Software Visualization

Stephan Diehl

Software Visualization

Visualizing the Structure, Behaviour, and Evolution of Software

With 124 Figures, including 75 in Colour, and 5 Tables

Springer

Author

Stephan Diehl

Universität Trier
Fachbereich Informatik
54286 Trier, Germany
diehl@uni-trier.de

Library of Congress Control Number: 2007923067

ACM Computing Classification (1998): D.2, I.3.8, J.6

ISBN 978-3-540-46504-1 Springer Berlin Heidelberg New York

Typeset by the author
Production: LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkelLopka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3100/YL - 5 4 3 2 1 0

To Christine, Luca, and Jean-Luc

Preface

Software systems are designed, implemented, tested, debugged, analyzed, and maintained by many changing developers. All these tasks can be facilitated by visualization.

In this book we give an overview of the various areas of software visualization, the art and science of generating visual representations of various aspects of software and its development process.

In contrast to visual programming and diagramming for software design, software visualization is not so much concerned with the construction, but with the analysis of programs and their development process.

So far, there exist only anthologies and proceedings about software visualization. This book is the first textbook on software visualization. Although written mostly for graduate students, the book is also a valuable resource for researchers as it provides a broad and systematic overview of the area with many pointers to literature and systems for further study.

As the field of software visualization is growing fast, the book is not meant to be comprehensive, but we have attempted to select seminal work as well as promising new approaches to illustrate some emerging principles in the field. Each chapter is followed by a list of exercises including both pen&paper exercises, as well as programming tasks.

This book is aimed at graduate students and researchers who are new to the field of software visualization. The book is meant to be read from end to end, though some readers may want to skip some of the more formal sections. Ideally, after reading the book, the reader will be able to

- identify recurring concepts in various areas of software visualization;
- understand the purpose of various visualization techniques;
- appreciate the use of visualization in software engineering.

We assume that the reader will have some programming experience, preferably in Java, and some basic knowledge of software engineering terminology. No prior knowledge of software visualization is required.

Additional material related to this book, including examples and program code for exercises, can be found at the following Web address:

`http://www.eposoft.org/svbook`

March 2007 *Stephan Diehl*

Contents

1 Introduction

Nobody has ever directly seen an atom, but most of us will think of an atom as a core surrounded by small spheres or orbital clouds. The important role that *visualization* plays in human reasoning in general and in scientific progress in particular has been emphasized by philosophers throughout the centuries.

> . . . thought is impossible without an image.
>
> (Aristotle, 350 BC)

> Imagination or visualization, and in particular the use of diagrams, has a crucial part to play in scientific research.
>
> (René Descartes, 1637)

> The understanding can intuit nothing, the senses can think nothing. Only through their union can knowledge arise.
>
> (Immanuel Kant, 1781)

Today computers have become an important tool for creating visualizations and helping the user to better understand complex phenomena. As a consequence visualization has become a discipline of computer science. So in the rest of this book, we shall use the term for this discipline, rather than for the cognitive activity of forming mental images. Gershon [Ger94] defines visualization as follows:

> Visualization is more than a method of computing. Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis.

Visualization plays a major role in the use of computers to support human reasoning, a field that was named “intelligence amplification”, or IA for short [Jr.96], in contrast to “artificial intelligence”, or AI for short, where the goal is that the computer itself becomes intelligent.

Visualization is heavily used in mechanical engineering, chemistry, physics, and medicine. Computer scientists have developed sophisticated systems to produce visualizations for these disciplines. Astonishingly enough, computer scientists have only made little use of visualization as a tool for designing, implementing and maintaining software (see Fig. 1.1). Even worse, many consider themselves as theoreticians and disregard visualization – an etymologically wrong dichotomy.[1] Programmers tend to adapt to the level of representation provided by the computer, instead of adapting the computers representations to their perceptive abilities.

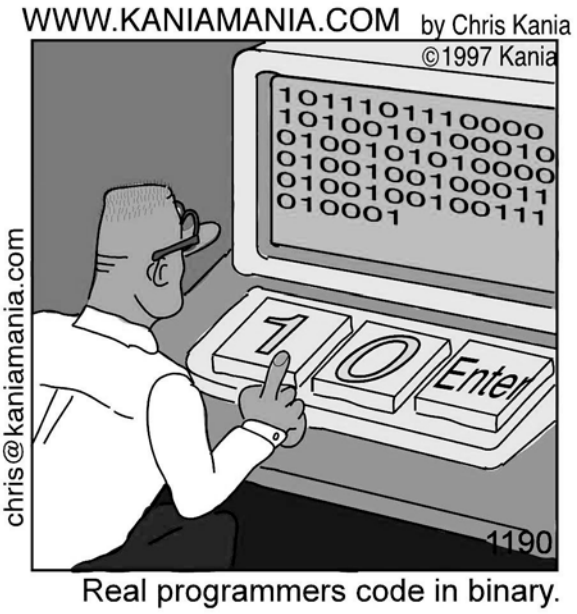

WWW.KANIAMANIA.COM by Chris Kania
©1997 Kania
chris@kaniamania.com
1190
1 0 Enter
Real programmers code in binary.

Fig. 1.1. No visualization required

Despite all of its formal and cryptic notation, the terminology of computer science is rich in *metaphors*. The goal of such metaphors is to evoke mental images to better memorize concepts and to exploit analogies to better understand structures or functions. Computer scientists use the terms "automata" and "machines" for mathematical models of computation. The terms "tapes", "trees", "leaves", "queues", "files", "folders", and "archives" are used to denote data structures. For example, a Turing machine is a mathematical model composed of sets, functions, and/or relations. The *machine* analogy lets us

[1] The word "theory" comes form the Greek word *theorein*, which means "to view". The early Pythagoreans supported their theorems not by proofs but by contemplation, i.e. by drawing sketches in sand.

transport aspects from the physical world to the mathematical world and thus helps to better understand the mathematical model. We might even think of gear wheels and how one wheel drives the others, once we start to turn one of them. It has often been noted that software is inherently intangible and invisible. The goal of software visualization is not to produce neat computer images, but computer images which evoke *mental images* for comprehending software better. Finding new metaphors thus will not just produce better visualizations, but it will also improve the way we talk about systems.

1.1 What Is Software Visualization?

So far, we have talked about visualization and its importance in human reasoning, in particular for science. Today there are two major disciplines of visualization: *scientific visualization* processes physical data, whereas *information visualization* processes abstract data.[2] As algorithms are a kind of information, we consider software visualization part of information visualization. In the following chapters we shall look at the use of visualization in the context of software development to foster understanding and insight. Many authors define *software visualization* as *the visualization of algorithms and programs* (a narrow definition).

This definition excludes a lot of uses of visualization techniques in computer science and has also hindered synergies in the past. In this text we define software visualization as *the visualization of artifacts related to software and its development process* (a wide definition). In addition to the program code, these artifacts include requirements and design documentation, changes to the source code, and bug reports, for example. In fact, researchers in software visualization develop and investigate methods and uses of computer graphical representations of various aspects of software, for example its static structure, its concrete and abstract execution, and its evolution. In short, they are concerned with *visualizing the structure, behavior, and evolution of software.*

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

Behavior refers to the execution of the program with real and abstract[3] data. The execution can be seen as a sequence of program states, where a program state contains both the current code and the data of the program. Depending on the programming language, the execution can be viewed on a higher level of abstraction as functions calling other functions, or objects communicating with other objects.

[2] There have been several attempts in the literature [Chi00] to make the distinction more clear, but there always remains an overlap of the two disciplines.

[3] see Sect. 4.4.7.

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

Thus, software visualization is the art and science of generating visual representations of various aspects of software and its development process. As Tilley and Smith put it, "Program understanding is the (ill-defined) deductive process of acquiring knowledge about a software artifact through analysis, abstraction, and generalization." [TS96]. The goal of software visualization is to help to comprehend software systems and to improve the productivity of the software development process.

1.2 Organization of This Book

In the following chapters we shall learn what kinds of information about software exist, how they are computed, what visual representations are appropriate for them, and how they are computed. Our selection of topics, systems and approaches is in no way complete, but we have tried to select seminal work, as well as newer approaches that we have found most promising.

As shown in Fig. 1.2, the book is organized around the three aspects of software discussed in Chaps. 3, 4 and 5: structure, behavior and evolution. Each of these three chapters discusses the visualization of one of these aspects at various levels of abstraction, from the program code level to the architecture level.

In Chap. 1 we discuss two definitions of software visualization and look at some initial examples. We also introduce the visualization pipeline, which consists of the various phases of the visualization process.

Chapter 2 presents background information about visual perception and cognition, and also introduces general information visualization techniques for textual, hierarchical and graph-based information.

In Chap. 3 we look at the visualization of static program properties, i.e. what textual and graphical means exist to display programs, and the results of program analyses. Next, we discuss the visualization of software architectures. We briefly look at the unified modeling language (UML) and alternative representations, followed by a look at software metrics. Then we take a closer look at tools that allow one to extract, draw and explore the architecture of a system.

In Chap. 4 we first discuss the general aspects of dynamic program visualization, then we briefly look at the dynamic visualization of software architectures. Then, algorithm animation is covered in more detail because most of the work on dynamic program visualization is related to algorithm animation. Finally, various visual debugging and testing techniques are presented, which can help the programmer find errors in programs. These techniques can be divided roughly into those showing the program memory and those showing the program code.

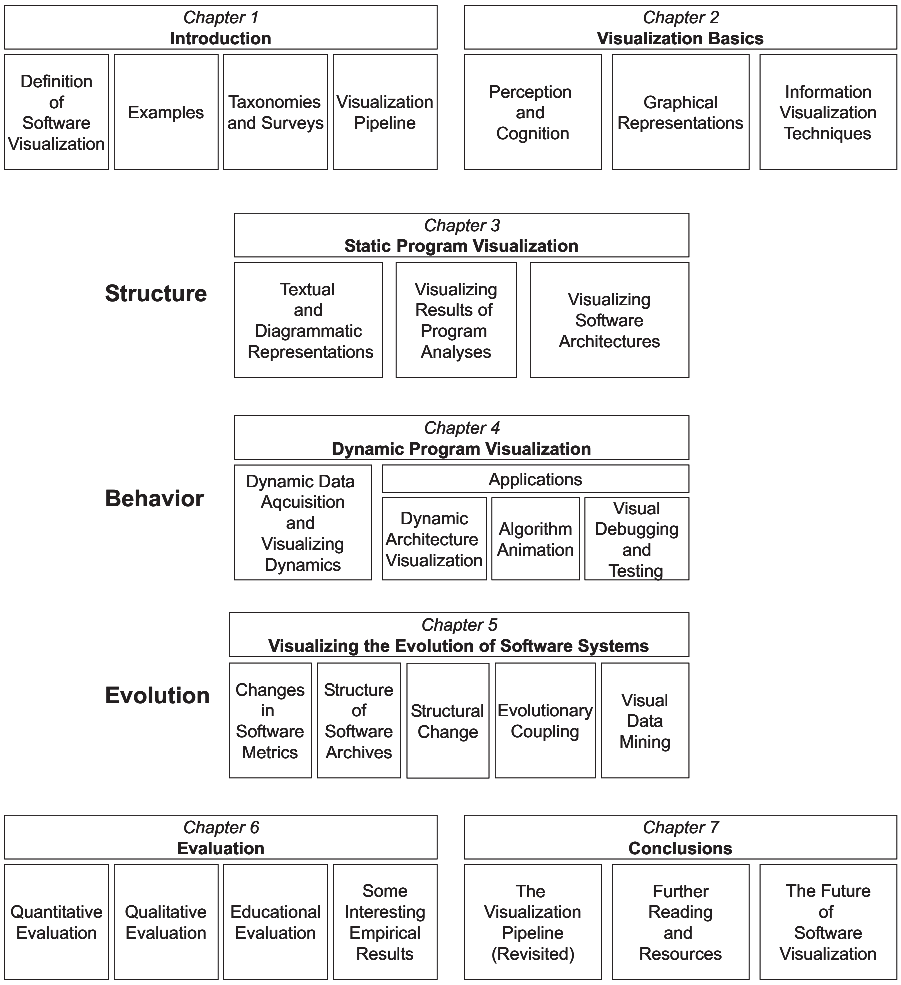

Chapter 1
Introduction
Definition of Software Visualization
Examples
Taxonomies and Surveys
Visualization Pipeline
Chapter 2
Visualization Basics
Perception and Cognition
Graphical Representations
Information Visualization Techniques
Structure
Chapter 3
Static Program Visualization
Textual and Diagrammatic Representations
Visualizing Results of Program Analyses
Visualizing Software Architectures
Behavior
Chapter 4
Dynamic Program Visualization
Dynamic Data Aqcuisition and Visualizing Dynamics
Applications
Dynamic Architecture Visualization
Algorithm Animation
Visual Debugging and Testing
Evolution
Chapter 5
Visualizing the Evolution of Software Systems
Changes in Software Metrics
Structure of Software Archives
Structural Change
Evolutionary Coupling
Visual Data Mining
Chapter 6
Evaluation
Quantitative Evaluation
Qualitative Evaluation
Educational Evaluation
Some Interesting Empirical Results
Chapter 7
Conclusions
The Visualization Pipeline (Revisited)
Further Reading and Resources
The Future of Software Visualization

Fig. 1.2. Organization of this book

Chapter 5 covers the visualization of software evolution. The techniques discussed allow a person to graphically represent the development history of a software system in order to explore changes of software metrics and structure, as well as the changes of dependencies of software artifacts over time.

