subareas: the artifacts' relations, visualized as a directed graph; the software metrics, visualized in charts; and the source code, visualized as enhanced text.

The model must be able to support any programming language, as in the case of large legacy systems. Thus object–oriented, procedural, functional and scripting languages must all be supported transparently. Only the artifact gathering tool is tailored to the target language.

Next, we describe what features both the data model and each view should provide support.

3.2. Graphs

Most of our interest is on exploring large and highly connected hierarchical typed directed graphs and derive some knowledge. We have seen there is no single tool that supports all required features for effective exploration: nested graphs (SHriMP), abstraction operations (Rigi), incremental layout, context keeping (SHriMP, PBS) and annotations. Instead each tool supports one or few of these.

3.3. Source Code

Source code inspection is a common practise in software assessment. Either beginning from source code or from a model, we are interested in obtaining alternative views of the same set of artifacts.

When inspecting a piece of source code, the corresponding subgraph and a set of related metrics should be displayed. Similarly, when pointing to either a graph or a chart entity, the related piece of source code would show marked up.

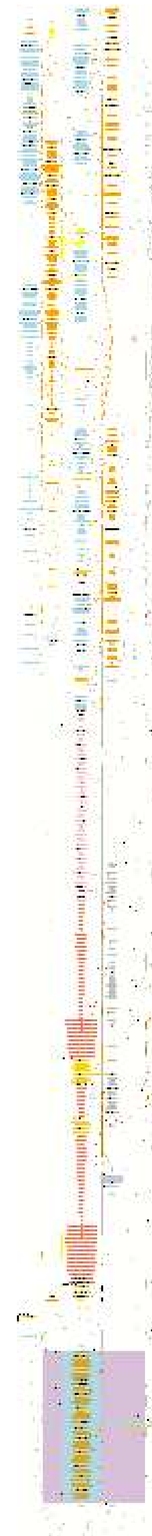


Figure 1. A graph containing all the dependencies of a system under assessment

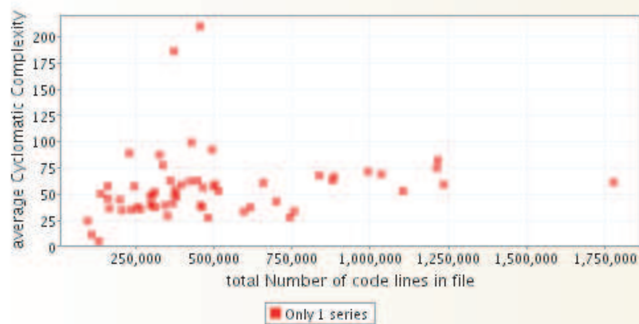


Figure 2. Complexity of systems versus their size

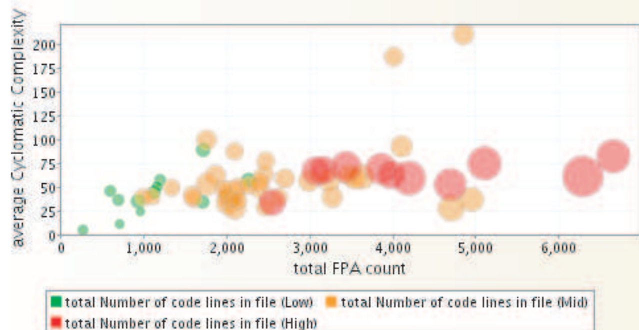


Figure 3. Complexity of systems versus function points and lines of code

3.4. Metrics

Software metrics gathering, at different granularity levels, is routine in our assessments. Currently we visualize them as independent charts (i.e. bar charts, pie charts, bubble charts).

Metrics are valuable to quickly point out measurable features such as complexity and size among other source code attributes.

Linking the metrics with the other views also aids in inspecting. For instance, selecting a complex system's program and then browsing its source code. We also realize it is helpful to merge metric information into the graph model as in [3].

4. Visualization Challenges

To carry out SRA we actually use practises from reverse engineering, software metrics measurements and automatic generation of documentation. Currently we are using a set of independent tools for the purpose. These tools are integrated by our Software Analysis Toolkit. The visualization part currently consists of graphs visualized using dot, charts displayed using JFreeChart, and marked up source code using a purpose built system. We browse the source code and make annotations manually.

We have conducted experiments with some tools mentioned in related work section. Other tools were dismissed after checking the list of features or after seeing the demos. The tools we have experimented with did not serve our goals. Most tools perform a single task well, but not others. Furthermore, we had problems integrating these tools within the Software Analysis Toolkit, and the usability for some of these tools is so terrible that we wonder whether they are used at all.

Our wished features for graph visualization tools are as follows:

4.1. Focus and Context

Usually we focus on a small part of the large system. When zooming in, the involved nodes should be placed close to each other to fit one screen while maintaining its context (i.e. its immediate neighbours). The context may be essential to identify possible erroneous relations. Solutions like nested graphs, multiple views, fish-eye views and showing neighbours seem feasible.

4.2. Annotations

The assessment process produces large amounts of results referencing both the detailed and the coarse level. Crucial is the reproducibility of the assessments, for updated versions of the system, and retrieval of old assessment results for comparison. This opens the way to trend analysis.

Even rudimentarily supported, by saving views and other data files, an integrated annotation tool would increase productivity. The Film strip feature in SHriMP, or the Saving View in Rigi are both promising methods. Structured storage would help in the report writing phase.

4.3. Layout

The graph layout reveals important relations that can be spotted by a quick visual inspection. However, obtaining a good layout is not trivial, not to mention that for large graph no layout has given satisfactory results.

Therefore, using the focus+context to reduce the graph size to display, layout algorithms can be used again. Nevertheless, it is also of importance to keep the mental map as the exploration proceeds. For instance, smooth transistions and minimal alterations to the graph structure should be enough. Animation cues are not discarded at all.

A not less important aspect is labeling. Labels should be readable at any zoom factor. Although if the node is too small, there is no need to show its label. SHriMP [9] approach seems the most advanced approach so far.

4.4. Graph Operations

During the understanding process we manipulate the graph by, for instance, grouping (abstraction) common artifacts into subnodes (hence nested graphs). Other operations are: navigation, search, filtering and selection. We are currently not convinced that this list is exhaustive, but more experiments are needed.

Navigation should allow to track the visited elements. Different approaches could be: a) list visted nodes in a separate view; b) move visited nodes close each other; c) not alter the layout. Options a), b) maintain context, while c) is adequate when only the final target is necessary.

The search space can be textual or structural. Locating certain names of artifacts is textual. Locating chains of node types and edge types is structural. Questions like "Is database X accessed directly or indirectly by any program in Y?" should be answered by a structural search. This search involves textual and structural search combined.

Abstraction as supported by Rigi merged with the SHriMP capabilities of nested graphs would be a nice start.

We have pointed out the importance of keeping focus and context while carrying operations that affect the graph structure. When having to understand small portions of large graphs, these operation features help raise model comprehension by reducing confusion by sudden changes of the layout.

5. Conclusions

A number of tools exist that partly solve the complex reverse engineering task of Software Risk Assessment.

Projects like SPOOL try to go further, providing environments were multiple tools collaborate to tackle as much of the reverse engineering tasks as possible in an elegant, clear way.

A standarized, integrated environment would allow users to access simultaneously a broad set of tools with the advantage that they could tightly collaborate each other, resulting in a productivity boost. The diverse techniques could be tried under a single environment allowing to more efficiently compare, investigate, create new research techniques benefiting from the already developed helper tools. For instance, to try a new visualization technique, only the view has to be programmed.

We have been developing a generic data repository to hold all data we derive from large software systems to perform assessments. We currently can visualize aspects of this data in various ways.

Our search for an interactive graph viewer that suits our needs has so far been interesting, but has not led to the tool we want. We have listed our wishes regarding such a tool.

Nevertheless, we are currently contemplating building the tool ourselves. We would like to invite the software visualization community to challenge our ideas, to tell us such a tool already exists, to tell us everything we know is wrong, and finally, to collaborate with us to build a system that would perfectly suit our needs.

References

[1] *The Eclipse project*. http://www.eclipse.org.

[2] Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proc. Int. Conf. Software Maintenance, ICSM*, pages 66–75. IEEE Computer Society, 1995.

[3] S. Demeyer, S. Ducasse, and M. Lanza. A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization. In *Proceedings of the Working Conference on Reverse Engineering*, pages 175 – 186, 1999.

[4] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, 1999.

[5] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, , and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, volume 36, pages 564–593, 1997.

[6] R. Kazman and S. J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[7] H. Müller. *Rigi*. http://rigi.cs.uvic.ca.

[8] S. Robitaille, R. Schauer, and R. K. Keller. Bridging Program Comprehension Tools by Design Navigation. In *Proceedings of the International Conference on Software Maintenance*, pages 22–31, October 2000.

[9] M.-A. Storey. *SHriMP*. http://shrimp.cs.uvic.ca.

[10] A. van Deursen and T. Kuipers. Source-Based Software Risk Assessment. In *Proceedings of the International Conference on Software Maintenance*, 2003.

Position paper:
MetaViz – Issues in Software Visualizing Beyond 3D

Juergen Rilling, Jianqun Wang, S. P. Mudur
Department of Computer Science, Concordia University
{rilling, jianq_wa, mudur}@cs.concordia.ca

Abstract

In this research, we present our MetaViz project that was initiated to investigate software visualization issues in the context of novel modeling and visualization techniques. The focus of our research is the visualization of software structures and their dynamic behavior using 3D visualization techniques. Rather than applying traditional 2D visualization techniques and transfer these to the 3D space, we are exploring novel visualization techniques and investigate computational, grouping and layout related issues of these visualization techniques.

1. Introduction

For large, complex software systems, the comprehension of such diagrammatic depictions is restricted by the resolution limits of the visual medium (2D computer screen) and the limits of user's cognitive and perceptual capacities. One approach to overcome or reduce the limitations of the visual medium is to make use of a third dimension by mapping source code structures and program executions to a 3D space [6]. Mapping these program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code. Scalability becomes a well-known barrier that exists in both 2D and 3D software visualization [9]. Improving layout algorithm and clustering management is one way to make visualization techniques scalable. In this paper, we will investigate some of the readability issues of visuals representations in 3D space, and present improvements in three different aspects. First, we will introduce the use of metaballs as a metaphor to visualize software systems to improve the more traditional representation of "Nodes and Arcs". Second, we will use hierarchic grouping of entities to abstract higher level entities and improve the usability. This will lead to an "overview first, zoom and filter, then details on demand" approach. Finally, we will address issues related to gird based 3D layout algorithms as some of the techniques to improve readability of the 3D visuals created [12].

The remainder of the paper is organized as follows. Section 2 introduces the MetaViz project; section 3 discusses the metaball visualization approach and the Marching Cube algorithm used to create the metaballs. Section 4 introduces 3D layout algorithms and quality aspects. In section 5 we discuss grouping and clustering to improve the comprehensibility of visual representations. Section 6 illustrates applications of MetaViz tool, followed by a discussion of future challenges in section 7.

2. Overview of MetaViz

The MetaViz tool was developed using Java 3D as an independent, but reusable 3D visualization environment to investigate the various application domains for the metaball metaphor. The tool provides a programming interface that allows for an easy extension and further reuse of the tool. The MetaViz tool consists of three major parts: the grid-layout, a clustering and grouping algorithm, and the metaball rendering engine (see Figure 1).

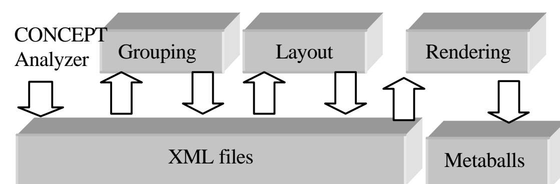


Figure 1. MetaViz architecture

The grid layout plug-in is based on an XML input file, which describes the software artifacts and their internal relationships. Within the MetaViz tool, users can select among 4 different layout algorithms, and provides users with feedback about the progress made in the layout optimization by displaying snapshots of the current layout state. Once a predefined optimum is reached, the layout optimization is completed and will be stored in an XML file for further processing.

The rendering engine of the MetaViz tool reads this XML file as input to generate and render the metaballs in

3D space, by mapping the structural software properties to the properties of the metaballs. After the completion of the rendering process, the user can navigate through the visuals and apply overview, select and zoom techniques to refine the current view.

3. Metaball Metaphor

Creating intuitive and useful abstraction is one of the major research issues in 3D software visualization [5]. As part of the MetaViz project we are investigating new metaphors for 3D software visualization that can provide additional guidance in supporting the comprehension process [6].

3.1. Metaball vs. "Nodes and Arcs"

Metaballs provide a three dimensional picture with smooth connection between metaballs and shading, which eases the difficulties of building mental model. Figure 2 illustrates the advantage of metaball over a 3D sphere-line graph. Both visuals display the same information and use the same layout. One of problems of the sphere-line graph is that it cannot convey some structural information in the same way as, for example, the metaballs. The fusion (the thickness of the connection) among two or several metaballs can be used to show clearly structural dependencies. The fusion can also be used to indicate the relationship among different software artifacts. Shading and blending are other options that can be applied to convey additional information not available in most traditional software visualization techniques.

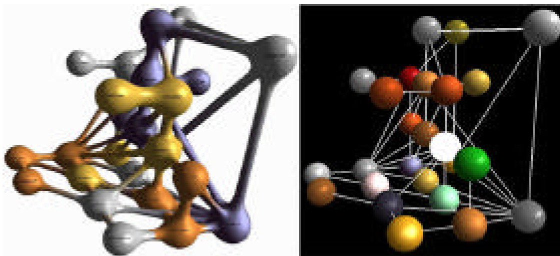


Figure 2. Metaball vs. sphere-line graph

However, it should be mentioned that the metaball has one major disadvantage - its computational complexity, caused by the Marching-cube algorithm that is used for rendering the metaballs. For visualization techniques to be useful and applicable, performance plays an important role. In what follows we discuss an optimization for the general Marching-cube algorithm that improves on the rendering speed of the algorithm.

3.2. Marching-cube Algorithm Optimization

The performance problem with the metaball rendering is directly related to the computations carried out by the Marching cube algorithm that computes the iso-surface of the metaballs [15]. One way to improve the performance of the algorithm is to avoid any surface computations for grid cells that do not contain any parts of the metaball iso-surface. The following table (Table 1) shows our test results for 64 metaballs with and without this optimization. For the experiment, we limited the recursion level to 9. It should be noted that the recursion level directly corresponds to the rendering quality of the metaballs. For a recursion level of 5 or lower, the resulting metaball surface becomes too coarse (polygonized) to be useful. For more than 9 recursions, the computation complexity becomes too large to be applicable.

Recursion times	Non-optimized	Optimized
5	1,093 ms	157 ms
7	55,469 ms	1,750ms
8	459,609 ms	9,031 ms
9	3,286,978 ms	58,343 ms

Table 1. Test results for 64 metaballs rendered by the marching cube algorithms optimization. (CPU: P4 2.8GHz, 1GB RAM)

4. 3D Grid Layout Algorithms

Within the MetaViz tool, we applied a grid layout approach, where the position of each node corresponds to integer coordinates. There are two major reasons for applying this approach for 3D visualizations. Firstly, the layout algorithm allows for a space efficient visualization of metaballs. For example, 1000 entities can be placed within a 10*10*10 grid using the metaball visualization. On the other hand, displaying the same 1000 entities using a cone tree [2], the size of each entity becomes too small at the lower levels of the tree.

Secondly the grid layout is reusability and can be applied for other visualization techniques that have similar layout and readability problems. The presented layout algorithm is developed as a separate Java package plug-in that can be reused by other visualization approaches within our CONCEPT project (e.g. UML diagrams, 3D worlds, etc.).

It should be noted that grid layout algorithms have two major shortcomings. One is their lack of real-time responsiveness, which makes them not suitable for interactive/animated applications. The other shortcoming is that the rearrangement of graphs may not preserve some designing properties, and the context (e.g. grouping, clustering) of the original design might be lost. [8].

4.1. Readability criteria

Software visualization techniques were originally introduced to support people during software and system comprehension. For a visualization technique to be useful, its readability becomes a major quality factor. Readability criteria have already been established and applied in information visualization to evaluate the quality of layout algorithms [10]. Figure 3 shows the importance ranking of different criteria for the readability of a visualization technique in general. In our implementation, we take into account most of these criteria. The objective function is a weighted sum of these numbers, with line object crossing having the highest weight, and the length of arcs having the lowest weight. Drawing space and density distribution can be further optimized by choosing a minimum grid size that can accommodate the given number of metaballs.

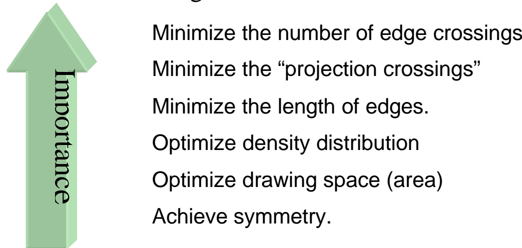


Figure 3. Visualization criteria

4.2. 3D Grid Layout as a State Searching Problem

The 3D grid layout problem can be described as placing m entities into n^3 positions to satisfy certain readability constrains, where $m <= n^3$.

The computationally intensive nature of 3D layouts has already been shown in existing layout algorithms in other application domains [3]. The complexity of evaluating a 3D layout is O (n3!). A brute force search is not feasible, because of the size of the search space [13]. One solution to the search problem is to apply a heuristic search. In what follows we apply the hill-climbing algorithm to improve the grid layout algorithm.

4.3. The Hill-climbing Search as a 3D Layout Problem

Initially, a search tree has to be created that starts from a root state and moves on to its children. In the case of a root state, the algorithm randomly places m entities within the given n^3 space. The algorithm swaps some of the entity positions and creates new states for the children. The number of children corresponds to the "branch factor", which directly influences the complexity of searching algorithm. We calculate the branch factor for certain operators (Table 2).

Operator	n entities into n positions	Branch factor
Switching two entities	(n-1)*n/2	351
Switching three entities	(n-1)*…*(n-4)	15,600
Switching four entities	(n-1)*…*(n-5)	3.6e+5
Switching five entities	(n-1)*…*(n-6)	7.9e+6

Table 2. The branch factors of searching trees for placing 27 entities into 27 positions

The hill-climbing algorithm is a state searching algorithm, with the goal to minimize the memory requirements to perform the search. A detailed analysis of existing searching algorithms that are studied extensively in the field of Artificial Intelligence can be found in [13]. The limitation of the hill climbing algorithm is that it can only find a local peak. In our research, we try to overcome this limitation by modifying the hill-climbing algorithm. In our extended version called as the competition hill-climber, we modify the hill-climbing algorithm by using a more expensive comparison to break away from the local peak and jump to another hill which has a higher peak than the current hill.

Thread	Projection crossings	Line object cross	Lines length	Objective Function value	Compute time (sec)
0	22	0	125	279	1284
1	10	1	130	221	1134
2	24	1	149	338	953
3	33	0	126	357	1161
4	20	0	126	266	1230
5	17	1	123	263	1353
6	18	0	124	250	1348
7	12	0	114	198	1325
8	16	0	129	241	968
9	11	1	115	213	1329

Table 3. Results of competition hill-climber (Test condition: P4 2.8GHz, 1 GB RAM)

Moreover, we apply a random positioning of the entities into the grid and use these as random starting states. These random starting states will then lead to different peaks. In our implementation, we use ten threads to evaluate ten different starting states using one of the above strategies. Finally, we compare the ten computed peaks and choose the peak with the best result. In our example (see Table 3), thread number 7 has the best estimation value.

5. Grouping

"Program analysis is a crucial part of many program understanding tools" [1]. Grouping can be described as a process of program analysis prior to the layout management. The layout algorithms are constrained by the amount of information to be displayed and the limited

screen space. Even, if one manages to create a layout, the resulting visual might have far too much information, causing an information overload. Therefore, limiting the number of entities to be displayed to the user is one of the key challenges in software visualization. For the visualization of large software systems, it is essential to provide some type of grouping to create a decomposition of the system. It has been shown that grouping can improve readability [11], by supporting a representation that is closely related to the mental model a programmer forms of a system [14] during typical comprehension tasks. Grouping or clustering can be applied to generate suitable abstraction levels and therefore allow for a reduction of the amount of information to be displayed on the screen. In this paper, we present two methods of grouping: metric-based grouping and feature-based grouping.

Metric-based grouping.

For metric-based grouping an internal relation table is created that analyzed the coupling among different classes. The number of function calls defines the weight of relationships among the different entities (coupling). In a first step, we identify entities that are strongly coupled with each other and group them closely together. During the second processing step, we identify a threshold that corresponds to the maximum number of entities displayed on the screen. Based on the coupling relationships and the maximum threshold, objects are placed on the screen. For example, the threshold for Figure 4 is four. After the first grouping iteration, ten groups are identified. Each of these groups can be treated as a separate entity, for which the process can be re-applied recursively to create the next higher level of abstraction.

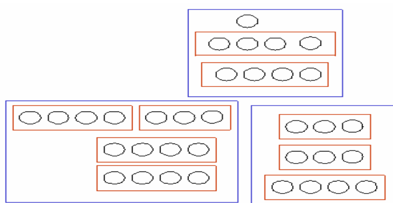


Figure 4. Metric-based grouping

Feature-based grouping

There exist several techniques for identifying features in software systems, e.g. (program slicing [12], concept analysis [7], etc.). For some applications, such as testing and debugging, programmers might be interested in focusing on particular features instead of the whole software system. Therefore, grouping software entities based on their features can help programmers to focus on related software parts and therefore reduces the cognitive and comprehension load.

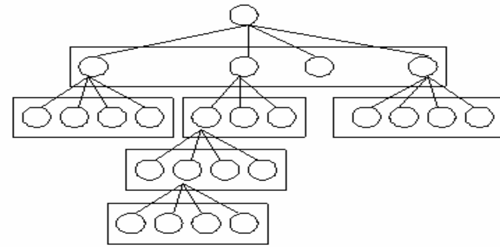


Figure 5. Feature-based grouping

Figure 5 shows an example of such a feature-based grouping. A program might be represented, like in this example as a hierarchical structure, with a feature consisting of several sub features.

6. Applying MetaViz

In this section, we illustrate some applications of MetaViz and its visualization techniques, layout algorithms and grouping techniques. For illustration purposes, we use the MetaViz implementation itself. MetaViz consists of 64 classes with a total of approximate 10,000 LOC. The examples will illustrate how metaballs in combination with different source code analysis techniques guide programmers during program comprehension. The examples include: a hierarchical representation, grouping based on coupling among different software entities, the animation of the layout algorithms and visualization of dynamic program aspects using metaballs.

6.1. Hierarchical Structure of Software

Typically, any larger software system is organized as a hierarchical structure and many software visualization techniques are developed for visualizing these hierarchies (e.g., tree structures such as cone-tree, cam-tree, and information-cube)[4]. Within MetaViz, we visualize these hierarchies by applying the metaball metaphor.

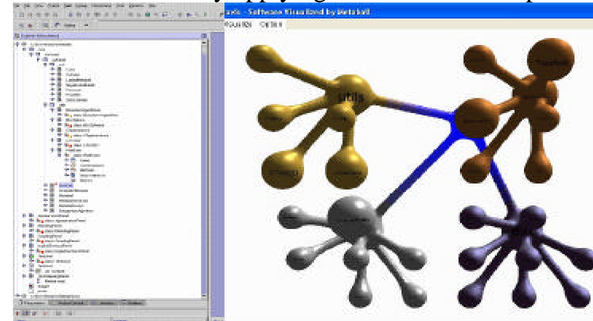


Figure 6. Hierarchical structure in SunONE (left); same structure using MetaViz's visualization

Figure 6 shows the MetaViz program structure in both a textual representation as it can be found in most IDEs (left) and a graphical representation using the metaball

approach (right). We additionally apply color coding to indicate package dependencies and hierarchy information.

6.2. Applying the Metaball Metaphor to Visualize Coupling Measurements

Visualizing the internal relationships and call dependencies of software systems is an essential part of many software visualization tools. In this application example, we apply coupling between object classes (CBO) as the relationship measure among software artifacts. The cylinders diameter connecting two metaballs corresponds directly to the existing CBO coupling among two entities. Furthermore, the cylinder provides a visual feedback of the strength of the coupling.

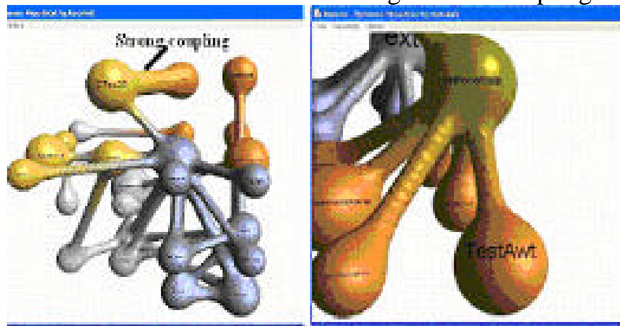


Figure 7. Metaball visuals of MPC measurements

6.3. Layout Management Animation

The MetaViz tool not only implements an optimized hill-climbing layout algorithm, but also has the option to visualize the layout computation and optimization process. The following two sample snapshots (Figure 8) are from one of these layout optimization sessions, providing users with instant visual feedback on the progress of the current layout optimization. It also allows the user to terminate the optimization process once layout meets the visual expectations (quality).