Chapter 6 provides background information on how to evaluate software visualization tools, then briefly presents some interesting results of empirical studies related to software visualization. We look at quantitative and qualitative evaluation methods, and discuss some empirical results related to software visualization. In particular, we emphasize qualitative methods that can be applied during the design of a system.

Chapter 7 summarizes the various approaches discussed in the book by looking at the visualization pipeline again.

1.3 Software Visualization and Visual Programming

In classical programming languages, programs are represented as text, and the meaning results from the linear order of lexical elements. *Visual programs* consist of graphical and often also textual elements. The meaning of the programs depends on the spatial placement of and the connection between these elements.

In accordance with various programming paradigms, there are control-flow, data-flow, functional, object-oriented, rule-based, form-based, and hybrid (multiparadigm) visual programming languages. Many visual programming languages just allow the user more or less to visually build the abstract syntax tree of a textual program.[4]

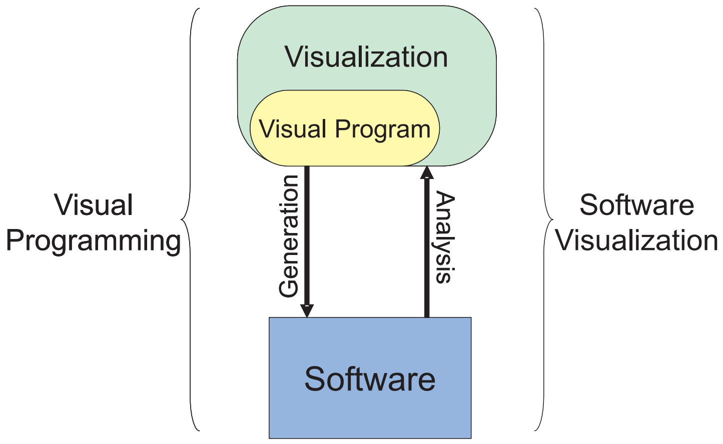


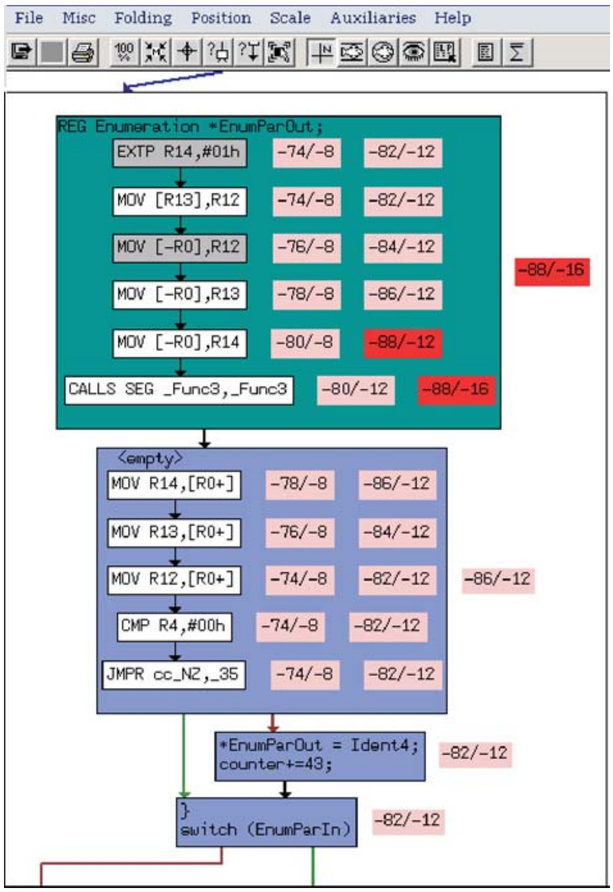
Fig. 1.3. Visual programming versus software visualization

As shown in Fig. 1.3, visual programming and software visualization complement each other. Software visualization generates visualizations from specifications of software systems, while visual programming generates software systems from visual specifications. Combining the two approaches allows round-trip visualization, for example by producing a visual presentation from the source code of a system, changing this visual presentation, and generating a new system.

[4] There are also integrated development environments for textual programming languages that have been called “visual” for marketing reasons.

1.4 Examples of Software Visualization Tools

To give the reader a first impression of how visualization techniques can help the software engineer, we shall briefly look at three examples of software visualization tools drawn from different areas: program development, education, and software evolution.



```
REG Enumeration *EnumParOut;
EXTP R14,#01h               -74/-8   -82/-12
MOV [R13],R12               -74/-8   -82/-12
MOV [-R0],R12               -76/-8   -84/-12     -88/-16
MOV [-R0],R13               -78/-8   -86/-12
MOV [-R0],R14               -80/-8   -88/-12
CALLS SEG _Func3,_Func3     -80/-12  -88/-16

<empty>
MOV R14,[R0+]               -78/-8   -86/-12
MOV R13,[R0+]               -76/-8   -84/-12
MOV R12,[R0+]               -74/-8   -82/-12     -86/-12
CMP R4,#00h                 -74/-8   -82/-12
JMPR cc_NZ,_35              -74/-8   -82/-12

*EnumParOut = Ident4;       -82/-12
counter+=43;

}                           -82/-12
switch (EnumParIn)
```

Fig. 1.4. Stack usage

1.4.1 StackAnalyzer: Static Program Visualization

The first example shows how visualization can be used to support programmers. The program analysis tool *StackAnalyzer* produces visualizations of control-flow graphs of embedded applications [EB02, Ang]. In these graphs, the results of a static program analysis are shown (see Fig. 1.4). For each instruction and each function, the analysis computes the stack usage – both of the user stack and the system stack. This information can for example be used to prevent runtime errors due to stack overflow. In Fig. 1.4, each instruction and each code block is annotated with stack height differences. The annotation is colored red if the difference exceeds a certain limit.

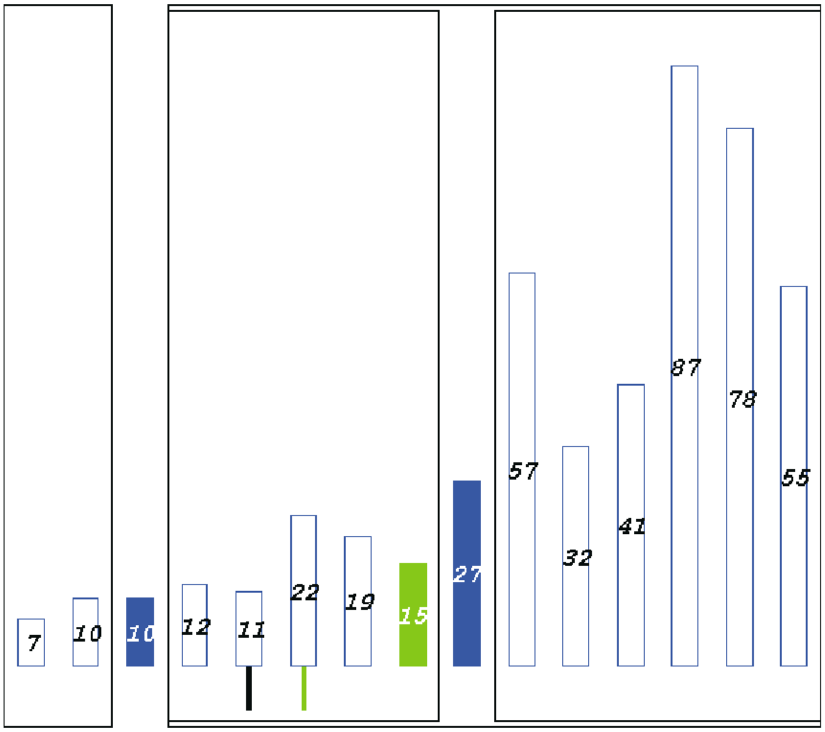
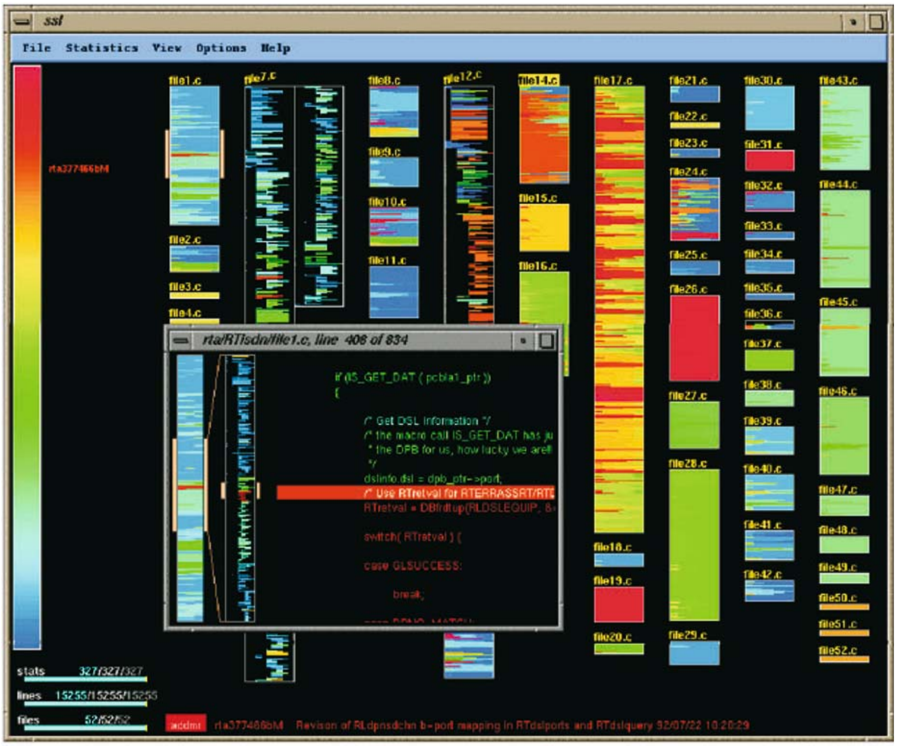


Fig. 1.5. X-Tango animation of the quicksort algorithm

1.4.2 X-Tango: Algorithm Animation

Dynamic visualizations are widely used as learning aids in computer science. In Fig. 1.5, a snapshot of an animation of the quicksort algorithm is shown. The animation was produced with the X-Tango algorithm animation tool [Sta90a]. The elements to be sorted are shown as vertical bars, nested recursive calls are indicated by the boxes around some of the bars, and the current pivot element, i.e. the element where the list is split, is colored green (black in the grayscale image). The elements to the left of the pivot element are smaller than or equal to the pivot element, those to the right are greater than the pivot element.



```
if (IS_GET_DAT ( pcbla1_ptr ))
{

        /* Get DSL information */
        /* the macro call IS_GET_DAT has ju
         * the DPB for us, how lucky we are!!
         */
        dslinfo.dsl = dpb_ptr->port;
        /* Use RTretval for RTERRASSRT/RTE
        RTretval = DBfrdtup(RLDSLEQUIP, &

        switch( RTretval ) {

        case GLSUCCESS:

                break;
```

rta377466bM Revison of RLdpnsdchn b-port mapping in RTdslports and RTdslquery 92/07/22 10:20:29

Fig. 1.6. Visualizing the age of program code changes (©1996 IEEE)

1.4.3 SeeSoft: Software Evolution

Project managers of large software projects need tools to get a quick overview of the state of the whole system, and to find trends in the evolution of the system. SeeSoft was developed at AT&T Bell Laboratories to visualize changes and metrics related to evolving large (several million lines of code), complex software systems [ESJ92, BE95, BE96]. As shown in Fig. 1.6, files are represented by rectangles. Within each rectangle, colored pixels or lines represent lines of the source code. In this example, the color indicates the age of the last modification. Blue (cold) is used for lines which have not been changed for a long time, whereas red (hot) is used for recently changed lines.