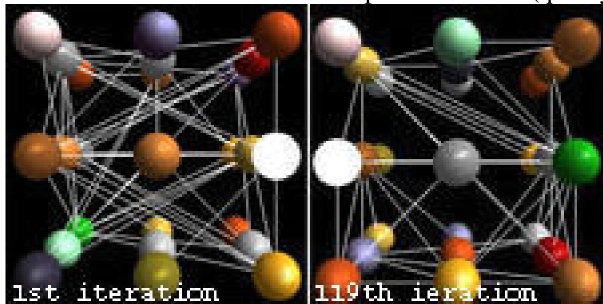


Figure 8. The Layout optimization Process

6.4. Visualizing Dynamic Information

Currently most visualization tools are restricted in their ability to support dynamic program information and aspects. Within the MetaViz tool, an execution trace is recorded for a specific program execution; allowing for a stepwise re-execution that can be displayed as a sequence of frames. Figure 9 shows four frames of such an animated re-execution. The metaballs in the picture represent executed classes, with the diameter of the metaball corresponding to the number of executed statements and the white ball indicating the current execution position.

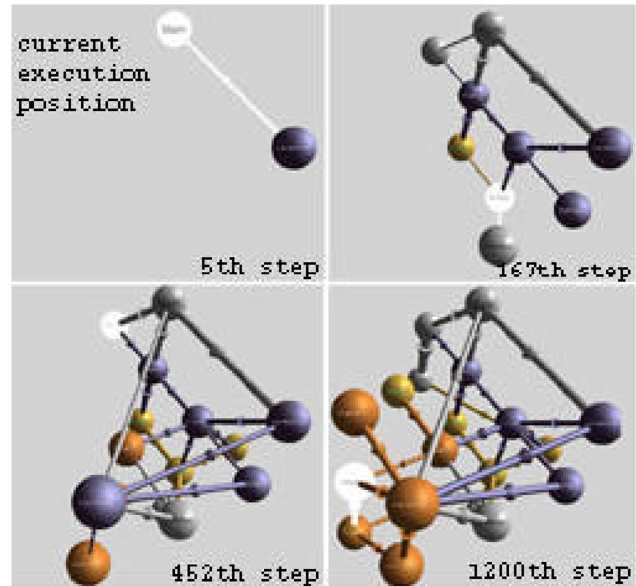


Figure 9. Dynamic visualization

7. Software visualization beyond the third dimension

In this article we presented our MetaViz tool, in which we address the following visualization issues: We introduced the metaball metaphor as a basic notation for our software visualizations, a grid based 3D layout and grouping techniques. One of the initial motivations for the MetaViz project was to explore novel software visualization techniques in 3D space. One of the immediate results of our prototype implementation was that 3D visualization techniques can enhance and benefit the comprehension process by enhanced utilization of screen space and additional visualization effects (shading, transparency, fusion, etc.). Source code analysis can provide additional insights and guidance in filtering and visualizing the information. Our grid layout technique improves the readability of 3D visuals on screen. Specifically, the ability to dynamically observe the behavior of the layout algorithm as it progresses helps the user to be an active participant in this visual generation process. Grouping based on object coupling and slicing based features were demonstrated, to avoid

information overloading. However, several main challenges remain that go beyond just moving to 3D space or applying some layout or clustering techniques.

Remaining key challenges are the re-creation of a mental model that closely corresponds to the mental model designers/programmers developed during the originally forward engineering process. No matter what layout, clustering or grouping algorithm one applies to identification and analysis of logical relationships among different software entities, they always will be limited by the quality of the algorithm and the lack of domain knowledge modeling. Overcoming these limitations requires additional information sources (other than source code) and domain knowledge has to be incorporated in existing layout, grouping and clustering algorithms.

With more and more applications moving into distributed and network centered environments; software visualization, analysis techniques, as well as grouping and layout approaches have to keep up with these changing requirements. Visualizing dynamic and behavioral aspects has additional challenges in the form of filtering large amount of information, the creation of meaningful abstractions and

8. References

[1] A. van Deursen and J. Visser. Building Program Understanding Tools Using Visitor Combinators. In Proceedings 10th International Workshop on Program Comprehension (IWPC'02), pages 137-146, IEEE Computer Society, 2002.

[2] Cockburn, A. & McKenzie, B., "An evaluation of cone trees," In People and Computers XV. Proceedings of the 2000 British Computer Society Conference on Human-Computer Interaction, University of Sunderland, 4--8 September, 2000.

[3] Fabien Jourdan, Guy Melançon A scalable force-directed method for the visualization of large graphs Workshop on info visualization. LIRMM, Monptellier. January 2002

[4] Ivan Herman, Guy Melancon, and M. Scoot Marshall, "Graph Visualization and Navigation in Information Visualization: a Survey", IEEE Transactions on Visualization and Computer Graphics, Vol6 No 3, 2000 Computer Science, 1995.

[5] Knight, C. and Munro, M. "Software Visualization conundrum", Department of Computer Science Technical Report 05/01, July 2001.

[6] Knight, C., and Munro, M. "The Power of (Software) Visualization", Department of Computer Science Technical Report 01/00, January 2000.

[7] Koschke, R., `An Semi-Automatic Method for Component Recovery', Proceedings of the Sixth Working Conference on Reverse Engineering, pp.256-267, Atlanta, October 1999.

[8] M. A. Storey, and H. A. Muller. "Graph layout adjustment strategies". In Graph Drawing '95, pages 487--499, 1995

[9] Maletic, J.I., Marcus, A., Collard, M. "A Task Oriented View of Software Visualization", in Proceedings of the the IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002), Paris, France, June 26, 2002, pp. 32-40

[10] R. Tamassia. New layout techniques for entity-relationship diagrams. In Proc. 4th Int. Conf. on Entity-Relationship Approach, pages 304--311, 1985

[11] S. Mancorids. B. S. Mitchell, Y. Chen, E. R. Gansner "Bunch: A clustering tool for the recovery and maintenance of software structures" In Proc; IEEE Inter. Conference on Software Maintenance, IEEE Computer Society Press, 1999, pp 50-59.

[12] J. Rilling, and S. P..”The Metaball Metaphor for Slicing Based Software Visualization", 2003.

[13] S. Russell and P. Norvig, "Artificial Intelligence A modern approach", Prentice Hall, 1995.

[14] V. Tzerpos, R.C Holt. "ACDC: An algorithm for comprehension-driven clustering", Int. Working Conference on Reverse Engineering, 2001

[15] William E. Lorensen and Harvey E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm", Computer Graphics (Proceedings of SIGGRAPH '87), Vol. 21, No. 4, pp. 163-169

KScope: A Modularized Tool for 3D Visualization of Object-Oriented Programs

Timothy A. Davis	Kenneth Pestka	Alan Kaplan
Department of Computer Science	Department of Computer Science	Panasonic Technologies
Clemson University	Clemson University	Princeton, NJ
tadavis@cs.clemson.edu	kapestk@clemson.edu	kaplana@research.panasonic.com

Abstract

Visualization of software systems is a widely used technique in software engineering. This paper proposes a 3D user-navigable software visualization system, termed KScope, that is comprised of a modular, component-based architecture. The flexibility of this construction allows for a variety of component configurations to validate experimental software visualization techniques. The first iteration of KScope is described and evaluated.

1. Introduction

Software visualization has become an important means by which software engineers can study and understand complex software systems at any stage during the software lifecycle – from initial development to legacy code maintenance. Currently, a popular 2-dimensional (2D) approach to software visualization is represented by the Unified Modeling Language (UML). The application of 3D visualization systems to software engineering is challenging, as in 2D representations, since the software components are abstractions that have no immediately recognizable shape or substance and choices must be made as to the physical object used to represent each software component.

While both 2D and 3D representations can take advantage of shape recognition, symbol set knowledge, and the color awareness abilities of viewers, 3D systems add depth, motion, distance, transparency, animation and spatial orientation as data transmission tools. These additional attributes allow a larger set of data values to be incorporated into a single view. Accordingly, a large amount of information can be communicated more quickly and with a higher assimilation rate [10]. The ability to navigate a 3D visualization space further allows a software engineer to transition from one view to another in a seamless manner.

As a means of testing various possible aspects of a 3D visualization system, a modularized tool, termed KScope, has been designed and a prototype implementation has been developed. The advantage of a component-based system is that it facilitates the creation of experiments in which some components are fixed, while others are altered or replaced in order to compare the efficiencies of various configurations. For instance, parsers for various languages might supply the definition of scene objects while the rest of the components are held fixed in one configuration, and thus allow a realistic evaluation of the degrees of variation in the output based on the language input.

The development of the KScope visualization system is planned as an iterative process. The first iteration defines the following five dimensions as specified in [8] :

- *task* – provide an analysis of java programs
- *audience* – researchers in the implementation of visualization systems
- *target* – Java class files as the data source
- *representation* – primitive visual objects in the analysis
- *medium* – a navigable 3D visualization with a system architecture that allows for the easy substitution of alternate components.

Subsequent iterations will expand the definition of these dimensions.

Several key issues arise when formulating a software visualization system: finding a suitable symbol set for representing abstract program concepts, placing these objects in 3D space to enhance understanding and minimize confusion, ensuring the system scales across systems of varying size, and finally, selecting some form of criteria for evaluating the effectiveness of the visualization.

2. Related Work

Research in the area of 3D visualization over the last few decades has covered many areas. Several studies [4] [10] validate that visualization of software systems increases the ability to acquire knowledge of a software sys-

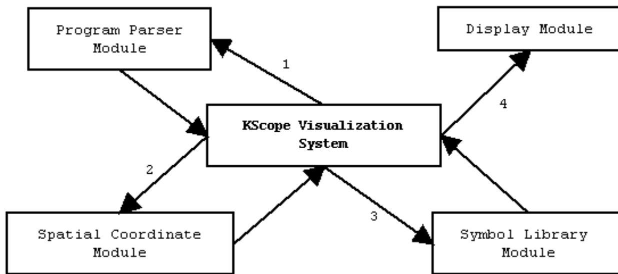


Figure 1: KScope architecture

tem, and that 3D visualizations are more effective than 2D ones in transmitting information to a software engineer. The exploration of visualizing C++ programs as a means of increasing program comprehension is illustrated by [4]. This work explores the basic visual symbol set of objects and relationships in a 2D system and justifies visualization as an important learning tool in understanding software.

A 3D representation increases the software engineer's information perception over that of a 2D representation [10]. Additionally, combining motion (e.g., rotation of the scene) and a 3D stereo view (e.g., through navigation) is a useful visualization technique to aid in understanding the structure of object-oriented code. In many instances, 3D visualization overcomes the limitations of 2D visualizations and in minimizing user confusion and increasing data comprehension [7].

The use of 3D visualization may be especially useful in understanding class structure of Java programs, as evidenced by the J3Browser tool [1]. Here, transparency, depth, ordering in space, and motion are used to convey a large amount of information in a highly effective manner. Of course, object placement and structure of the visual scene are significant in the overall success of the system.

Symbol sets and object metaphors are also significant in creating effective software visualizations. One symbol set, the "Software World" metaphor [5], uses a cityscape based on classes represented as districts within a city, with each district composed of buildings that represent class methods. This type of symbol set can be effective, but in all cases, the symbol set should reduce the complexity of the concept being visualized [6].

More recent customized symbol sets, such as the static shaded 3D symbols in [3], are also effective in user understanding as compared to the standard UML class diagram symbols. The symbols in this set are constructed using a basic symbol alphabet known as *geons* [2] and outperform the UML set in all the illustrated experiments.

In evaluating the effectiveness of 3D visualization systems, two distinct elements should be considered: representation of objects and the mode of visualization [11]. The first element involves the representation of abstract concepts as physical entities, and covers the symbol set used in the visualization. The second set of criteria deal with the visualization itself. These criteria will be used in assessing the effectiveness of KScope.

3. Implementation

3.1 Architecture

The implementation of the KScope tool is based on the following of components (see Figure 1):

- KScope Visualization System – acts as the main driver and calls the other modules in the order shown along the arrow connectors
- Program Parser Module – extracts meaningful information about the structure of the program under inspection
- Spatial Coordinate Module – determines the location in 3D space of each of the scene objects
- Display Module – maintains the interactive 3D environment on a chosen visual system

Java, a reflective language, was chosen as the language to be analyzed by this first iteration of the KScope visualization tool. The extensive Java Byte Code Engineering Library (BCEL) [??] is freely available and allows for easy implementation of the parsing stage. The initial symbol set is made up of simple primitive objects including cubes, pyramids, and lines that can be rendered quickly in the display component. KScope uses the Java3D graphics library because of its cross-platform capabilities and the ease of configuring across various display devices.

Our sample test case is based on classes and interfaces coded in such a way as to illustrate five UML relationship types represented within the main class, and within the classes and interfaces used by the main class. Classes are named to show their relationships with the class under analysis. For example, the parent class of Child_Main_1 class is named "Parent." The types of relationships illustrated include: inheritance, association, dependency, composition, and implementation. *Inheritance* in Java is based on extending the functionality of a parent class, while *implementation* is based on an implementing class defining the methods of the interfaces. There is an *association* relation between classes when a reference to a class object existing outside the class under analysis is passed as an argument to a class constructor. A *dependency* exists where a class object is an argument to a method of the class under analysis. A *composition* relation exists when a class object is created within the class under analysis and has a lifetime less than or equal to the lifetime of the class (e.g., a class object as an attribute of a class).

Figure 2 shows our test case as analyzed by the 2D visualization system, Together Version 6.0 [9]. The entire static class representation uses standard UML notation to represent the various relations.

Figure 3 shows the same example test case as analyzed by KScope. In KScope, coloring is used as a significant element in defining symbols. A multicolored cube represents the main class (i.e., the class containing the main method) under analysis. The cube shape is used to indicate a class, while a pyramid indicates an interface. The dark blue shaded cubes represent what are called terminator classes, which are those classes defined outside the directory of the class under analysis, including the standard Java library classes and other predefined libraries. Terminators are primarily used to limit the extent of the analysis. The light purple pyramids represent interfaces, while the light green pyramid represents a terminator interface. The color of the connecting line indicates the relationship of the connected objects: association is red, dependency is blue, composition is magenta, implementation is black, class inheritance is green, and interface inheritance is yellow.

Two forms of text-based information are available to the user: class and interface names, and additional information displayed by selection with the left mouse button. For classes and interfaces in the current directory, a BCEL-derived analysis appears; in the case of terminators, the appropriate JavDoc appears.

3.2 Object Placement

Within KScope spatial orientation is used to indicate the direction of relationships. Class inheritance proceeds upwards, as indicated by the green vertical lines connecting classes. An interface is placed on the horizontal plane of the class that implements it, while interface inheritance, like the class method, is indicated vertically. The composition, dependency and association relations are placed below the main class with appropriately colored connecting lines. Terminators are place above the main class.

The placement of each type of object is determined by separate placement algorithms. The main class is always placed at the origin of the display universe (0, 0, 0). Classes that are part of an inheritance hierarchy are placed above their respective child class (i.e., parent classes are always higher than their children). Related classes are set in place based on conical calculation; that is, a count of the related classes is used to divide a circle into equal arcs with classes placed at the end point of each arc. The entire circle is displaced on the negative Y axis based on the generation of the parent class. The radius of a placement circle is based on the number of objects in the set and the number of generations below the main class. The terminator classes are placed in a similar manner with displacement on the Y-axis in a positive direction. Interfaces are offset in the positive X direction with inheritance in a vertical displacement.

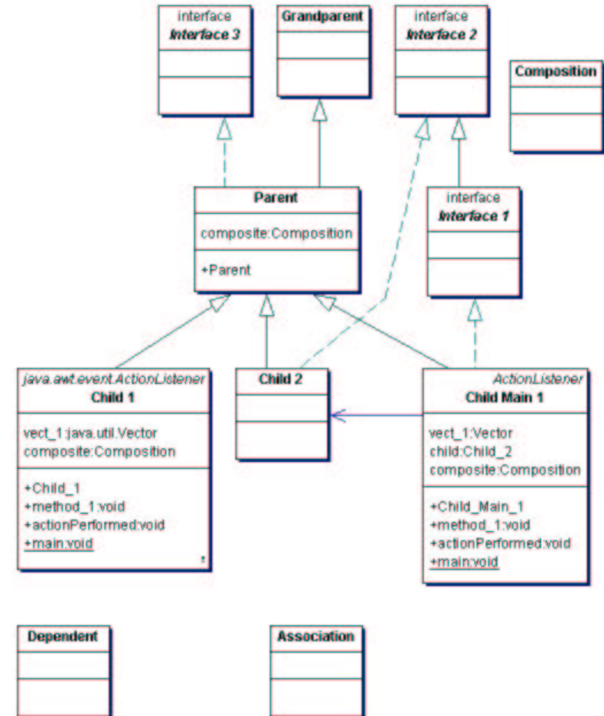


Figure 2: Together 6.0 class analysis

3.3 User Navigation

User navigation is performed via the keyboard (see Table 1).

Table 1 Keyboard navigation

Key	Action
←	move viewpoint left
→	move viewpoint right
↑	move viewer toward objects
↓	move viewer away from objects
page up	move visual objects up (viewpoint down)
page down	move visual objects down (viewpoint up)
=	return to start view

4. Results

Figure 3 shows the initial view of the test case under analysis. The user can select (through a drop-down menu) three other views: related classes, interfaces and their related classes, and terminators, as shown in Figures 4, 5 and 6, respectively. In all of the views, the entire software representation initially rotates at a constant rate.

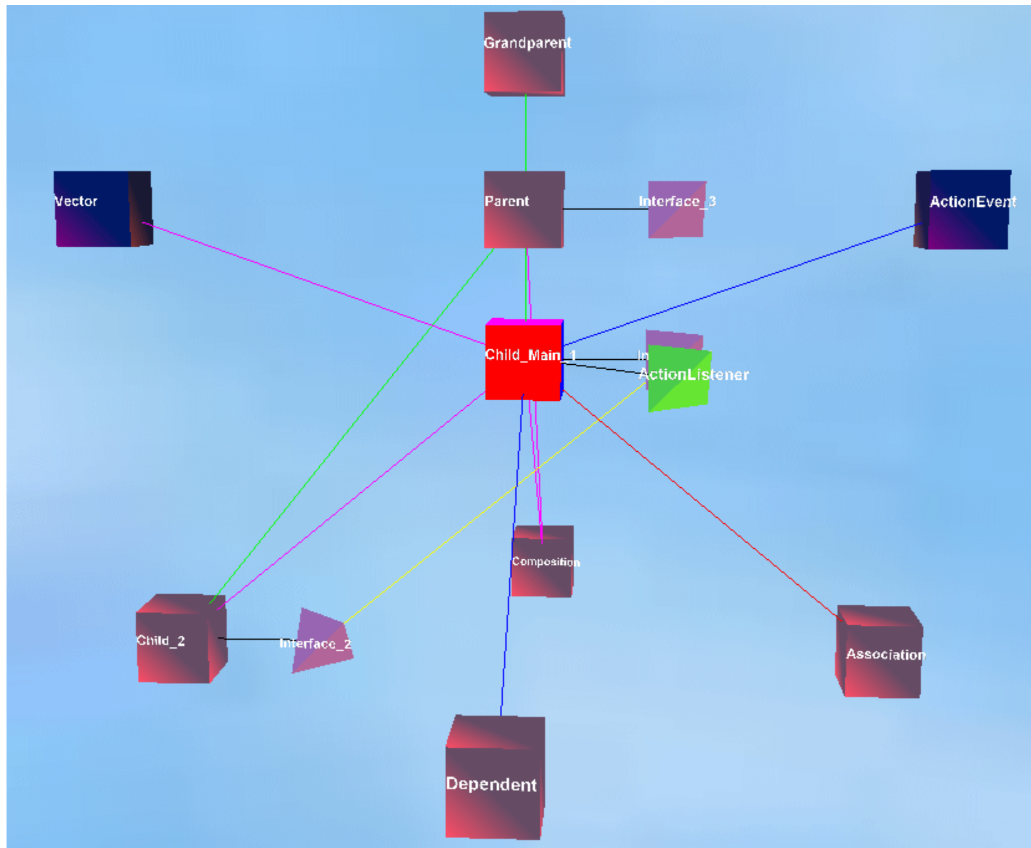


Figure 3: KScope: all views

The user has the option to start or stop the rotation as desired. Note that Figures 2 and 3 analyze the same system.

In each view an object can be inspected by left clicking on it. This brings up a pop-up window (see Figure 7) with a textual description of the class or interface, as explained in Section 3.1. Various details of the class are listed within this text area. Additional items can also be included; the items shown were selected for brevity.

A complex system can create a complex visualization, as shown in Figure 8 (KScope's self analysis); therefore, the system includes a user choice of level of detail. The user can right click on a class to change the view to one that contains the class and terminators related to that class (see Figure 9).

4.1 Evaluation

The evaluation of KScope is based on the criteria discussed in Section 2 [11]. First, the abstract representation contained in the system is considered:

- individuality – color is used to express the individuality of the types of objects
- distinctive appearance – classes are distinguished from interfaces by 3D shape
- high information content – colors, shape and text are used to present data to the user
- low visual complexity – primitive shapes, such as the cube and pyramid, keep visual complexity low
- scalability of visual complexity and information content - KScope performs well for small to medium sized programs, but the scalability to large programs has yet to be demonstrated
- flexibility for integration into visualizations - the flexibility of the overall system is reduced when color is applied as a distinguishing characteristic to the object symbols (i.e., color is no longer available as a characteristic of some other aspect of the analysis); however, such tradeoffs are often necessary
- suitability for automation - selection of simple primitives require low processor time to display

When considering the visualization criteria as pertaining to KScope the following is noted:

- simple navigation with minimum disorientation – the small number of keyboard and mouse commands makes KScope's user interface extremely simple
- high information content – large amounts of information is presented in each view
- well-structured with low visual complexity – ar-

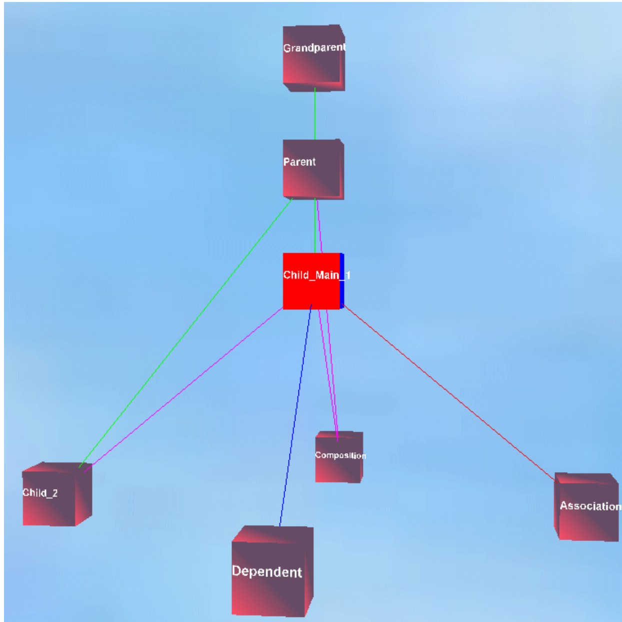


Figure 4: KScope: “Relationships” view

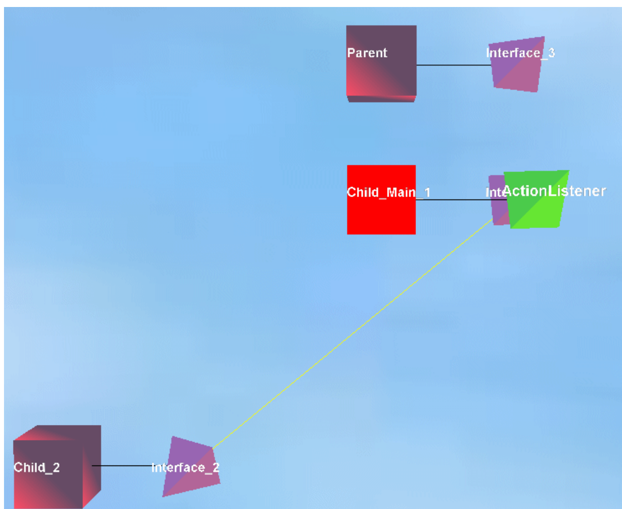


Figure 5: KScope: “Interfaces” view

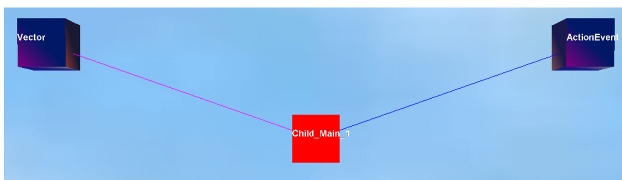


Figure 6: KScope: “Terminators” view

rangement of the objects in hierarchical cone-like formations represents a structuring of the visualization; however, the arrangement of objects with a multitude of connector lines does lead to a complex visualization; rotation of the scene, as well as selectable views for reduced object sets and relationships, gives the viewer an enhanced understanding of the virtual space and increases the ability to compre-

Child_Main_1

//
className = Child_Main_1
Class Type = Concrete
//
Parent 1 classname = Parent
Parent 2 classname = Grandparent
Parent 3 classname = java.lang.Object

Interface 0 name = Interface_1
Interface 1 name = Interface_2

Fields:
java.util.Vector vect_1
Child_2 child
Composition composite

Methods:
public void <init>(Association arg1)
public void method_1(Dependent arg1)
public void actionPerformed(java.awt.event.ActionEvent arg1)
public static void main(String[] arg0)

Figure 7: Class pop-up

hend the relationships of the objects

- varying levels of detail - the user’s ability to select a class and view the related terminals is a direct application of this criterion
- resilience to change – as the views are changed, relationships of objects is maintained in either a subtractive or additive selection of views
- good use of visual metaphors – no attempt was made to find visual metaphors
- approachable user interface – the limited controls and simple selection process of the user interface satisfy this criterion
- integration with other information sources – the text-based representation of the object integrates the 3D view with another source of information
- good use of interaction – user interaction with the objects is shown through the ability to navigate the visual space and to select alternate sources of information
- suitable for automation – the system is fully automated in the generation of the visualization.