1.5 Taxonomies and Surveys

Several researchers have proposed taxonomies to classify software visualization research and tools. In this section we briefly review some of these taxonomies, because they provide various ways to look at and structure software visualization research. Then we discuss the results of three surveys: one about the amount of published research work and two about the importance of visualization for software engineers.

Myers introduced a taxonomy for program visualization [Mye90] which identifies six regions arranged in a 2×3 matrix as shown in Fig. 1.7. He distinguishes data, code, and algorithm visualization, where algorithm visualizations represent algorithms at a higher level of abstraction than program code.

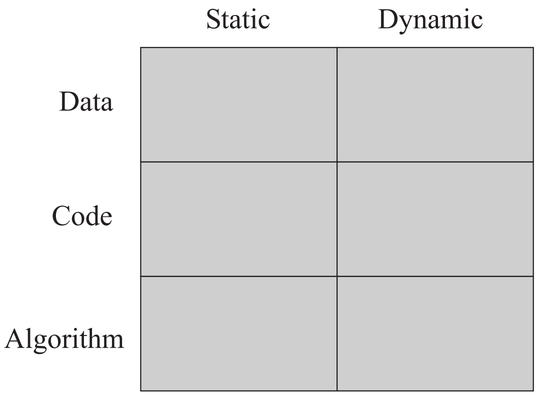


Fig. 1.7. Taxonomy introduced by Myers

Three years later, Price et al. suggested a more hierarchical taxonomy of software visualization [PBS93]. These authors distinguish program visualization, which consists of code and data visualization, from the more abstract algorithm animation. In addition, they introduced a number of aspects which could be used to classify software visualization tools:

Scope: What is the range of programs used as inputs for the visualization?
Content: What kind of information about the software is visualized?
Form: What are characteristics of the output of the system (e.g. the medium)?
Method: How is the visualization specified?
Interaction: How does the user control the system?
Effectiveness: How well does the system convey information to the user?

At about the same time, Cox and Roman used a very similar set of aspects to classify some existing software visualization tools [CR93]: scope (code, data state, control state, and behavior) abstraction, specification method, interface, and presentation.

In an attempt to identify open research questions in software visualization, the current author did a literature survey [Die02a] and classified research papers into a 4×4 matrix, shown in Fig. 1.8. The two dimensions of the matrix are the classical abstraction layers of software systems (hardware, virtual/abstract machine, program, and system) and the static and dynamic phenomena of these layers. The map is incomplete in the sense that one could add additional layers (e.g. operating system) or structures (e.g. project structure). In the matrix, shades of gray indicate how much published research exists in the corresponding areas of software visualization. The survey gives us a rough orientation in relation to the research activity in these areas.

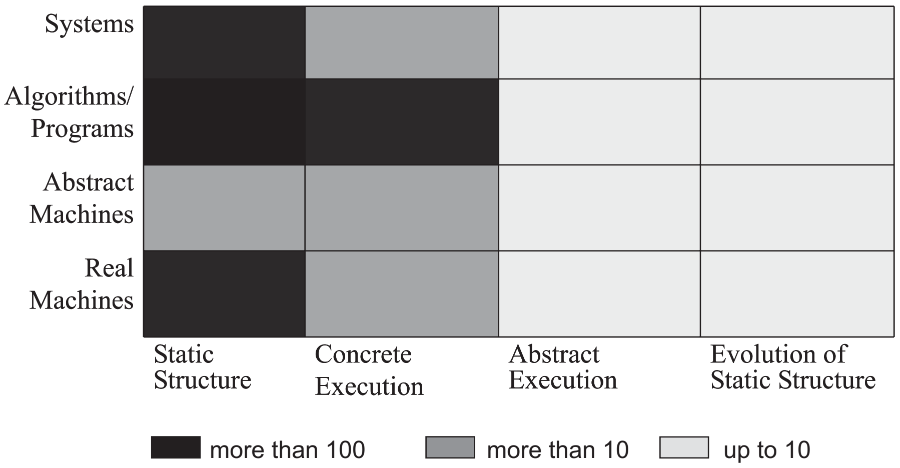


Fig. 1.8. Literature survey

In a recent survey [Kos02] based on questionnaires filled in by 111 researchers in software maintenance, reengineering, and reverse engineering, Rainer Koschke reported that 40% found software visualization absolutely necessary for their work and that another 42% found it important but not critical. Thus, according to this survey, for 82% of the participating software engineers, software visualization is important.

In another survey [BK01], with 107 participants mostly from industry, Bassil and Keller found the following reasons why practitioners apply software visualization. In order of decreasing importance the benefits of software visualization tools were:

- savings in time and money;
- better comprehension of software;
- increase in productivity and quality;
- management of complexity;
- to find errors.

When the participants were asked about the problems of current software visualization tools that need to be solved, the integration of software visualization tools into other (third-party) tools, and improved import and export of data and visualizations ranked highest.

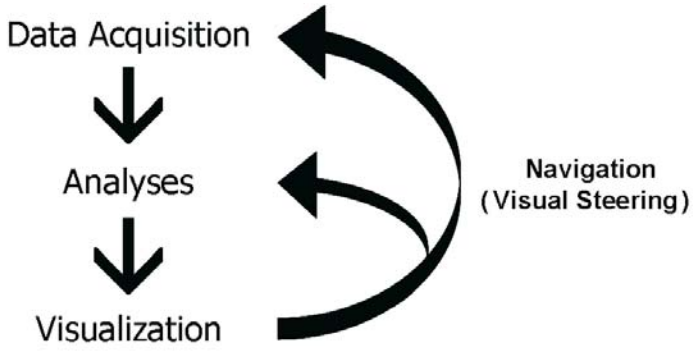


Fig. 1.9. The visualization pipeline

1.6 The Visualization Pipeline

The creation of computer images is just the last step in the *visualization pipeline*. As shown in Fig. 1.9, information produced by one stage is used as input by the next stage.

Data acquisition: There are various sources of information about a software system, including its source code, its design, user documentation, state changes during its execution, test results, and mailing lists. The methods used to extract and gather relevant data from these sources are as different as the sources are.

Analysis: Typically, the amount of information is too much to be immediately presented to the user. Various kinds of analysis, such as filtering, static program analysis, or statistical methods, can be used to reduce the amount of data and to focus on the important parts.

Visualization: The resulting data is mapped onto a visual model, i.e. transformed into geometrical and graphical information, and then rendered onto the screen or some other kind of medium as a single image or a series of images.

In interactive visualizations, the user can control the previous steps of the pipeline on the basis of the graphical output produced earlier. This method of interaction is sometimes called computational or visual steering [JPH$^+$99, MvWvL99].

In this book, we describe software visualization tools by the tasks they support and the techniques used at the various stages of the visualization pipeline.

Exercises

Exercise 1: Give examples of visual metaphors used in computer science terminology that have not yet been mentioned in the book.

Exercise 2: What visualization tools do you know that are used in the engineering, natural, medical or social sciences? What visual metaphors are used? Can you imagine ways to apply some of these tools in software engineering?

Exercise 3: Consider the visualization tools of the previous exercise. Do they also use a visualization pipeline similar to the one that we discussed in Sect. 1.6? If so, can you describe how they implement the three stages?

2 Visualization Basics

The goal of visualization is to convey information through the human visual system into the human brain by drawing images on the computer screen. Hence, visualization involves humans and machines. Visualizations that ignore technical or cognitive idiosyncrasies are doomed to failure.

In this chapter we briefly present some basic physiological, psychological, and technical knowledge about visualization.

More information on visualization in general and on information visualization in particular can be found in various related books [CMS99, War00, Spe01, Che04].

2.1 Perception and Cognition

Nihil est in intellectu quod non erat in sensu.[1]

[Aristotle, 384–322 B.C.]

Perception is the processing of sensory information and thus part of human cognition, which also includes awareness, reasoning, and learning. 75% of all information from the real world is visually perceived; only 13% is perceived through the auditory sense and the remaining 12% through other senses.

According to Nobel Prize winner Roger W. Sperry, the human brain consists of two processing units, each located in one of its hemispheres [Spe68]. While the "left brain" does the verbal, analytical, rational, temporal, and sequential reasoning, the reasoning of the "right brain" is nonverbal, synthetic, intuitive, nontemporal, and parallel. Visualization helps to exploit the mind's capacity by integrating both hemispheres. Using both verbal and nonverbal representations for the same kind of information is often referred to as the dual-coding theory [Pai90].

[1] "Nothing exists in the mind, that has not been before in the senses."

2.1.1 Visual Memory

In particular, *visual memory* turns out to be astonishingly good. Shepard [She67] showed 600 very different kinds of pictures and sentences to test persons and later asked them to recognize those pictures and sentences which they had seen before. For the pictures, the error rate was only 1.5%, compared with 11.8% for sentences. Standing [Sta73] did a similar experiment with 10 000 pictures and found an error rate of 17%. As a matter of fact, many memorization techniques, also called mnemonics, exploit visual memory by internal visualization.

While our visual memory can store a vast number of pictures, it seems to store only the gist of them and thus we seem to be blind to small changes in an image [Wol98, ROC97].

2.1.2 The Human Eye

Figure 2.1 shows a schematic drawing of the human eye. Light falls through the lens onto the retina. On the surface of the retina there are two kinds of receptors: about 6 million cones for color vision and 100 million rods for black-and-white vision. The receptors are not evenly distributed. At the intersection of the optic axis and the retina, the density of cones is very high and there are only few rods. Consequently, this is exactly the point with the best vision (the fovea centralis) with respect to both resolution and color vision. In the periphery of the fovea, the density of cones is much smaller than that of rods and thus this area allows only black-and-white vision. In general, the density of receptors decreases with distance from the optic axis.

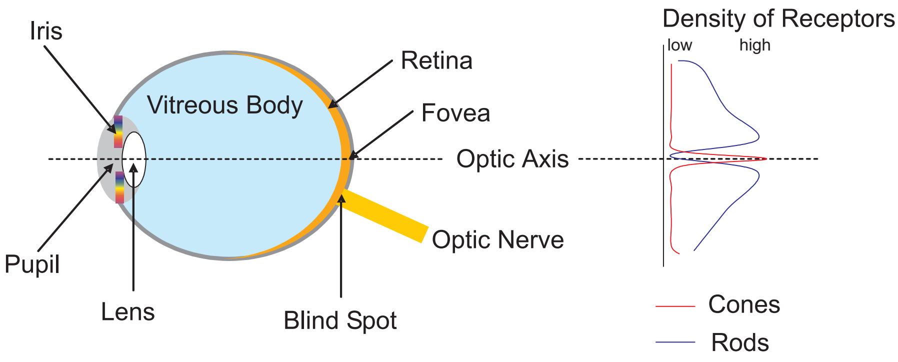


Fig. 2.1. The human eye and the distribution of receptors

Many people are not aware of the blind spot in their eyes. This spot is caused by the optic nerve connecting to the eye. With the diagram shown in Fig. 2.2, you can experience your own blind spot.

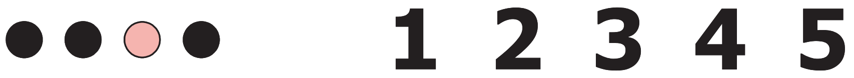


Fig. 2.2. Blind spot: close your right eye and focus with your left eye on each of the numbers. Depending on the distance from your eyes, one of the dots will disappear

2.1.3 Light, Color, and Color Perception

Light is a kind of electromagnetic radiation. More precisely, radiation within a narrow frequency band. The speed of light c (about 300 000 km/s) is a fundamental physical constant. Because of the physical law $c = \lambda f$, light can be characterized unambiguously by its wavelength λ or by its frequency f. The wavelength is usually measured in nanometers: $1\,\text{nm} = 10^{-9}\,\text{m}$. Visible light ranges from 700 nm (red) to 400 nm (violet) and is actually a mixture or, more precisely, a superposition of light waves of different wavelengths and intensities.

Color is the human perception of light. The hue of a color is related to its dominant wavelength, whereas brightness is related to the intensity or amplitude of the wave. Owing to the physiology of the retina, different mixtures of light can lead to the same color perception.