5. Conclusion and Future Work

Future work will focus on additional spatial coordinate selection algorithms, symbol sets, and language parsers. Further additions to the symbol library could be achieved by extending the “geon” symbol alphabet [3] into a virtual environment and thus, greatly expand the number of geon-defined objects and the variety and number of relationships expressed in a single view.

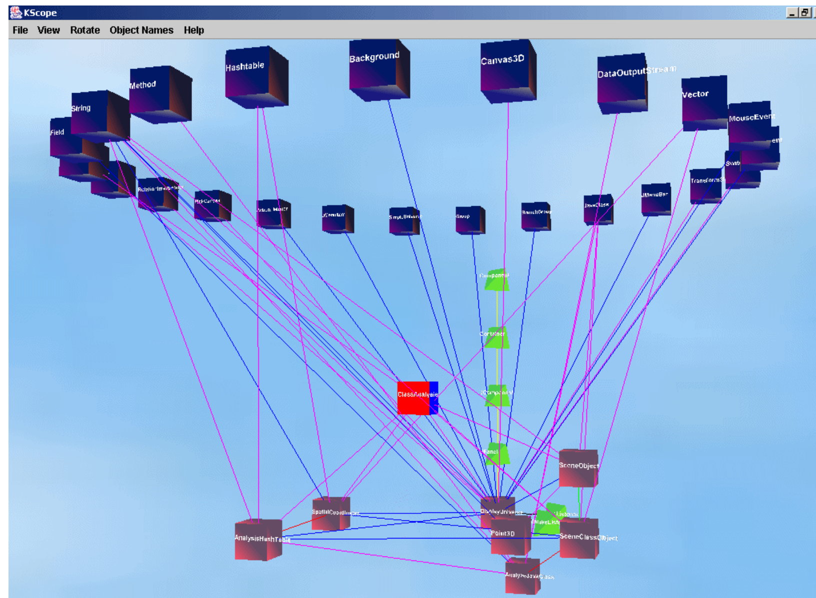


Figure 8: KScope analyzes KScope

The modular structure of KScope lends itself to various extensions. Improvements to the system would involve additions to the current number of alternative modules. By increasing the variety of components, such as additional selections of the type of spatial coordinate placement algorithms, and additional symbol sets for the symbol library module, the tool will be made the basis for a series of experiments as to the quality of information presented to the user and the user's ability to comprehend the information.

Overall KScope presents the visualization of software in an interesting and visually stimulating manner. The colors, spatial placement, alternate views, multiple methods of presenting data, and viewer interaction create an effective educational environment for the user. Increasing the functionality of the system and experimenting with different configurations will build on the strengths illustrated by KScope.

6. References

[1] K. Alfert and F. Engelen, “Experiences in 3-Dimensional Visualization of Java Class Relations,” *Transactions of the SDPS,* September 2001, Vol.5, No. 3, pp 91-106.

[2] I. Biederman, “Recognition-by-Components: A Theory of Human Image Understanding,” *Psychological Review*, Vol. 94, No. 2, 1987, pp 115-147.

[3] P.Irani, C.Ware and M.Tingley, “Using Perceptual Syntax to Enhance Semantic Contents in Diagrams,” *IEEE Computer Graphics and Applications*, (in Press)

[4] D. F. Jerding and J. T. Stasko,”Using Visualization to Foster Object-Oriented Program Understanding,” *Technical Report GIT-GVU-94-33,* July 1994.

[5] C. Knight and M. Munroe, “Virtual but Visible Software,” *Visualization Research Group*, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, UK. 1998.

[6] C. Knight and M. Munroe, “Comprehension with[in] Virtual Environment Visualization,” *Visualization Research Group*, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, UK. 1999.

[7] H. Koike, “The Role of Another Spatial Dimension in Software Visualization,” *ACM Transactions of Information Systems*, Vol. II, No. 3, July 1993 pp 266-286.

[8] J. I. Malectic, A. Marcus and M. Collare, “A Task Oriented View of Software Visualization,” *VISSOFT 2002*, June 26, 2002.

[9] http://www.borland.com/together/index.html, 2003.

[10] C. Ware, D. Hui and G. Franck, “Visualizing Object Oriented Software in Three Dimensions,” *CASCON 93 Conference Proceedings*, Toronto, Ontario, Canada, October, 1993, pp 612—620.

[11] P. Young and M. Munroe, “Visualizing Software in Virtual Reality,” *6th International Workshop on Program Comprehension: IWPC'98.*

Figure 9: Level of detail

Self-Organizing Maps Applied in Visualising Large Software Collections

James Brittle and Cornelia Boldyreff
Department Of Computer Science
University Of Durham
{*j.g.brittle,cornelia.boldyreff*}*@durham.ac.uk*

Abstract

The self-organizing map's unsupervised clustering method can be used as a data visualisation technique. Within this paper different techniques to visualise self-organizing maps (SOM) and their effectiveness are investigated in relation to the organisation of a large software collection and its visualisation.

GENISOM, an offspring component of the GENESIS software engineering platform, incorporates the generation, maintenance and viewing of Self-Organizing Maps.

The results from our studies indicate that a hybrid of 2D and 3D visualisations is favoured by users. Extensive usability tests also show that the majority of users found that the additional information a SOM provides, aids browsing and searching of a software collection. Further work is addressing the problems found in the application of SOM within a software engineering environment.

Key Words: Self Organizing Maps, GENISOM, software visualisation

1 Introduction

Interactive exploration of a large software collection or large software systems, where the user looks at individual artefacts one at a time would be greatly aided by ordering them according to their contents.

There exist many possibilities to achieve this organisation, e.g. as a graph or a hierarchical structure. A common organisation is one in which the artefacts are represented by points on a 2D plane and the geometric relations between them relate to their similarity. Such representations are often called document maps. These organised collections of data facilitate a new dimension in information retrieval namely the possibility to locate pieces of relevant or similar information that the user was not explicitly looking for.

Document maps can be constructed through a number of methods including the data visualisation technique, the Self Organizing Map (SOM), invented by Prof. Teuvo Kohonen in the early 1980s [7]. SOMs are an unsupervised, clustering algorithm, which use neural networks. They have been demonstrated to aid programmers in the process of reverse engineering by discovering common features within legacy code [2] and to assist in object recovery[1], although visualisation of the associated maps is not explicitly considered in reports of this research.

A prominent problem within the field of Software Engineering concerns reuse. Reusable assets are in abundance, over the web and in libraries but it is extremely difficult to locate reusable software artefacts that are relevant to a particular application. The necessary organisation is often lacking and difficult to achieve given the dynamic nature of such software collections. This problem can also be found where a large evolving software system consists of an ever growing number of components and the managment and hence the comprehension of the associated software artefacts tends to be increasingly difficult.

Having suitable visualisations of such software collections can mitigate the problem identified above. The application of information visualisation builds upon the strenghts of humans and computers as

> "by properly taking advantage of peoples' abilities to deal with visual presentations, we may revolutionise the way we understand large amounts of data" [5]

Within this paper the use of Self Organising Maps as a means of visually presenting large software collections is demonstrated. Section 2 describes the implementation of GENISOM. Section 3 details the different visualisations the software can produce. Section 4 presents the results of the evaluation of the tool and subsequent adaption of maps. Section 5 identifies further work.

2 GENISOM

GENISOM is a client/server application designed to manage and enable viewing of SOMs. Figure 1 illustrates a simplified architecture diagram for the GENISOM system.

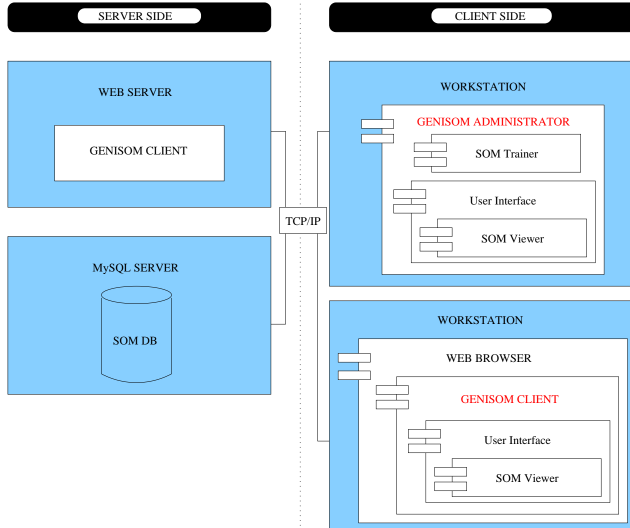


Figure 1. GENISOM architecture

The GENISOM Administrator is used to manage SOMs; it enables their creation and maintenance. The generated SOMs are then stored within a MySQL database, from which the GENISOM Client can then retrieve the data and display the generated maps. The Client is web based using Java Web Start[1] to aid its accessibility to users. The architecture of the system enables distributed software engineering teams to work together (see Figure 2).

Input data to create a SOM can be descriptions of any content as long as it is stored within a database table. For example each record could be a description of a reusable artefact or of a software component within a large software sytem such as a method in a class.

There are two main use case scenarios for the system; firstly the Administrator could be used by the librarian of a reuse library and the Client by the system developers (see Figure 3), therefore aiding them in the location of reuse candidates. Secondly both tools could be used by members of a software development team to aid program comprehension, or help in decisions regarding restructuring and reengineering of the system.

3 Visualisations of Maps

The GUI for the SOM evolved through a number of stages, due to the interactive design approach adopted.

[1] http://www.java.sun.com/products/javawebstart/

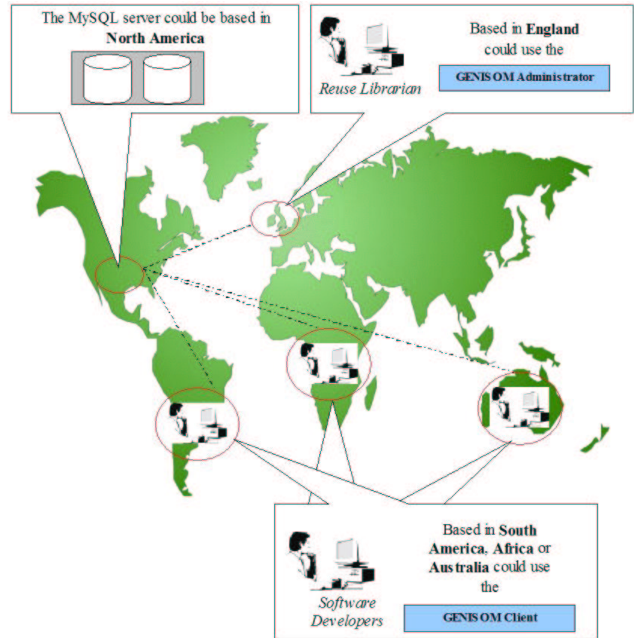


Figure 2. Possible geographic layout of GENISOM system

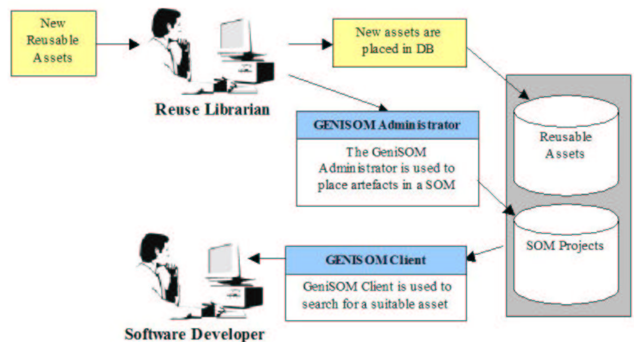


Figure 3. Reuse infrastructure with GENISOM system in place

Earlier designs drew inspiration from and were similar to WEBSOM [8], a web based example of the use of SOMs for organisation of document collections. Feedback through quick and dirty evaluations [12] was critical of the initial colour scheme used. This resulted in the final 2D map as illustrated in Figure 4. The improved colour scheme is based upon the use of the primary colours, essentially to help distinguish different elements of the map more clearly than the heatmap approach applied in WEBSOM.

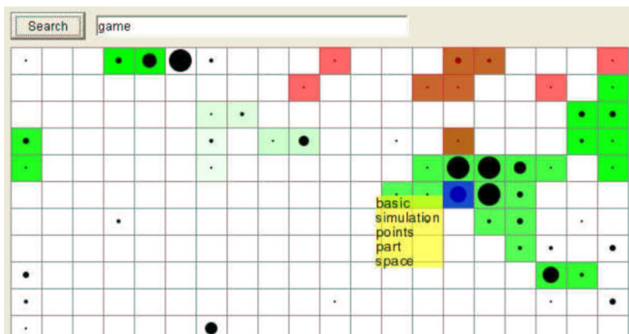


Figure 4. Screenshot of 2D Map

The neural net is arranged as a grid with the inputs (e.g. reusable artefacts) being attached to the neurons. The black dots on grid cells (i.e. neurons) indicate that inputs have been matched to them, with the size of the dot representing the number of them. The green shading of the grid cells indicates the boundaries of similar clusters of neurons.

The map is interactive allowing the user to select a particular neuron (the blue cursor indicating the selection), this displays the neuron's labels, five or less words that best describe the inputs matched to it (see Figure 5). This labelling approach, taken from LabelSOM [13], is a commonly used method within SOM software. As well as displaying the labels for a selected neuron, details of the inputs matched to it are also displayed in a side bar though this is not depicted in the figures.

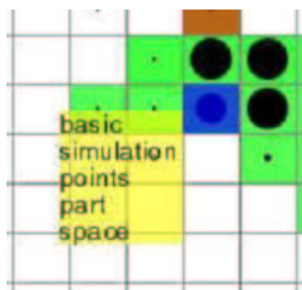


Figure 5. Closeup screenshot of 2D Map

Browsing is aided by a search system coupled to the map which highlights the results in red for a certain search string.

Using 2D limits the amount of information that can be visualised. Improvements to the GENISOM GUI therefore naturally led on to the development of a 3D map using the Cityscapes technique which has already found application in software visualisation [6]. Using this technique in GENISOM each value is plotted as a column (or 'building'). The 'buildings' are plotted on the same horizontal plane, allowing differences in height and position to be analysed. In relation to a SOM, each building represents a neuron and the height of the building relates to the number of inputs that are matched to it. The implementation of this used Java3D[2] and in the final system the option was made available to switch between using 2D and 3D interfaces.

Figure 6 illustrates the 3D map; the user's view can be rotated and zoomed in and out. Furthermore, the user can also interact with the 3D map in the same manner as the 2D map following the same colour scheme (see Figure 7).

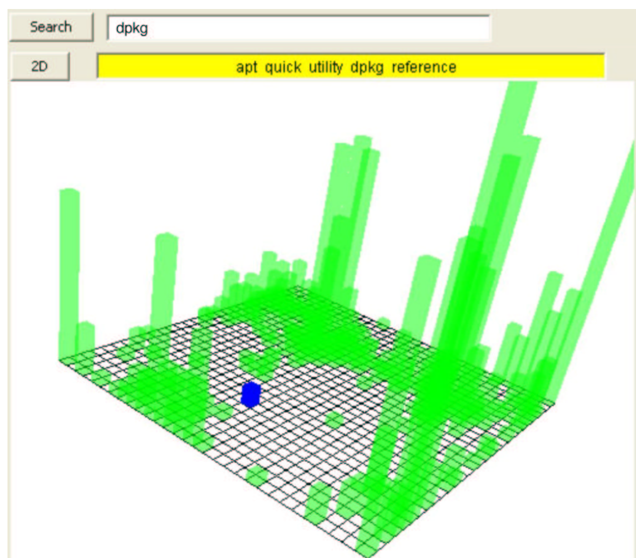


Figure 6. Screenshot of 3D Map

4 Evaluation and Results

A series of different evaluations were carried out to assess the success of the GENISOM software. As test input data, a selection of 300 Debian Packages were gathered from the Debian Project. This is an open source development of a free OS, Debian GNU/Linux. New contributors to the project as well as users interested in studying the Debian project could be the potential beneficiaries of these maps. In the evaluations all participants were from the Department of Computer Science at the University of Durham.

[2] http://java.sun.com/products/java-media/3D/

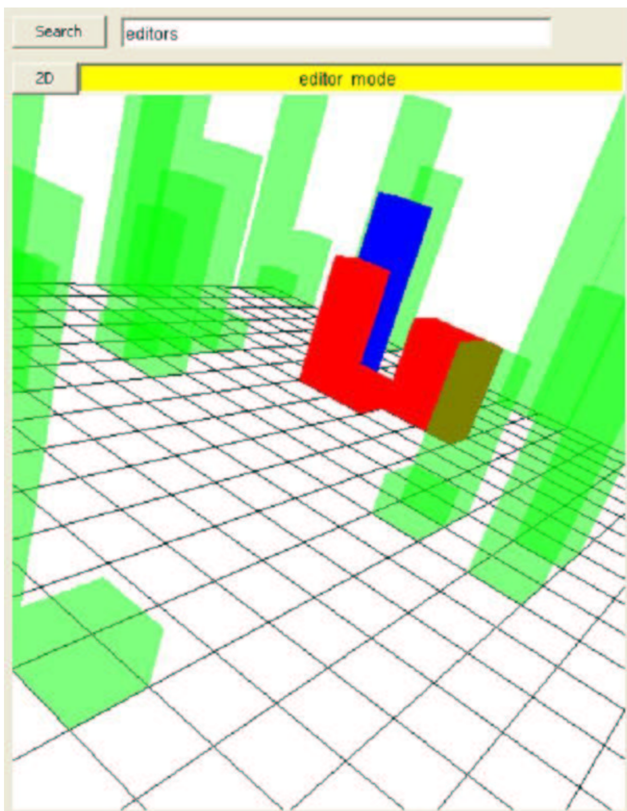


Figure 7. Screenshot of 3D Map

4.1 Administrator

The Administrator was assessed for usability as it is a critical area of the software, in that the user should be able to generate maps proficiently. Results from a heuristic evaluation showed that the technical terminology used within the software and also the task of assessing the 'goodness' of the SOM produced could be problematic. The former was found to be the case after carrying out usability testing and questionaires. The anticipated problem of assessing the 'goodness' of a map did not actually occurr in practice during the evaluations.

Overall the Administrator has a steep learning curve, which is to be expected with such a technical piece of software. However it is predicted that once the terminology (i.e. language) barrier has been overcome, operation of the system can be quickly mastered with training. Trials carried out with software engineering researchers proved this to be the case.

4.2 Client - 2D versus 3D

Secondly the Client was evaluated, in two distinct trials. The first of these involved a comparison between the two types of interface with respect to usability and efficiency. A population of 10 computer scientists carried out a series of usability exercises, which involved searching for information held within the map.

Statistical results from these evaluations indicated that the 2D interface was superior to the 3D. The 2D was more efficient with respect to the time required to complete the tasks and more effective due to the observed signs of frustration shown by the users. All participants considered that the Cityscape technique did inform them far better on the spread of the population (i.e. the heights of the buildings were more intuitive indicators of the population density than the different sizes of black dots), however this feature was not considered a necessity. Overall an overwhelming majority, 90%, preferred to use the 2D interface.

Reasons for their opinions mostly arose from the difficulty in selecting a 'building' (i.e. a neuron) in the Cityscape, as certain buildings could obscure others meaning that the view of the scene would have to be changed to enable selection of the obscured buildings. Other comments noted the poor navigation through the 3D scene, which is a common problem with such applications. It is difficult to control the 3D space with interaction techniques that are currently in common use since they were designed for 2D manipulation (e.g. dragging and scrolling) [11].

Overall conclusions drawn from the evaluation were that the combination of the two maps actually complemented each other. The 2D map was quick and simple to use; however, the 3D map did add functionality to the browsing

which concurs with the findings of Koua and Kraak [9].

4.3 Client - Major Evaluation

In the second part of the evaluation of the Client, a key question sought to determine whether using SOMs was an improvement against other methods. This led to the creation of the GENISOM Testbed, an application that could be downloaded by participants and would essentially guide them through an evaluation while automatically logging the results.

The Testbed consisted of two searching tasks using the 2D map and as a comparison two tasks using a standard search system, similar to the functionality of the Google web search engine. It also included a questionnaire to record the users' experiences.

114 participants took part in the evaluation, on the basis of the statistical results the 2D map proved to be less effective, as it took users longer to complete set tasks with the 2D map. User opinions on the effectiveness of the new system were mixed though; over 30% thought that it was more effective and only 52% stated it was less effective. Overall, however, the evaluation statistically regarding the times to complete tasks was seen to be flawed due to the difference in learning curves of the two systems. The 2D map provides a fairly complex interface and therefore has a steep learning curve, while users will already have obtained most knowledge of how to operate the standard search system due to prior experience with web search engines.

Predictions regarding future fair usability tests are promising for the SOM as users were found to vastly reduce their times for the second task using the map even though the second task was considerably more difficult. Also participants' views on whether the additional information the SOM provided, aided searching and browsing showed that over 68% thought this to be true, which concurs with Merkl's findings:

"such a library system has its benefits when developers look for a particular component during the development process of a software system. Additionally the self-organizing map can be used easily for interactive exploration of the software library - a feature that is of vital importance in software reuse"[10]

Although Merkl's paper is very positive towards the application of Self-Organizing Maps to reuse libraries, its results were subjective. The asset population was small in comparison to what would be found in a normal reuse library and there was no formal evaluation of the prototype system developed.

4.4 SOM Algorithm

Evaluation of the algorithm behind the Administrator consisted of creating a number of different maps with varying input size, 25 to 13000 Debian Packages. The very nature of the standard SOM algorithm is that a number of trial maps need to be created to achieve the best map:

"an appreciable number (say, several dozens) of random initialisations...and different learning sequences ought to be tried, and the map with the minimum quantization error selected."[7]

The results from the evaluation back up this theory, however, they present a possible problem for the use of SOMs in this application. Generating maps is a time consuming activity due to the memory latencies of self-organizing neural networks. Coupled with the factor that many maps will need to be trialled the feasabilty of using the standard SOM algorithm in a software engineering environment is fairly low, unless the software collection is stable and justifies the initial investment in the map production. Further work into researching more advanced algorithms is therefore a main priority of our work.

4.5 Outstanding Problem

Feedback highlighted one major criticism of both the GENISOM 2D and 3D map interfaces. The criticism was that initially the view of the maps was 'uninformative' as information, i.e. grid cell label, was only displayed once the cell had been selected. Therefore the user must systematically select cells to build up a mental model of the map's contents. However, displaying all the labels at the same time led to a cluttered map therefore it was decided let the user have control over which label was displayed. A better mechanism for providing the user with an overview of the map's contents other than exposing the textual labels over the whole map is being sought. Currently structuring the SOM hierarchically is being investigated.

5 Conclusions and Further Developments

The evaluation of GENISOM highlighted many different areas of improvement. Firstly regarding the SOM algorithm, a hybrid algorithm the Growing Hierarchical Self-Organizing Map (GHSOM)[3] is proposed as a replacement. This has a number of benefits: the training process is virtually automated minimizing the number of parameters and in theory allowing the best map to be generated on the first attempt. This also has the advantage of simplifying the generation of the maps, which was found to be a troublesome area within the Administrator.

The GHSOM also enables a hierarchical structure of SOM to be built up, firstly this would aid visualisation of

the information in respect to the SOM's organisation and the earlier stated problem regarding labelling of the map. Secondly if actual source code was used as input data then it would facilitate object finding as demonstrated by Chan and Spracklen [2].