There are three kinds of cones in the human eye, which are sensitive to magenta, green, and yellow-to-red. Each cone thus reacts to an interval of different wavelengths, and these intervals overlap. For example, blue light stimulates the green receptors most, but the other receptors will also react. The human brain combines the signals from all these receptors and produces an impression of color. Different combinations of stimulation of the three kinds of receptors form the human color space. Thus the color we perceive is actually a product of our brain and we have to distinguish perceived colors from physical colors, i.e. those that we attribute to certain wavelengths. While color impressions, such as blue or green correspond to a continuous range of wavelengths, color impressions such as brown, grey, or white are mixtures of different wavelengths. For example, the color impression white results if the three kinds of cones are stimulated in equal measure. Even worse, color perception may differ between male and females [VT04], between cultures, and even in the same individual depending on the adaption of the eye to the surrounding illumination (this is also known as the Purkinje effect) or on what he or she has seen before, because of the inertia of the visual system.

2.1.4 Pattern Perception

Pattern perception is the task of deciding whether visual elements such as lines and areas belong to the same object. On the basis of experimental results, Gestalt theory [Wer23, PR94] suggests that the human brain uses the

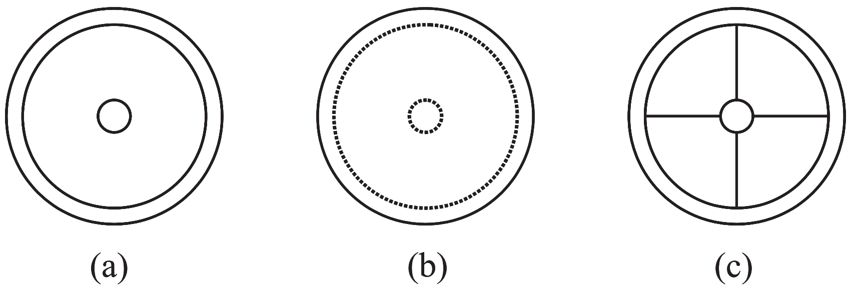


Fig. 2.3. Pattern perception: which circles belong to the same object?

following criteria in decreasing importance to perceive patterns: connectedness, proximity, similarity of color or shape, continuity of curves, symmetry, closure of areas, relative size, orientation, background, and transparency. For example, in Fig. 2.3a the two outer circles seem to form an outer ring because of proximity, in Fig. 2.3b the inner circles seem to form a disk because the have a similar line style, and in Fig. 2.3c they form an inner disk because of connectedness. When we look at these figures, our brain may switch from one interpretation to another.

2.1.5 Preattentive Perception

Features of visual objects are preattentive [HBE96] if they are perceived within 200 ms, i.e. the time interval it takes before the eye reacts and moves. Such features include the orientation, length, and width of lines, the size of an object, curvature, number, intersection, color, luminance, flicker, direction of movement, gloss, spatial depth, and light direction. In each of the three groups of objects in Fig. 2.4, one of the elements is immediately spotted because of its color, shape, or size.

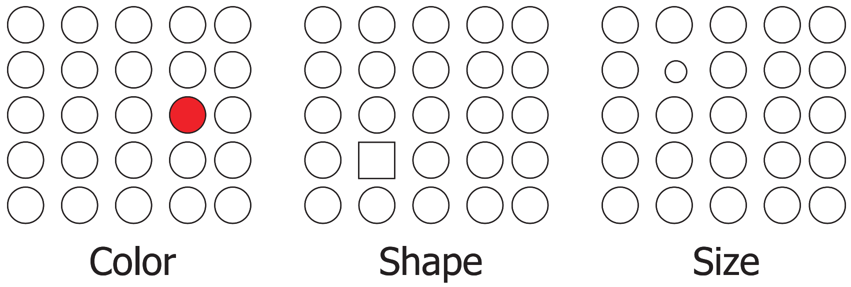


Fig. 2.4. Preattentive perception

2.1.6 Motion Perception

In the evolution of humans, motion perception has been an important ability both for protection from predators and for hunting. Motion perception is the task of deciding whether visual elements such as lines and areas perform the same movement in subsequent pictures. It is based on pattern recognition, but in addition it requires one to detect changes of features such as form, color, or position. To detect these changes, a correspondence between elements in subsequent pictures has to be established. The illusion of backward-spinning wagon wheels in Western movies demonstrates that it is possible that the mind can match the wrong elements – here, the spokes on a wheel, because of their proximity, as shown in Fig. 2.5.

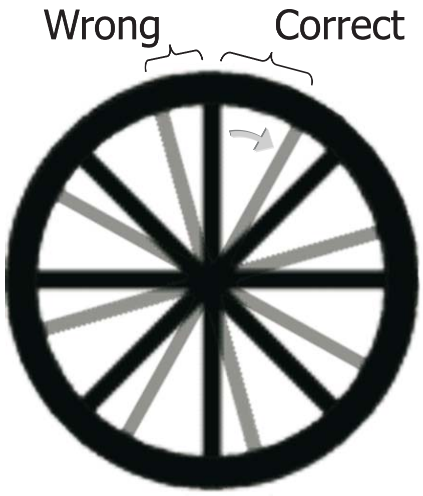


Fig. 2.5. Correspondence of spokes on a rotating wagon wheel

If an object starts moving within a certain time interval after another object has stopped moving, the latter is perceived to cause the movement of the former object. If the time interval becomes too large, typically longer than 160 ms, then the mind does not establish this causality. Interestingly, the two objects do not have to be to next to each other. Instead of drawing a line, movement can be used to link two objects which are placed in different parts of the screen. An undirected relation is indicated by both objects performing the same movement, and a directed link by two sequential movements as discussed above.

2.1.7 Implications for the Design of Visualizations

As discussed above, the human eye allows color perception only in the center, no color is perceived in the periphery. The effectiveness of recognizing shapes decreases with the distance from the center owing to the decreasing density of receptors. Movements are perceived effectively in the periphery and usually people react by orienting their focus to the moving objects.

Thus, in the design of a visualization, color should be used for detail information, shape can be used for detail information about a single object but also be used to show the relation between different objects, and motion can be used as a stimulus in the periphery to attract attention and to establish links between remote objects.

When choosing color palettes, color-deficient vision should be taken into account. While complete color blindness, also called monochromacy, is very rare (only 0.003% of the whole population), many people have problems in seeing certain colors. Surprisingly, 7% of the male population but only 0.5% of the female population suffer from a red-green deficiency. Color palettes with yellow–blue variations are more acceptable for people with color-deficient vision. The reader can easily find such palettes on the web, for example at `http://www.btplc.com/age_disability/ClearerInformation/Colours/index.htm`.

In visualization, color is often used to represent values from a discrete or continuous numerical interval. In this case each value is mapped to a color – the value is *color coded*. The following color scales are widely used, because they are very intuitive:

Cold-to-hot color scale: blue for cold, green for warm, red for hot.
Traffic light: green for safe, yellow for possibly dangerous, and red for dangerous.
Terrain color scale: green for vegetation, brown for no vegetation, and white for snow.

Linear optimal color scales [LH92] are used when it is important that the perceived difference between colors is proportional to the distance between the values they encode. Whenever possible, software visualization tools should allow the user to choose among different predefined color scales, as shown in Fig. 2.6.

Eye-tracking experiments have shown that distribution of the spontaneous attention of a person looking at a screen is not evenly distributed. If we divide the screen into two upper and two lower areas, then the left upper part receives 40%, the right upper part 20%, the left lower part 25%, and the right lower part only 15% of the user's attention. Accordingly, important information should be placed in the areas of high attention, and less important information in those of low attention. Like many psychological results, this one depends heavily depends on culture. For example, in Arabic-speaking countries one reads from right to left, and so the upper right part is most

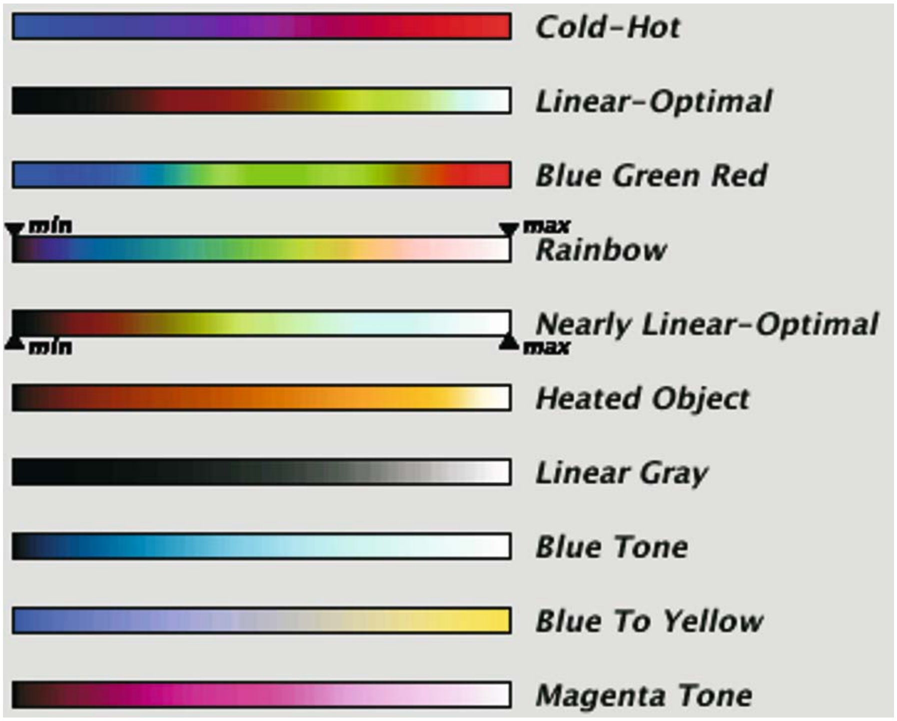


Fig. 2.6. Color scales in the EpoSee tool

prominent. It is quite illuminating to look at Web pages that allow one to switch between Arabian and English, Hebrew and English, or Chinese and English, for example, because not only the font and the orientation of the writing but also other properties of the layout change.

2.2 Graphical Representation

To visually encode information, one can use text as well as two- or three-dimensional computer graphical representations. As software visualization mostly deals with software artifacts and their interrelations, graph-based representations play a major role.

2.2.1 Graphical Primitives and Properties

Visualizations are built from points, lines, areas, and volumes. These primitives have various properties: size, length, width, height, volume, position, orientation, angle, slope, color, grayscale, texture, and shape.

In addition, graphical objects can have dynamics, i.e. one or more properties change over time. Dynamics range from simple blinking or translation of objects to complex animations.

All these properties can be used to encode information. Consider, for example, a traffic light. For each of its three lights, its color (green, yellow, or red), its shape (e.g. round, walking man, or arrow), its relative position (e.g. the lowest light is the green one) and its dynamics (blinking) are meaningful. For color-blind people, in the case of the traffic light, it is important that position and color convey redundant information.

2.2.2 Text

Text consists of words. Words are sequences of characters from an alphabet, and words usually have a meaning. Textual information can be augmented by visual cues such as underlining, color, font, face, and orientation to show additional information, highlight important parts, or make the structure more explicit. The art of presenting text in a readable and visually pleasing way is known as typography [Bri02].

2.2.3 Diagrams

A diagram is a graphical representation where the geometrical relations between its parts illustrate relations between the objects represented by those parts. These geometrical relations include neighborhood, linkage, containment, and overlapping. For example, graphs use linkage, while Venn diagrams use containment and overlapping to visualize relations. In the following section, we take a closer look at graphs.

Larkin and Simon have characterized diagrams as representations where information is indexed by its two-dimensional location [LS87], whereas textual information is indexed by its sequential position. If done right, diagrams group relevant information together to make searching more efficient, and use visual cues to make information more explicit.