Regarding the different visualisations of SOM, the presented results suggest a combination of the 2D and 3D maps. Research into combining the two would be worthy of further consideration. Investigating other 3D techniques to visualise SOMs may lead to conclusions on whether the technique of Virtual Data Mining is applicable in this application.

The integration of the software to the GENESIS platform [4] is also proposed, giving scope for real life trials of the software and actual use by software engineering professionals. Investigations could also be carried out into the applicability of the SOM not just to software components but to a variety of reusable artefacts: test cases, designs and documentation. This would allow the possibility of further work following the findings of Chan and Spracklen to investigate in more detail the software analysis properties of the SOM.

References

[1] A. Chan and T. Spracken. Discovering common features in software code using self-organizing maps. In *Proceedings of the International Symposium on Computational Intelligence*, Kosice Slovakia, August 2000.

[2] A. Chan and T. Spracklen. Object recovery using hierarchical self-organizing maps. In *Proceedings of the International Conference on Engineering Applications of Neural Networks*, Kingston Upon Thames UK, July 2000.

[3] M. Dittenbach, D. Merkl, and A. Rauber. The growing hierarchical self-organizing map. In *International Joint Conference on Neural Networks*, volume 24, Como, Italy, July 2000.

[4] M. Gaeta and P. Ritrovato. Generalised environment for process management in cooperative software engineering. In *International Computer Software and Applications Conference*, volume 26, pages 1049–1059, Oxford, England, August 2002. IEEE.

[5] C. Glymour, D. Madigan, D. Pregibon, and P. Smyth. Statistical themes and lessons for data mining. In *Data Mining and Knowledge Discovery*. Kluwer Academic Publishers, 1997.

[6] C. Knight. *Virtual Software In Reality*. PhD thesis, Durham University, 2000.

[7] T. Kohonen. *Self-Organizing Maps*. Information Sciences. Springer, second edition, 1997.

[8] T. Kohonen, S. Kaski, K. Lagus, J. Salojarvi, J. Honkela, V. Paatero, and A. Saarela. Self organization of a massive document collection. In *IEEE Transactions on Neural Networks*, volume 11, pages 574–585, May 2000.

[9] E. Koua and M. Kraak. An evaluation of self-organizing map spatial representation and visualization for geospatial data: Perception and visual analysis. Technical report, International Institute for Geo-Information Science and Earth Observation (ITC), 2001.

[10] D. Merkl. Self-organizing maps and software reuse. In *Computational Intelligence in Software Engineering*. World Scientific, 1998.

[11] J. Nielsen. 2d is better than 3d. useit.com - Jakob Nielsen's Alertbox, 1998.

[12] J. Preece, Y. Rogers, and H. Sharp. *Interactive Design: beyond human-computer interaction*. John Wiley and Sons, 2002.

[13] A. Rauber. Labelsom : On the labeling of self-organizing maps. In *International Joint Conference on Neural Networks*, 1999.

The end of the line for Software Visualisation?

Stuart M. Charters, Nigel Thomas and Malcolm Munro
Visualisation Research Group
Department of Computer Science
University of Durham,
South Road,
Durham,
DH1 3LE, UK
S.M.Charters@durham.ac.uk Nigel.Thomas@durham.ac.uk Malcolm.Munro@durham.ac.uk

Abstract

This position paper addresses the issue of how software visualisation should develop in the future. A number of useful visualisations have been developed by the software visualisation community but these have usually been through standalone tools. Is it now time to consider if and how these visualisation tools can be integrated into development environments and be used as roundtrip visualisation tools. If this is not addressed then software visualisation research may have come to a useful end.

1 Introduction

Software visualisation is a relatively young research area where great progress has been made in developing ideas, representations and tools to aid program comprehension during the maintenance and evolution of software. This development is paralleled in the software development community where visual representations of systems have been used to help in for example requirements capture and design [11] [7]. The representations and tools to aid program comprehension take a variety forms from animations of algorithms and data structures, through dynamic run-time information, to tools which present static view of software structures linked to source code views [8] [9] [10] [3] [2] [4]. Despite these advances it is time to question if the time has come when no new advances are being made and all advances are just variants on the old theme.

Software development is an evolutionary process, often one of iterative refinement. Segments of code are written, tested, refined and built upon. Software visualisation needs to support this iterative process, if tools are stand alone the effort to evaluate changes to code using those tools is considerable.

In an ideal world the maintenance and evolution of a system is carried out on the appropriate level of structure within the lifecycle model because of the in built traceability between the documents that result from each phase. For example, corrective maintenance is concerned with the code, and changes are made at this level, whereas perfective maintenance is concerned with changing requirements and hence the changes should be made to the requirements and then be reflected through to changes in the design documents and the code. In practise however, maintenance and evolution is carried out at the code level because the traceability has been destroyed through excessive maintenance or it never existed in the first place. In this situation changes are rarely reflected in the other lifecycle documents that define the system. Thus the need for program comprehension supplemented by visualisation.

It is recognised that program comprehension occurs in different ways, top-down, bottom-up or a combination of the two [12]. This program comprehension can be systematic or as-needed, the integration of visualisation allows developers and maintainers to use whichever strategy is required or suits them best to achieve the understanding they require to make changes. Another view of program comprehension is the feedback loop strategy, where the program is compared against the mental model of the problem solution held by the developer. The use of visualisation throughout the implementation phase would complement this feedback loop strategy as the developer saw the program growing visually allowing them to continually compare this against their mental model. Work has already been done to integrate different views to allow the use of different comprehension strategies within a number of tools [9].

2 A Simple Analogy

This section draws a simple parallel between the development of HTML pages and program code. It is not intended to be a comprehensive comparison but to act as a simple analogy to illustrate a possible way forward for software visualisation.

When developing an HTML page one approach is to use a simple text editor to write raw HTML and then to view that HTML in a browser. In the maintenance of that HTML page the maintainer will iterate between the editor and the browser, always changing the HTML in the editor. The use of the browser can been seen as using a visualisation of the HTML code to check that it is correct and to get some understanding of how the HTML works. This type of use is termed a one-way trip, in that the code (HTML in this instance) is edited and the visualisation is used to give some understanding. This can be seen as a simple analogy with one way that program code is developed and where visualisation is used to help understand some aspect of that code. The programmer will use simple visualisation tools (such as call graphs and control flow diagrams) to supplement their understanding of the program source code.

Another way to develop HTML pages is to use a development environment such as Dreamweaver. Here the main interface is a visual one that allows WYSIWYG layout and editing of the underlying HTML without having to resort to understanding the HTML. Changes in the visual interface generate or update the underlying HTML. In addition these development environments allow the direct editing of the underlying HTML and changes made via the textual representation are reflected in the visual interface. This type of use is termed roundtrip editing.

Of course it is recognised that Dreamweaver is a tool with complete integration that operates on a somewhat restricted language (HTML) that is inherently visual. The Dreamweaver type of tool [5] [1] [6] works with HTML because there is a one to one mapping between the HTML tags and the visual representation. With software visualisation tools this one to one mapping is less obvious and harder to achieve due the complexity of programming languages and the nature of the visualisations.

Visualisation of software systems show the relationships between the components of the system at different levels of granularity and at different levels of abstraction. They come in many forms and range for example, from high-level architectural representations, through design notations (UML) to structural representations such as call graphs and control flow diagrams. To these may be attached attributes that show, for example, the relationships between names (variable, class etc.) used in systems. These visualisations have been shown for example, using conventional node and arcs, tables, and grids, and have utilised real life and abstract metaphors in a two-dimensional or a virtual reality world.

3 Roundtrip visualisations

Roundtrip visualisation is used to describe visualisation systems that are linked with the data from which they are generated in such a manner that changes to the underlying data updates the visualisation and changes made through the visualisation itself are reflected in the underlying data. An example of roundtrip visualisation for software is the construction of a visualisation that represents the class structure of a Java project and where if the structure of the classes is modified then the visualisation is updated and similarly if the visualisation is used as a mechanism to restructure classes then the code reflects that restructuring.

Current visualisation tools tend to be one-way trip. One model is where the source code is edited and this is reflected in the visualisation but not the other way round. An example of this is a call graph visualisation linked to a source code editor that changes as the calling structure of the code is modified. A further one-way trip mode is where the visualisation is edited and this is reflected in the code but not the other way round. An example of this is the JBuilder GUI designer where the visualisation consists of a canvas and a palette of GUI components. The canvas can be 'edited' in order to change the GUI and these changes are reflected in the java source code. However if the generated GUI java code is edited directly then the changes are not necessarily reflected on the canvas.

The limiting factor is that the visualisations developed so far do not have the required properties for roundtrip visualisations. The current visualisations of software are at the wrong level of abstraction or of the wrong granularity and thus are one-way trip visualisations.

4 Conclusion

Current progress in software seems to confined to:

- improving abstractions to reduce information overload;
- developing new representations using abstract or real world metaphors; and
- improving layout of existing representations;

and are instantiated in one-way trip standalone tools. These are all laudable research aims and can sustain software visualisation research for a while longer.

The way forward for software visualisation is to address the issues of roundtrip visualisation. To support roundtrip visualisation an alternative approach is required, partial integration with the development environment is needed to

allow for access to the source data by the visualisation and for changes made in the visualisation to be reflected in the development environment. However this integration must be sufficiently flexible, for example by the use of a standard integration method for visualisations, that different types of visualisations can easily be integrated and that visualisations can be integrated with different development environments. The roundtrip nature of the integration would need to ensure that changes made using the visualisation are reflected in the source and that changes to the source were reflected in the visualisation.

If this issue is not addressed then it really is the end of software visualisation. We must develop new visualisations that can easily integrate as roundtrip visualisations.

References

[1] Adobe. Pagemill. *http://www.adobe.com/*, 2003.

[2] J. Cain and R. McCrindle. Software visualisation using c++ lenses. *Proccedings of 7th International Workshop on Program Comprehension*, May 1999.

[3] C. Knight and M. Munro. Comprehension with(in) virtual environment visualisations. *Proccedings of 7th International Workshop on Program Comprehension*, May 1999.

[4] C. Knight, M.-A. Storey, and M. Munro. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[5] Macromedia. Dreamweaver. *http://www.dreamweaver.com/*, July 2003.

[6] Netscape. Netscape composer. *http://www.netscape.com/*, 2003.

[7] P. W. Parry, M. B. Ozcan, and J. I. Siddiqi. The application of visualization to requirements engineering. *Technical report, Computing Research Centre, Shefield Hallam University, England*, 1998.

[8] M. P. Smith and M. Munro. Runtime visualisation of object orientated software. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[9] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. *In Proceedings of the IEEE Symposium on Information Visualization*, 1997.

[10] C. M. B. Taylor and M. Munro. Revision towers. *First IEEE International Workshop on Visualizing Software For Understanding And Analysis*, 2002.

[11] A. Teyseyre, R. Orosco, and M. Campo. Requirements visualization. *Workshop de Investigadores en Ciencias de la Computacin(WICC'99)*, 1999.

[12] A. Von-Mayrhauser and A. M. Vanns. Program comprehension during software maintence and evolution. *IEEE Computer*, August 1995.

CFB: A Call For Benchmarks - for Software Visualization

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242 USA
jmaletic@cs.kent.edu

Andrian Marcus
Department of Computer Science
Wayne State University
Detroit, MI 48202 USA
amarcus@cs.wayne.edu

Abstract

The paper argues for the need of a benchmark, or suite of benchmarks, to exercise and evaluate software visualization methods, tools, and research. The intent of the benchmark(s) must be to further and motivate research in the field of using visualization methods to support understanding and analysis of real world and/or large scale software systems undergoing development or evolution. The paper points to other software engineering sub-fields that have recently benefited from benchmarks and explains how these examples can assist in the development of a benchmark for software visualization.

1 Introduction

Recently, the development of benchmarks has been highlighted [15] as a means to increase the scientific maturity of a discipline. Sim et al [15] detail a number of fields in Computer Science and Software Engineering that have proposed benchmarks to further research and understanding of the fields.

With regards to reverse engineering and program analysis a recent benchmark on dealing with fact extraction [16] motivated a number of improvements on tools such as cppx [3]. Also, developing a benchmark for clone detection was recently discussed at the International Workshop on Program Comprehension 2003 with a main goal of formalizing the meaning of source code clones and the like.

A number of individuals have argued for the Software Visualization community to develop a standard benchmark to support the research in the field. This important issue was discussed at the ICSE'01 Workshop on Software Visualization, VISSOFT'02, and most recently at the ACM Symposium on Software Visualization (SoftVis'03).