2.2.4 3D Graphics and Rendering

Many computer graphics systems, in particular those which have to render graphics in real time, use the following architecture. Objects are given by mathematical models or directly by a mesh of polygons. A polygon mesh defines the hull of an object as a set of faces. These faces can be subdivided into triangles. These triangles are determined by their corner points in the local coordinate system of the object. The triangles pass through a sequence of processing units in the rendering software. The output of one unit is used as input by the next one. This sequence of processing units is called a graphics or rendering pipeline (see Fig. 2.7). Modern 3D graphics cards provide partial or even complete hardware implementations of some of the processing units. The units are as follows:

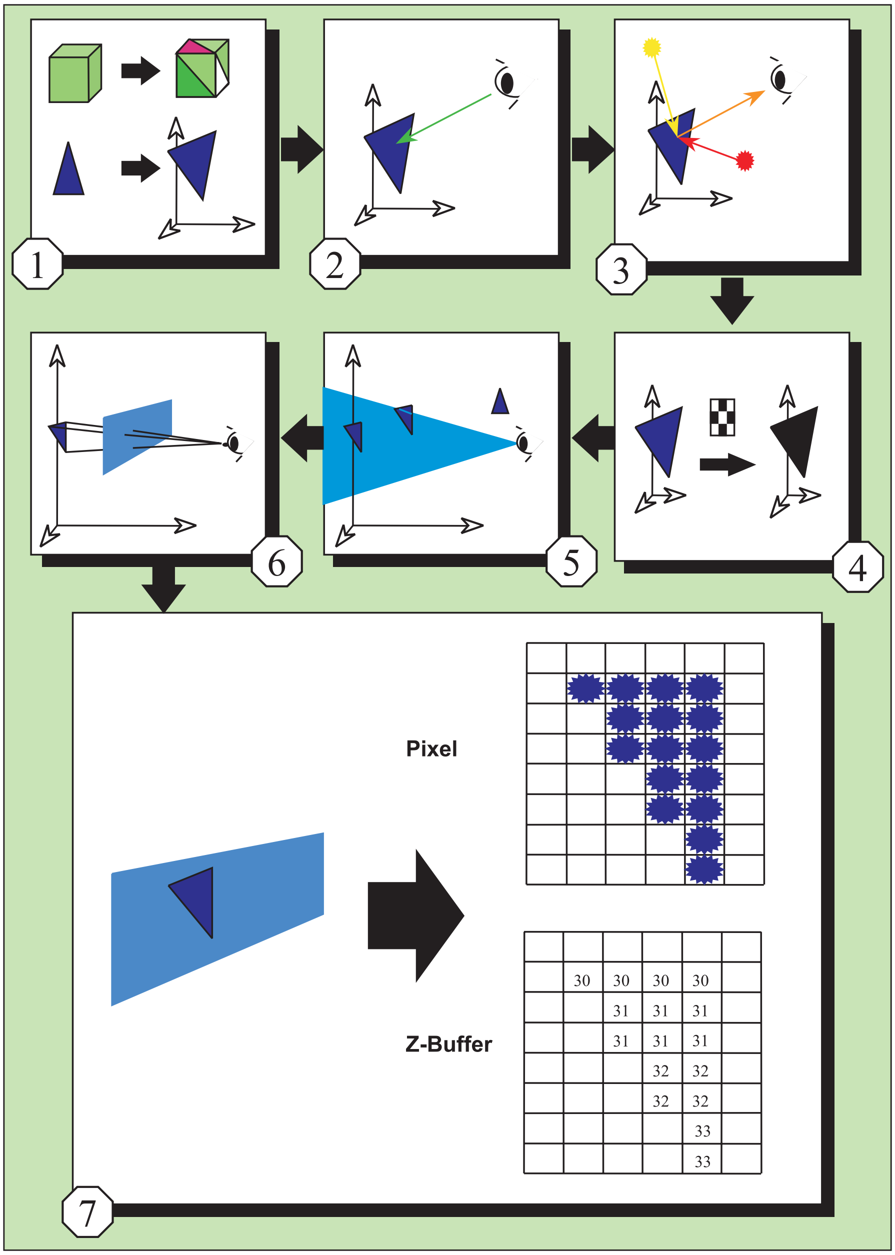


Fig. 2.7. 3D computer graphics pipeline

1. *Transformation:* The triangle is placed in the global coordinate system, i.e. rotations, translations, and scaling are applied.
2. *Backface Culling:* Triangles have front and back faces. In certain situations back faces do not have to be rendered. For such triangles, the unit checks whether the viewer sees the front or back face. In the latter case the triangle is not processed further. As an example, consider a cardboard box. If the box is closed, then one cannot see the inner faces of the sides from outside, as they are always hidden by other sides.
3. *Lighting:* At this stage, the color of the triangle is determined. Its own color, brightness, transparency, and reflection properties, as well as the color of the rays of light which fall on it, have to be combined. The resulting color is computed using lighting equations, which only weakly relate to optical laws as developed in physics, but yield realistic color effects with less computational effort.
4. *Texture Mapping:* Two-dimensional images are mapped onto the flat surface of the object. By a clever choice of such images, one can achieve the impression of a three-dimensional surface. At this stage, the corner points of the triangle are mapped to coordinates of the texture.
5. *Clipping:* If the triangle is not in the field of view, this unit removes it from the pipeline, i.e. it is not processed further. This stage, also known as view frustrum clipping, is an instance of polygon culling.
6. *Projection:* The triangle is now projected onto the viewing plane. For this, one computes the points of intersection between the viewing plane and the straight lines connecting the corner points of the triangle with the current position of the viewer.
7. *Rasterization:* So far, every triangle has been given only by its corner points. Now a raster is put onto the viewing plane. The resolution of this raster usually relates directly to the resolution of the computer display. The cells of the raster are called pixels (= picture elements). Every pixel which lies within the triangle is given the color of the triangle. In fact, a second raster, the Z-buffer, stores the distance of the point from the current position of the viewer. If a pixel is to be set to a new color, the unit checks whether the point is closer to the viewer than the value currently stored in the Z-buffer. If this is true, the pixel is set to that color and the new distance is stored in the Z-buffer. That is, the hiding of one object by another object is computed pointwise.

For simplicity, we have assumed that a triangle has a unique color. This is also called “flat shading”. There are other shading algorithms, for example the Gouraud and Phong shading algorithms which use interpolation to compute the color for each pixel. More information on computer graphics and rendering algorithms can be found in established textbooks [FvDFH96, Wat93].

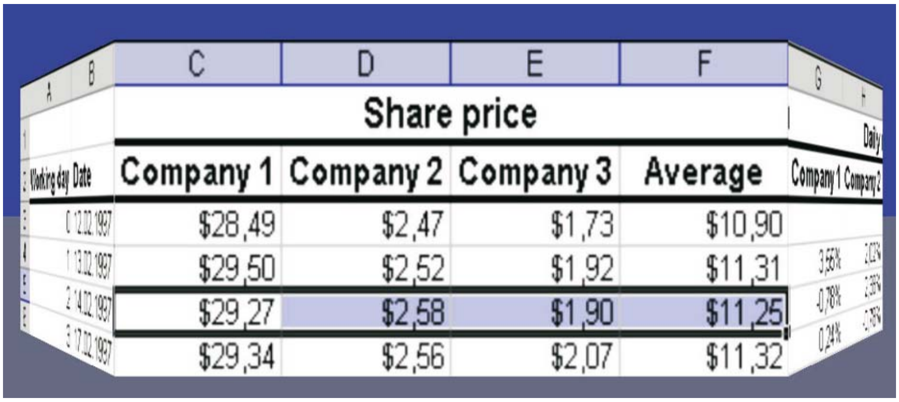
2.3 General Information Visualization Techniques

The goal of information visualization is to display abstract quantities and relations in a natural and intuitive way, in order to help the user better understand and gain insights into the data.

Many of the visualization techniques follow one of the following two principles:

Interactive Exploration: To explore the data, interactive visualizations should allow the user to first get an overview, then zoom and filter, and get the details on demand. This principle was called the Information Seeking Mantra by Ben Shneiderman [Shn96].

Focus + Context: A detailed visualization of some part of the information – the focus – is embedded within a visualization of the context, i.e. more coarse-grained information about parts related to the focus. Thus focus + context techniques provide both an overview and detail at the same time.



	A	B	C	D	E	F	G	H
1			Share price					Daily
2	Working day	Date	Company 1	Company 2	Company 3	Average	Company 1	Company 2
3	0	12.02.1997	$28,49	$2,47	$1,73	$10,90		
4	1	13.02.1997	$29,50	$2,52	$1,92	$11,31	3,56%	2,02%
5	2	14.02.1997	$29,27	$2,58	$1,90	$11,25	-0,78%	2,38%
6	3	17.02.1997	$29,34	$2,56	$2,07	$11,32	0,24%	-0,78%

Fig. 2.8. Perspective Wall

2.3.1 Visualization of Textual Data

The perspective wall [MRC91] (Fig. 2.8) is a three-dimensional technique to present textual (and graphical) data by integrating a central view of details with two perspective views, one at each side, to display the context. The technique uses horizontal perspective distortion to compress the visual representation of the context. The Tablelens [RC94] uses both horizontal and vertical distortion to show some cells of a table in more detail than others. The rows and columns that form the context can actually be reduced to the width or length of a pixel.

2.3.2 Graph Drawing

Graphs are mathematical structures widely used to describe relationships between objects. In graphs, the objects are called *nodes*, and the relationships are called *edges*.

Graphs are often characterized by one or more of the following properties. First of all, graphs can be directed or undirected, which means that the edges may have a direction or not. Graphs can be cyclic, i.e. contain a cycle, or acyclic, i.e. contain no cycles at all. A graph is disconnected if its nodes can be partitioned into two sets such that there is not a single edge between a node in one of the sets to a node in the other set. Otherwise, the graph is called connected. A graph is bipartite if its nodes can be partitioned into two sets such that there is not a single edge between two nodes within the same set. A graph is planar, if it can be drawn such that no two edges intersect. Otherwise, it is called nonplanar. Finally, a *tree* is a connected, directed graph which has exactly one node that has no incoming edges and all other nodes have exactly one incoming edge. The former node is called the root of the tree. Nodes with no outgoing edges are called leaves.

Graph drawing is the art of drawing a diagram of a graph to facilitate understanding of relations between objects [dBETT98, KW01, San96]. Graph-drawing techniques are used to lay out bus maps, family trees, word graphs, VLSI designs, and workflow diagrams, for instance. Graphs play a crucial role in software engineering. Petri nets, control-flow graphs, syntax trees, finite-state diagrams, and complex data structures are graphs.

The goal of drawing a graph is to convey its underlying information. To this end, various aesthetic criteria [PCJ96] have been investigated, including:

Crossing minimization: If a graph is nonplanar, it should be drawn with as few crossings as possible.

Bend minimization: Edges should have as few bends as possible. Recently, studies have shown that continuity of edges is more important than the number of bends.

Area minimization: The area of the drawing should be small and there should be a homogenous density or an even distribution of the nodes.

Length minimization: Shorter edges are more easy to follow. To this end, one tries to minimize the the total edge length and the length of the longest edge.

Angle maximization: Small angles between outgoing edges and in bends make it difficult to discriminate the edges.

Symmetries: Symmetries in the underlying graph should be reflected in the diagram.

Clustering: For large graphs, parts of the graph which are strongly interconnected – called clusters – should be drawn separate from other such parts. Edges drawn within a cluster should be shorter than those connecting different clusters.

Graph-drawing techniques differ in the kinds of graphs they are suitable for, as well as in the basic principles underlying the drawing algorithm. Some important classes of drawing algorithms are orthogonal, force-directed, hierarchical, tree, and circular layout algorithms. The following were produced using the DGD tool [GARG].

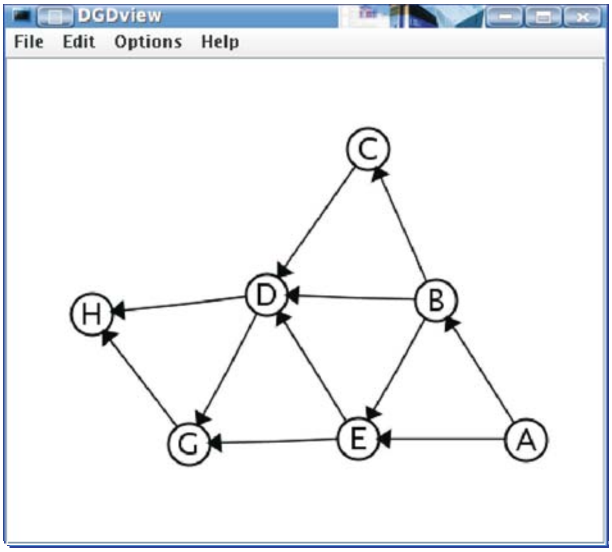


Fig. 2.9. Force-directed layout

Force-directed layout (Fig. 2.9): The graph is viewed as a physical system composed, for example, of springs connected to each other, where the springs represent the edges of the graph and the connection points the nodes [Ead84]. The layout is computed by minimizing the energy of the system. Often, in addition, other physical metaphors are such as polar and parallel magnetic fields or gravity are used.
Hierarchical layout (Fig. 2.10): The computation of a hierarchical layout of a graph following the Sugiyama approach [STT81] includes several phases. First all nodes are distributed in discrete layers (the ranking phase), then the nodes of each layer are arranged, and finally the layout, including the edge routing, is computed from the layers and their arrangement.
Orthogonal layout (Fig. 2.11): In an orthogonal layout, the edges run either horizontally or vertically and edge bends must have an angle of 90 degrees. There are many different kinds of algorithms for producing orthogonal layouts. Typically, the goal is to minimize the number of edge crossings and bends. Several of the algorithms are based on minimal network flow algorithms.

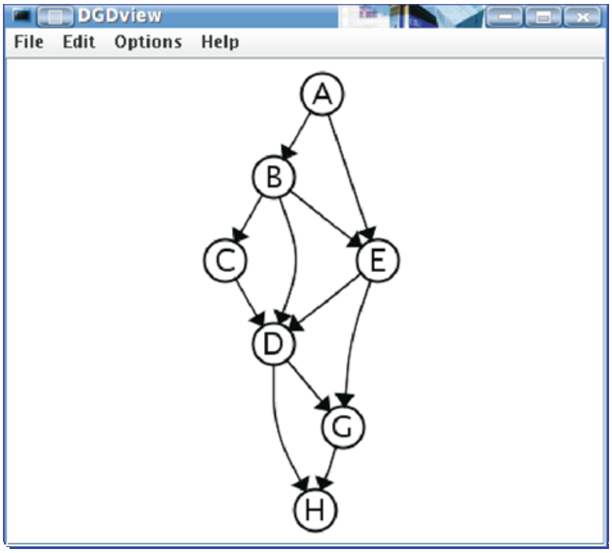


Fig. 2.10. Hierarchical layout

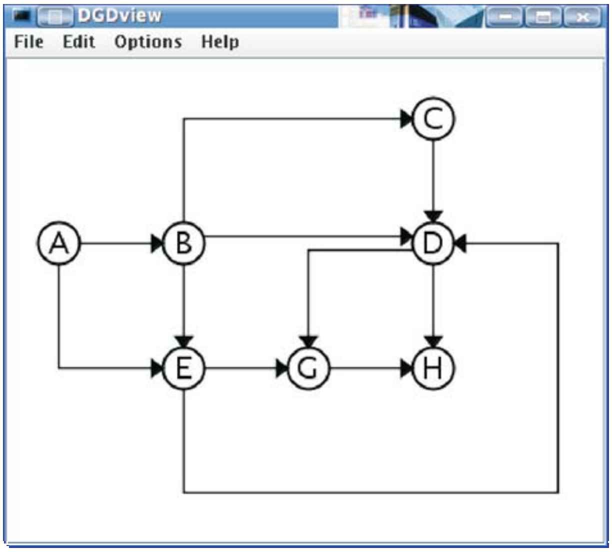


Fig. 2.11. Orthogonal layout

Dynamic graph drawing (Fig. 2.12): Dynamic graph drawing addresses the problem of layouting graphs which evolve over time by adding and deleting edges and nodes. This results in an additional aesthetic criterium known as "preserving the mental map" [MELS95]. The term *mental map* refers to the abstract structural information a user forms by looking at the layout of a

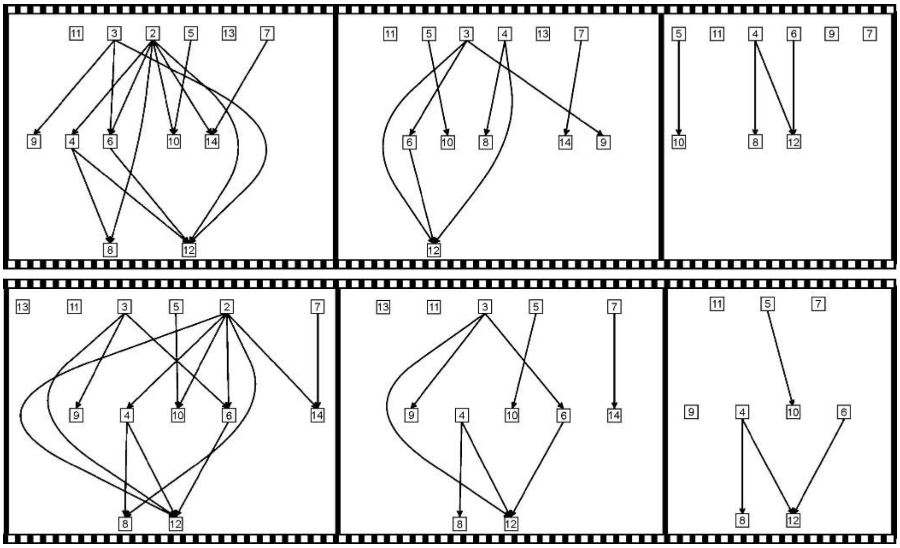


Fig. 2.12. Two graph animations of the same sequence of graphs

graph. The mental map facilitates navigation in the graph or comparison of it and other graphs. In the context of dynamic graph drawing changes to this map should be minimal. Incremental algorithms try to change the layout just as far as to accomodate the update. Unfortunately, in the worst case they have to compute the layout of the whole graph [Bra01]. In some applications all changes are even known beforehand, e.g. if we want to visualize the evolution of a social network based on an email archive, or the evolution of program structures stored in software archives. In these kinds of applications each graph can be drawn being fully aware of what graphs will follow [DG02, EHK$^+$03].

The two animations in Fig. 2.12 show the same sequence of graphs. The animation in the top portion of the figure was produced using an ad hoc approach, whereas the animation in the bottom portion of the figure was produced using the hierarchical FLT algorithm [GBPD04].

2.3.3 Visualization of Hierarchies

Hierarchies are a common and powerful tool for structuring information; graph-theoretically, hierarchies are trees.

Trees are typically drawn as node and edge diagrams, where each node is represented by a box, circle, or ellipse, while the edges are represented by lines (see the left part of Fig. 2.13. These kinds of tree diagram waste much space in the upper part, as the number of nodes per layer of the tree increases – in the worst case exponentially – with depth. Many screen-filling techniques have been developed to fit large hierarchies onto the screen and avoid empty screen space. Instead of connecting nodes by lines, most of these use alternative representations such as containment or neighborhood.

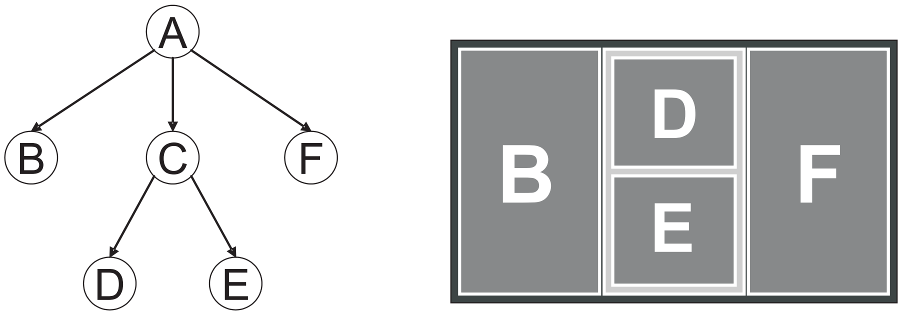


Fig. 2.13. Node and edge tree diagram vs. treemap

Treemaps [JS91] visually encode the child property by containment instead of connectedness. The basic idea of treemaps is to recursively divide an area into nonoverlapping subareas according to a given hierarchy. In many applications, the size of each subarea in the treemap is proportional to some weight, which in the simplest case is just the sum of all elements in the corresponding subtree. Many variants have been developed during the last decade, including ordered treemaps [SW01], squarified treemaps [BHvW00], cushion treemaps [vWvdW99], and Voronoi treemaps [BD05].

Information pyramids [AWP97] are a three-dimensional variant of treemaps. As with treemaps, every node is represented by a rectangle. The rectangle in the lowest layer represents the root, and layers with smaller rectangles representing subtrees are put on top. The size of each rectangle is proportional to the size of the subtree that it represents. Both information pyramids and treemaps provide a good overview, but it is difficult to focus on nodes other than leaf nodes.

Another approach used to cope with the exponential growth of nodes per layer are cone trees [RMC91]. cone trees are a three-dimensional extension of node-edge tree diagrams. For each layer the children of a single node are placed on a disk. By rotation, the nodes on the disk are brought to the front and others are moved to the background. A shadow of the whole cone tree is cast on the floor below the tree to provide a 2D overview of its structure and density. Navigation in cone trees is complicated, but they allow one to focus on a certain path in the tree while still showing the whole tree.

Information slices [AH98] and the visualizations of the Sunburst system [SZ00] are 2D diagrams that place nodes of the same subtree on a sector of a disk (or semidisk) and each layer of the tree is represented by a ring of the disk (see Fig. 2.14). Here the child property is visually encoded by proximity and color.

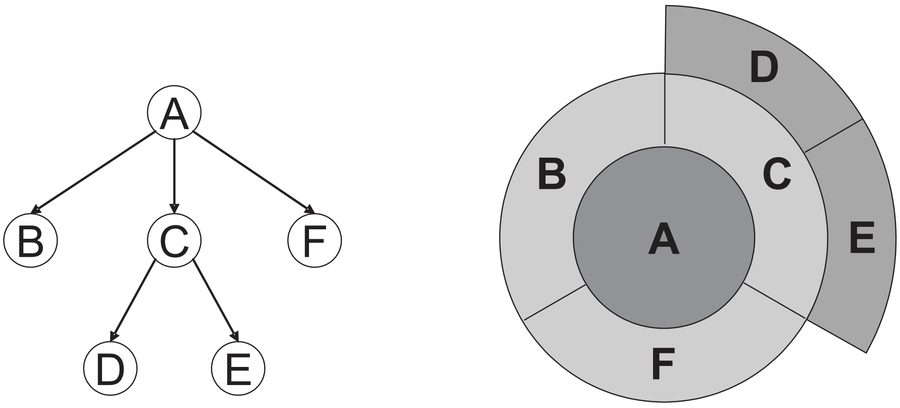


Fig. 2.14. Node and edge tree diagram vs. disk-based visualization

While the above visualization techniques try to exploit the screen space efficiently, they are poor visual metaphors when it comes to transferring properties from the visual objects to the represented objects. Botanical visualization [KvdWW01] takes the tree metaphor literally and represents hierarchies as trees in 3D. Nonleaf nodes are represented as branches, and sets of leaf nodes as fruit – a sphere with a spot or cone for each leaf node in the set.

2.4 Visual Metaphors

A *visual metaphor* is an analogy which underlies a graphical representation of an abstract entity or concept with the goal of transferring properties from the domain of the graphical representation to that of the abstract entity or concept. As Lakoff and Johnson put it [LJ80], "The essence of metaphor is understanding and experiencing one kind of thing in terms of another." A well-known example is the desktop metaphor that governs the design of the graphical user interface of most modern operating systems. A common problem with metaphors is that the properties that we transfer from the source to the target domain might be different from those originally intended by the designer [Mad94].

While simple two- or three-dimensional geometric metaphors such as circles, rectangles, cones, boxes, and cylinders, as well as graph- and matrix-based metaphors, still dominate in information visualization, other more sophisticated metaphors have been explored.

For geographic information, maps are widely used to display spatial information. As we are used to interpreting such maps and using them for navigation in the real world, it does not come as a big surprise that the *map metaphor* is often applied to display multidimensional abstract information by selecting two of the dimensions to span the 2D space as well.

The *landscape metaphor* goes even further. Here, three abstract dimensions span the 3D space, and real-world objects such as hills, valleys, rivers, and streets are used to represent abstract entities or relations. The *city metaphor*, which can be seen as a part of the landscape metaphor, represents abstract entities and relations by houses, buildings, towers, and streets.

Relational information is often depicted using the *galaxy* or *solar system metaphor* [GYB04]. Technically, such a visualization can be produced by using a graph layout algorithm such as force-directed layout or more sophisticated clustering algorithms to place the nodes in two or three dimensions. Nodes are displayed as planets or stars. Instead of drawing edges between these nodes, the relation is visually represented by proximity, i.e. two related nodes are placed close together.

When choosing visual metaphors, one has to make sure that the number of visual representations provided by the metaphor suffices to cover all objects in the software domain. Furthermore, the metaphor should be used in a consistent way[2]. Very often, the metaphor does not provide enough visual representations and the designers of a visualization resort to combining representations from different metaphors.[3]

2.5 Summary

In this chapter we have discussed that what we see, in particular its resolution and color, relies on the physiology of the eye, and that there are rules which guide the perception of patterns and motion. This knowledge can help to explain why certain visualizations work and others do not. Optical illusions help researchers to understand how the human visual cognition works. The show limits and problems that might occur in visualizations.