In particular, our recent discussions with Stephan Diehl, general chair of SoftVis'03, and Margaret-Ann Storey, an organizer for VISSOFT'02 and '03, motivated us to develop a Call-For-Benchmarks in Software Visualization. We will motivate why this may be the best means of developing a benchmark (suite) for software visualization research. We feel there is a need for a suite of problems that address different aspects of software visualization and argue for this type of approach. Additionally, we will propose a set of guidelines to help organize this call.

2 Aspects of Software Visualization

The focus of the benchmark will be to exercise software visualization systems/tools/methods in light of their applications toward supporting industrial software development, maintenance, and evolution. In order to frame this task we define five dimensions of software visualization [6]. These dimensions reflect the why, who, what, where, and how of the software visualization. The dimensions are as follows:

- Tasks – *why* is the visualization needed?
- Audience – *who* will use the visualization?
- Target – *what* is the data source to represent?
- Representation – *how* to represent it?
- Medium – *where* to represent the visualization?

These dimensions define a framework capable of accommodating a large spectrum of software visualization systems. This viewpoint subsumes such diverse topics as program visualization, algorithm animation, visual programming, programming by demonstration, software data visualization, and source code browsers. This diversity is reflected in the taxonomic descriptions of the field by researchers such as Price [9, 10], Roman [14], Myers [8], and Stasko [17].

Foremost, the benchmark should highlight different types of tasks. For instance one could propose a benchmark with the task of visualizing possible ADTs in legacy code or visualizing the run time activation of classes over a system. These are specific tasks that require (possibly) very different visualization metaphors and tools.

Before we continue this discussion let us present a general reference model for information visualization.

This will help focus the particulars of the benchmark with regard to the underlying pre-processing and analysis that must accompany any software visualization tool or method.

3 A Reference Model for Visualization

Card [1] proposes that visualization is a mapping from data to a visual form that the human perceives. Figure 1, adapted from [1], describes these mappings and serves as a simple reference model for visualization. In this figure, the flow of data goes through a series of transformations. The human may adjust these transformations, via user controls, to address the particular application task.

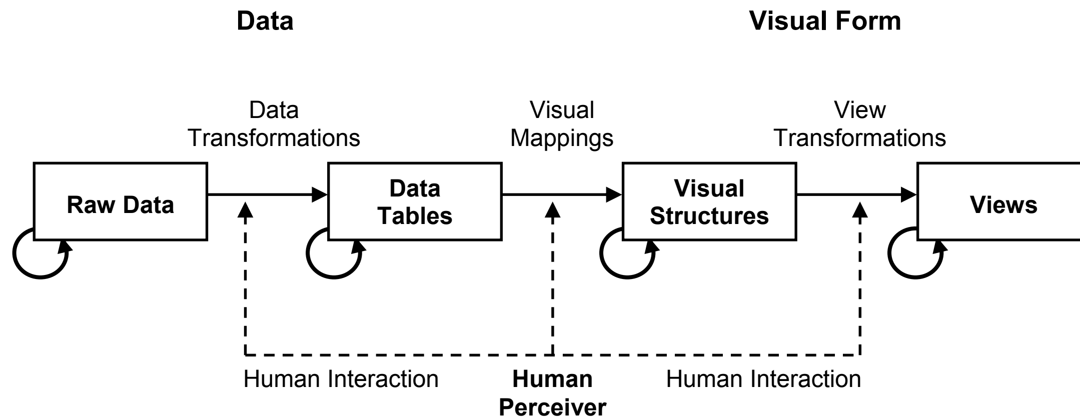
The first transformation converts raw data into more usable data tables. The raw data is typically in some domain specific format that is often hard, or impossible, to work with. This is very apparent when working with trace data generated from program executions. Data tables [1] are relational depictions of this data. Information about the relational characteristics of the data (meta data) can also be included in the data tables. Meta data is descriptive information about the data [19]. From here, visual mappings transform the data tables into visual structures (graphical elements). Finally, the view transformations create views of the visual structures by specifying parameters such as position, rotation, scaling, etc. User interaction controls the parameters of these transformations. The visualizations and their controls are all with respect to the application task.

The core of the reference model is the mapping of a data table to a visual structure. Data tables are based on mathematical relationships whereas visual structures are based on graphical properties processed by human vision. Although raw data can be viewed directly, data tables are a vital intermediate step when the data is abstract [2, 5, 12].

Software visualization maps to this reference model directly. The raw data is source code, execution data, design documents, etc. In the case of execution (trace) data, the readability is minimal. However, source code is readable, at least on a small scale, that is, one can hardly keep in mind more than a few dozen lines of source at one time. Data tables, an abstraction of the raw data, take the form of abstract syntax trees, program dependence graphs, or class/object relationships for example. A variety of software analysis tools can generate this type of data (table). Visual structures are then the software-specific visualizations we render. These visual structures are typically very specific to a particular software engineering task.

This model also points out the need to transform raw data into something more usable. This includes initial acquisition, quality, and granularity of the data. While these issues are not high profile for source code, they are a key component for dealing with the huge amounts of data that can be generated from execution traces, or from parse trees of large systems.

The software visualization process maps on top of this reference visualization model. Roman [14] and Price [9, 10], each define their own general model of the software visualization. Their views are more domain-specific and



Raw Data: idiosyncratic formats
Data Tables: relations (cases by variables) + meta data
Visual Structures: spatial substrates + marks + graphical properties
Views: graphical parameters (position, scaling, clipping, etc.)

Figure 1. Reference Model for Visualization. Visualization can be described as a mapping of data to visual form that supports human interaction for making visual sense [1].

omit aspects related to generation of views and data transformations. These models drive the definition of their respective taxonomies. We believe the general information visualization reference model should also be taken into direct consideration by a software visualization system designer.

4 Composition of the Benchmark

Given this general reference model we can now define benchmarks in terms of each of its specific components in conjunction with the task, audience, target, etc being addressed. A question that must be raised at this point is whether the underlying program/data/run time analysis methods are an integral part of the software visualization method? That is, can we (completely) decouple the visual structures and views from the underlying raw data and data tables? Obviously in general the answer to this question is no. However, the authors own work [7] along with others [18] counters this to some degree within a broad, abet limited, set of problem domains.

Of course, the concept that a software visualization tools is quite task specific and tightly coupled with the underlying data analysis is what makes construction of a single general benchmark, for software visualization, quite difficult (impossible). However, for a visualization tool to be widely utilized it should be interoperable with a variety of tools and environments.

This being the case, what then must the benchmark be composed of? We believe the general consensus is that a number of distinct problems (i.e., tasks, target, and audience) of differing domains, each with its own data set must be developed. The data set could include raw data but alternatively include data tables (or both). Providing data tables will drastically improve the ability to compare the visual aspects of methods as opposed to the underlying analysis methods.

Furthermore, the stated task of a given benchmark must be well directed at software engineering problems. We could easily fall into comparing 2D graph layout algorithms, whereas the real software visualization problem is more like the comparison of UML class diagrams layout methods (in a 2D space). Of course there must be an agreed upon quality measure. For class diagrams, recent work on the esthetics of UML diagram layout [4, 11] can help provide guidelines. In this case, the data table (UML class model) is all that is necessary.

Broader problems may include the visualization of cross cutting concerns within a given system. Here one must supply a system (raw data), with known aspects, and (hopefully) pointers to (or a list of) these aspects within the source (data tables). In this case the weaker the data table, the more of an analysis problem this becomes.

Development of a benchmark for visualizing the run time behavior of a system may be more difficult for some particular tasks. Providing an execution trace for a given system along with specific features of that trace that are deemed interesting is quite straight forward. However, developing a benchmark for visualizing the execution of a system in real time such as the research being done by Reiss [13] may be more difficult. However this could be posed as a specific question such as with debugging or bottleneck location. Of course the underlying analysis and data gathering is a permanent issue.

5 Call For Benchmarks

To develop a benchmark suite for software visualization we propose a Call For Benchmarks much like a Call For Papers. We issue this call to all researchers active and/or interested in software visualization. The plan is to collect all proposed benchmarks, review each, and have a round of revisions/clarifications. The collection will assembled and made available to the research community on the web. This should coincide with a related conference or workshop and the benchmark could be presented in a working session or the like to motivate individual research groups to apply the benchmarks to their work.

The goal is to collect the results of using the benchmarks and present the findings in a paper, presentation, and/or web site.

We, the authors, invite benchmark proposals. The submitted benchmarks should include:

- Description of the proposed benchmark
- Software engineering task being addressed
- The data (sets) necessary (source code, models, data tables, etc.)
- What types of data analysis are necessary (if any) to apply the benchmark
- An evaluation method
- Types of user interaction required

E-mail your benchmark proposal to both jmaletic@cs.kent.edu and amarcus@cs.wayne.edu. For further information visit the web site www.sdml.info.

6 References

[1] Card, S. K., Mackinlay, J., and Shneiderman, B., Readings in Information Visualization Using Vision to Think, San Francisco, CA, Morgan Kaufmann, 1999.

[2] Chi, E. H., Barry, P., Riedl, J. T., and Konstan, J., "A spreadsheet approach to information visualization", in Proceedings of Information Visualization Symposium '97, 1997, pp. 17-24,116.

[3] CPPX, "CPPX - Open Source C++ Fact Extractor", Web page, http://swag.uwaterloo.ca/~cppx/, 2001.

[4] Eichelberger, H., "Nice Class Diagrams Admit Good Design", in Proceedings of ACM Symposium on Software Visualization (SoftVis'03), San Diego, CA, June 11-13 2003, pp. 159-168.

[5] Levoy, M., "Spreadsheet for images", Computer Graphics, vol. 28, 1994, pp. 139-146.

[6] Maletic, J. I., Marcus, A., and Collard, M. L., "A Task Oriented View of Software Visualization", in Proceedings of 1st IEEE Workshop of Visualizing Software for Understanding and Analysis (VISSOFT'02), Paris, France, June 26 2002, pp. 32-40.

[7] Marcus, A., Feng, L., and Maletic, J. I., "3D Representations for Software Visualization", in Proceedings of 1st ACM Symposium on Software Visualization (SoftVis'03), San Diego, CA, June 11-13 2003, pp. to appear.

[8] Myers, B. A., "Taxonomies of Visual Programming and Program Visualization", Journal of Visual Languages and Computing, vol. 1, no. 1, March 1990, pp. 97-123.

[9] Price, B. A., Baecker, R. M., and Small, I. S., "A Principled Taxonomy of Software Visualization", Journal of Visual Languages and Computing, vol. 4, no. 2, 1993, pp. 211-266.

[10] Price, B. A., Baecker, R. M., and Small, I. S., "An Introduction to Software Visualization", in Software Visualization, Stasko, J., Dominque, J., Brown, M., and Price, B., Eds., London, England MIT Press, 1998, pp. 4-26.

[11] Purchase, H. C., "Effective information visualisation: a study of graph drawing aesthetics and algorithms", Interacting with Computers, vol. 13, no. 2, December 2000 2000, pp. 147-162.

[12] Rao, R. and Card, S. K., "Exploring large tables with the table lens", in Proceedings of ACM Conference on Human Factors in Computing Systems (CHI'95), 1995, pp. 403-404.

[13] Reiss, S. P., "Visualizing Java in Action", in Proceedings of ACM Symboposium on Software Visualization (SoftVis'03), San Diego, CA, June 11-13 2003, pp. 57-66.

[14] Roman, G.-C. and Cox, K. C., "A Taxonomy of Program Visualization Systems", IEEE Computer, vol. 26, no. 12, December 1993, pp. 11-24.

[15] Sim, S. E., Easterbrook, S., and Holt, R. C., "Using Benchmarking to Advance Research: A Challenge to Software Engineering", in Proceedings of 25th International Conference on Software Engineering (ICSE'03), Portland OR, May 3-10 2003, pp. 74-83.

[16] Sim, S. E., Holt, R. C., and Easterbrook, S., "On Using a Benchmark to Evaluate C++ Extractors", in Proceedings of 10th International Workshop on Program Comprehension, Paris, France, 2002, pp. 114-123.

[17] Stasko, J. T. and Patterson, C., "Understanding and Characterizing Software Visualization Systems", in Proceedings of IEEE Workshop on Visual Languages, Seattle, WA, September 1992, pp. 3-10.

[18] Storey, M.-A. D., Best, C., and Michaud, J., "SHriMP Views: An Interactive Environment for Exploring Java Programs", in Proceedings of Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Ontario, Canada, May 12-13 2001, pp. 111-112.

[19] Tweedie, L., "Characterizing interactive externalizations", in Proceedings of Conference on Human Factors in Computing Systems (CHI '97), 1997, pp. 375-382.

Notes

Notes

Notes

Notes