We have also briefly looked information visualization techniques and graph drawing, because they are used by many software visualization systems. Thus software visualization research gains from results in fields such as cognitive psychology and graph drawing, but also researchers in those fields can use software visualization as a test bed for their techniques and theories.

Exercises

Exercise 1: How would Fig. 2.13 and Fig. 2.15 look if you were to use triangular instead of rectangular areas, such that the areas of the triangles are proportional to the size of the elements and subtrees that they represent?

[2] For example, when a file on the Apple desktop is dragged onto the trash can, the file is deleted, while when a disk is dragged onto the trash can, it is only ejected.

[3] Who would ever put windows on a desktop in the real world?

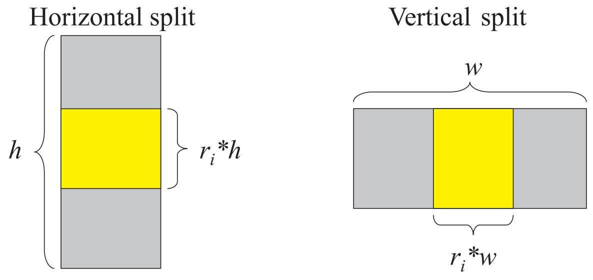


Fig. 2.15. Computing box sizes in a treemap

Exercise 2: Implement a treemap in Java. Given the name of a directory your program should show the treemap of the directory such that the size of each box indicates the size of the file or directory and the color of each box the age or type of the file. Let s_i be the size of the ith element in a directory. Then the total size s is $s = \sum_{1 \leq i \leq n} s_i$ and the ratio of the box for the ith element is $r_i = s_i/s$. The width and height of the boxes are computed as shown in Fig. 2.15.

To make things easier you can find on the Web page given in the Preface the Java source code of the class `DirectoryTree`. The method `getDirectoryTree()` of this class reads the name, size and age of a directory and all its subdirectories and files into a data structure. The source code also contains a class `LOCS` that maps numbers in the range 0–255 to a color scale. Make your visualization more interactive. When the user moves the cursor on a box, the name of the file or directory should automatically be shown below the cursor. (Hint: see class `MouseMotionListener`.)

3 Static Program Visualization

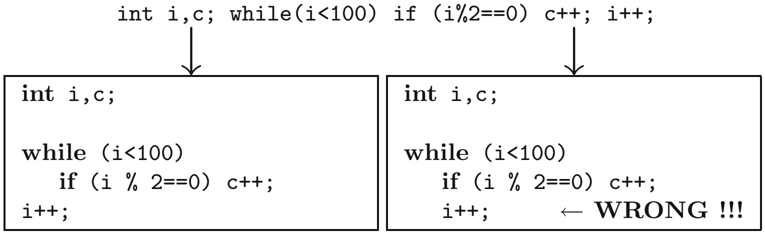
In this chapter we look at various ways to visualize the structure of a program given the program text as a sequence of characters. We discuss both text-based and diagrammatic methods, and take a closer look a syntax-directed, recursive method to compute the layout of control-flow diagrams. Then the computation of static properties of programs is discussed and we give some examples of systems that visualize results of static program analyses. Finally, we look at the visualization of structure of software at higher levels of abstraction, including its architecture.

3.1 Textual Representations

Program text is a sequence of characters. Typically one distinguishes two kinds of characters: printable and nonprintable characters. Contiguous sequences of printable characters form strings. Nonprintable characters such as blank and line feed separate these strings. There are special strings or characters such as parentheses or keywords, for example `begin` and `end`, that serve as delimiters. The strings enclosed by these delimiters form a block.

3.1.1 Pretty Printing

The goal of pretty printing is to make the nesting of these blocks visible while using a minimal number of lines for each block. Originally pretty printing was restricted to the use of indentation, spaces, and line breaks to make the structure of a program more explicit. Declarations can be vertically aligned by tabbing. Different widths of spaces in declarations make operator precedence more explicit. With the advance of technology, also fonts, font face, and colors are now used: for example, bold face may be used for keywords and italic for comments. Different font sizes can indicate nesting levels (lexical scope). As can be seen in the example in Fig. 3.1, if done wrongly pretty printing can suggest wrong nesting. In the pretty-printed text on the right, the statement `i++` seems to be part of the body of the loop.



```
int i,c; while(i<100) if (i%2==0) c++; i++;
```

```
int i,c;

while (i<100)
   if (i % 2==0) c++;
i++;
```

```
int i,c;

while (i<100)
   if (i % 2==0) c++;
   i++;      ← WRONG !!!
```

Fig. 3.1. Pretty printing

Automatic pretty printing is not a trivial problem. Among the issues addressed by pretty-printing algorithms are language independence, efficiency, use of available space such as line width and window height, and incremental updating in editors. Most current implementations are extensions and optimizations of Oppen's algorithm [Opp80]. This consists of two parallel processes. The first computes the size needed for each string and block, while the second uses this information to produce the final layout, taking the available line width into account.

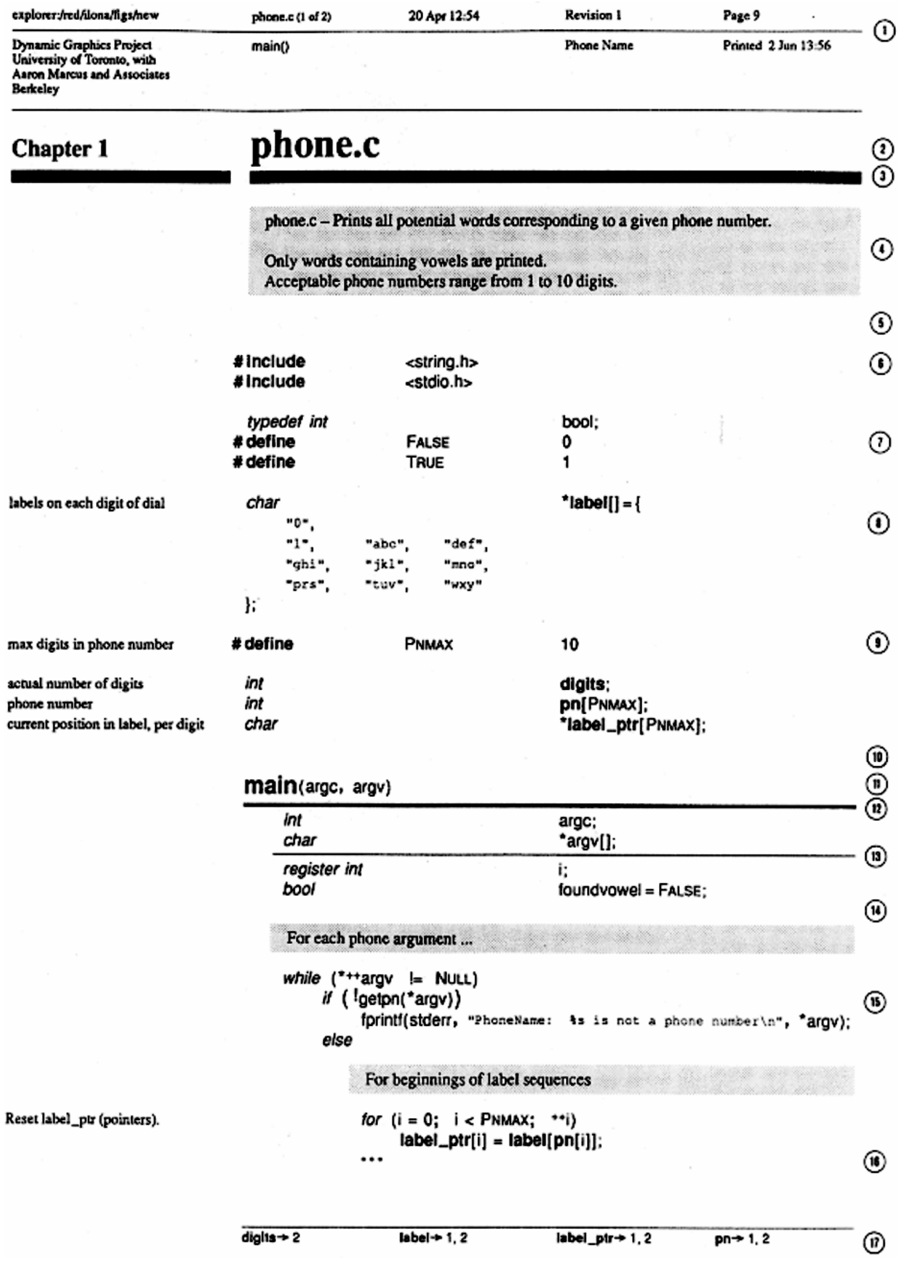
3.1.2 Program as Publication

In 1984 Donald Knuth, the inventor of the document typesetting system TEX, introduced the term *literate programming* [Knu84, Knu92], evangelizing the idea that programs should be considered as works of literature. To facilitate the production of well-documented and neatly typeset programs, Knuth developed the `WEB` tool:

> I'm pleased that my work on typography, which began as an application of computers to another field, has come full circle and become an application of typography to the heart of computer science.
>
> [Knu84]

With less focus on tool support, Baecker and Marcus investigated the use of typography to increase the readability of programs [BM98, BM89]. These authors suggested that *program books* should be produced with the same care as other textbooks. Their program book begins with the front matter, including a cover page, a title page, an abstract, a program history, information about the authors, and a table of contents.

The first chapter contains the user documentation, for example in the form of a tutorial on how to use the program. The second chapter gives an overview of the program structure through a program map and the call hierarchy. The program map is a table, with thumbnails of each program code



explorer:/red/ilona/figs/new | phone.c (1 of 2) | 20 Apr 12:54 | Revision 1 | Page 9

Dynamic Graphics Project
University of Toronto, with
Aaron Marcus and Associates
Berkeley

main() | Phone Name | Printed 2 Jun 13:56

Chapter 1 **phone.c**

phone.c – Prints all potential words corresponding to a given phone number.

Only words containing vowels are printed.
Acceptable phone numbers range from 1 to 10 digits.

```
#include            <string.h>
#include            <stdio.h>

  typedef int                              bool;
#define             FALSE                  0
#define             TRUE                   1
```

labels on each digit of dial

```
  char                                     *label[] = {
        "0",
        "1",        "abc",      "def",
        "ghi",      "jkl",      "mno",
        "prs",      "tuv",      "wxy"
};
```

max digits in phone number

```
#define             PNMAX                  10
```

actual number of digits
phone number
current position in label, per digit

```
int                                        digits;
int                                        pn[PNMAX];
char                                       *label_ptr[PNMAX];
```

```
main(argc, argv)
    int                                    argc;
    char                                   *argv[];

    register int                           i;
    bool                                   foundvowel = FALSE;
```

For each phone argument ...

```
    while (*++argv  != NULL)
          if ( !getpn(*argv))
                fprintf(stderr, "PhoneName:  %s is not a phone number\n", *argv);
          else
```

For beginnings of label sequences

Reset label_ptr (pointers).

```
                for (i = 0;  i < PNMAX;  ++i)
                      label_ptr[i] = label[pn[i]];
                ...
```

digits→ 2 | label→ 1, 2 | label_ptr→ 1, 2 | pn→ 1, 2

Fig. 3.2. A page from a program book (©1989 ACM)

page with major function names emphasized. Each subsequent chapter contains the pretty-printed program code of a source file (here files with extension `.c` and `.h`) with comments in the margins (see Fig. 3.2). The last chapter of the book provides the programmer documentation, including the installation and maintenance guides. At the end of the book several indexes such as the cross-references, caller index, and callee index are given. Finally, on the back cover page, the highlights of the content of the book are summarized.

3.2 Diagrammatic Representations

Since the early days of computer science, diagrams have been used to show the structure of programs. In these diagrams relations between program parts are visually encoded by the following methods:

Position: For example, the start node of a finite automaton is placed on the left and the start node of a control-flow graph is placed at the top, while the final node is placed on the right or at the bottom, respectively.
Linkage: Edges in a call graph, indicate which function invokes which other function, and edges in a control-flow diagram connect subsequent actions.
Neighborhood: In control-flow diagrams, alternative actions are often placed next to each other.
Containment: In Nassi–Shneiderman diagrams, a box representing a complex action contains the boxes of its subactions.

3.2.1 Jackson Diagrams

We start our discussion of diagrammatic techniques with Jackson diagrams, which decompose a program hierarchically [Jac75]. According to the Jackson structured-programming methodology, the data structures involved are first hierarchically decomposed using these diagrams, and then the program structure should follow this decomposition.

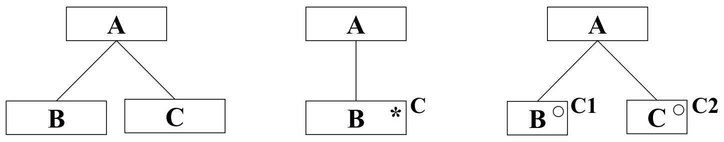
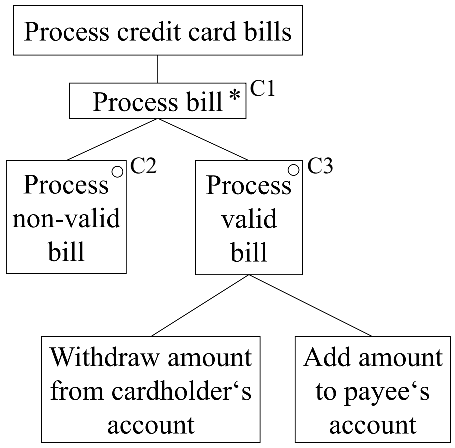


Fig. 3.3. Sequences, iterations, and alternatives in Jackson diagrams

The basic elements of Jackson diagrams are actions. Actions are decomposed into subactions, as shown in Fig. 3.3. A sequence `A` consists of the execution of a subaction `C` after a subaction `B`. An iteration `A` consists of multiple repetitions of `B` as long as an iteration condition `C` holds. Finally, an



Process credit card bills
Process bill *
C1
Process non-valid bill
C2
Process valid bill
C3
Withdraw amount from cardholder‘s account
Add amount to payee‘s account

Fig. 3.4. Example: Jackson diagram for designing a system

alternative `A` is either a subaction `B` if a condition `C1` holds, or a subaction `C` if a condition `C2` is true.

A Jackson diagram of the processing of credit-card bills is shown in the example in Fig. 3.4. Condition `C1` is true if the stack of bills is not empty, `C2` is true if the number of the credit card is not valid, and `C3` is true otherwise. The diagram shows the hierarchical decomposition of the task of processing credit card bills. Each credit card bill is processed one after another. There is a subaction for valid and nonvalid bills. For valid bills, the amount has to be withdrawn from the cardholder's account and added to the payee's account.

```
int fact(n) {
  if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

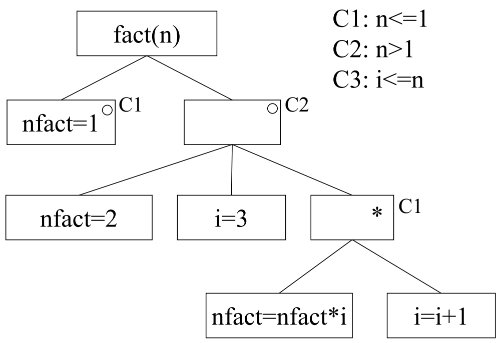


Fig. 3.5. Example: Jackson diagram of a factorial program

Jackson diagrams were originally meant for the top-down design of application programs. Nevertheless, we can also use them to visualize the structure of mathematical algorithms. For example, the Jackson diagram for a factorial program is shown in Fig. 3.5.

3.2.2 Control-Flow Graphs

In 1947 Goldstine and von Neumann [GvN47] introduced control-flow graphs (CFGs for short) to depict the structure of programs.

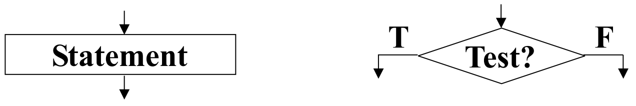


Fig. 3.6. Statements and alternatives in a control-flow graph

In these graphs, rectangular nodes represent events, activities, processes, functions, or statements, whereas nodes in the form of a diamond contain branch conditions and can have several exits (outgoing edges). Edges in the graph are drawn as arrows and depict transitions from one statement to another, i.e. the flow of control (see Fig. 3.6). Later, many more graphical elements were added and have been standardized in DIN 66001 (flowcharts).

```
int fact(n) { if (n>1)
   { nfact=2;
     int i=3;
     while(i<=n)
      { nfact=nfact*i;
        i=i+1;
      }
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

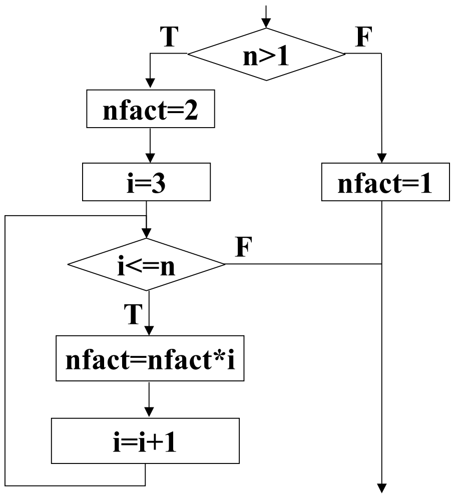


Fig. 3.7. Example: Control-flow graph of a factorial program

In Fig. 3.7 a control-flow graph of the factorial program shown earlier is depicted. To produce this graph, the `for` loop has been converted[1] into a `while` loop. For many applications of control-flow graphs, it is convenient to combine sequences of statements into a single node, called a basic block, as shown in Fig. 3.8.

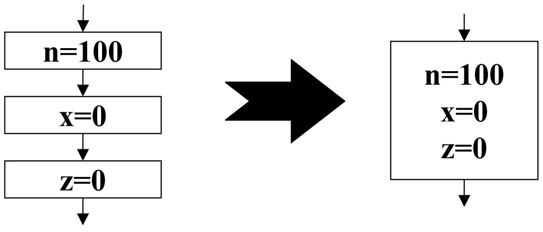


Fig. 3.8. Basic block

Automatic Generation of CFGs The first CFGs were drawn by hand to develop, explain or debug programs. But soon researchers developed programs to automatically compute and layout the control-flow graph of a given program [Hai59, Knu63]. One of those early systems was developed by A. E. Scott on an IBM 705 [Sco58]. The laid-out graph was drawn by a text printer with a very limited character set. Figure 3.9) shows a representation very similar to the one produced by Scott's algorithm for a factorial program. Forward edges are drawn to the right, backward edges to the left of the program text.

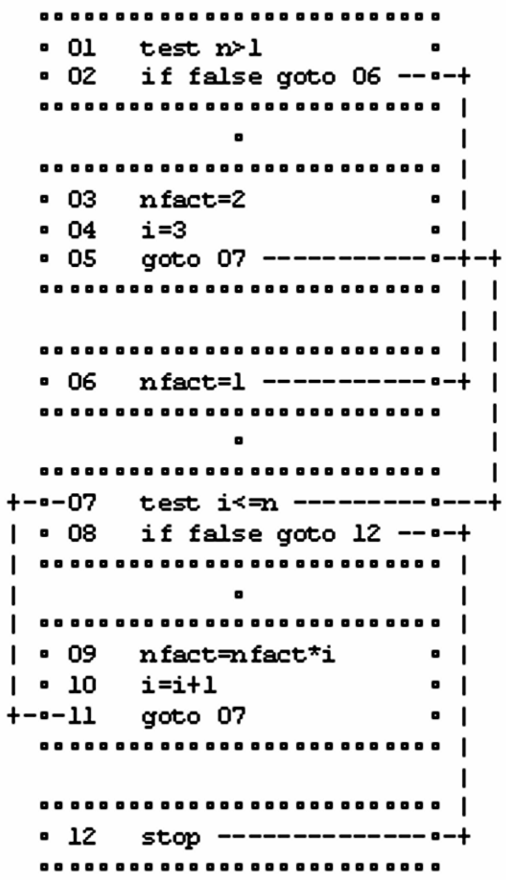
We shall now give the syntax of a simple programming language, define how to compute the CFGs of programs in that language, and, finally, give a simple layout algorithm for drawing such graphs.

Syntax of a simple programming language A program in the language that we shall consider in the rest of this chapter consists of assignments, alternatives, and loops. Expressions can occur on the right-hand side of an assignment, or in conditions of alternatives and loops. The syntax of the language is given in Fig. 3.10. To make the following presentation easier, we also assume that every statement, i.e. program point, has been given a unique identifier. In the following, we shall use the notation $L_G(A)$ to denote the language defined by a grammar G for a nonterminal A, or, more formally,

$$L_G(A) = \{w|w \text{ is a terminal word with } A \overset{G}{\rightarrow}^{*} w\}$$

In particular, $L_{G_{simple}}(\mathsf{S})$ is the set of all programs which can be written in our simple programming language.

[1] This is actually a phenomenon that we find very often in both static and dynamic program visualization: the program is transformed into a semantically equivalent program that can be visualized more easily.



```
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪
  ▪ 01    test n>1           ▪
  ▪ 02    if false goto 06 --▪-+
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
               ▪              |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
  ▪ 03    nfact=2            ▪ |
  ▪ 04    i=3                ▪ |
  ▪ 05    goto 07 -----------▪-+-+
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ | |
                              | |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ | |
  ▪ 06    nfact=1 -----------▪-+ |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪   |
               ▪                |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪   |
+-▪-07    test i<=n --------▪---+
| ▪ 08    if false goto 12 --▪-+
| ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
|              ▪              |
| ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
| ▪ 09    nfact=nfact*i      ▪ |
| ▪ 10    i=i+1              ▪ |
+-▪-11    goto 07            ▪ |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
                              |
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪ |
  ▪ 12    stop --------------▪-+
  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪
```

Fig. 3.9. Control-flow graph of a factorial program printed as text

$$
\begin{array}{rl}
G_{simple} = \{\ \mathsf{S} \longrightarrow & \mathsf{V{=}E}^{\ \mathsf{P}} \\
 & \mid \mathsf{S;S} \\
 & \mid \texttt{if (E) \{S\} else \{S\}}^{\ \mathsf{P}} \\
 & \mid \texttt{while (E) \{S\}}^{\ \mathsf{P}} \\
\mathsf{V} \longrightarrow & \text{variable name} \\
\mathsf{E} \longrightarrow & \text{expression} \\
\mathsf{P} \longrightarrow & \text{identifier of program point } \}
\end{array}
$$

Fig. 3.10. Syntax of a simple programming language

Computation of a CFG Let $s, s_i \in L_{G_{simple}}(\mathsf{S}), v \in L_{G_{simple}}(\mathsf{V}), p \in L_{G_{simple}}(\mathsf{P})$, and $e \in L_{G_{simple}}(\mathsf{E})$. Figure 3.11 shows how the CFG of a statement is built from the CFGs of the statements that it contains. For an assignment v=e^p, the CFG simply consists of a node representing the assignment.

For a sequence s_1;s_2, of statements there is an arrow from the end of the CFG built for the first statement to the entry of the CFG built for the second statement. For a loop `while(`e`) {`s`}`^{p}, a node with the condition e is connected (a true branch) to the CFG built for the body of the loop. From the end of that CFG, an arrow is drawn back to the loop condition. Finally, for the alternative `if(`e`) {`s_1`} else {`s_2`}`^{p}, a node representing the condition is connected to the CFG built for the first statement (a true branch) and to the CFG built for the second statement (a false branch).

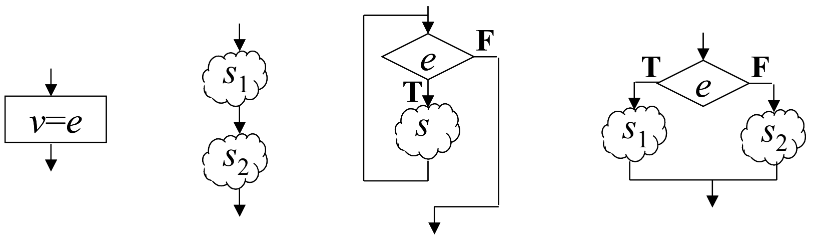


Fig. 3.11. CFGs for assignment, sequence, `while` loop, and alternative instructions

To precisely define the computation of a control-flow graph, we first give a formal definition of a control-flow graph. A control-flow graph is a tuple (V, E, in, out), where

- V is a set of nodes,
- $E \subseteq V \times L \times V$ is a set of edges,
- $L = \{\epsilon, \mathtt{T}, \mathtt{F}\}$ is a set of labels,
- and $in, out \in V$ are start and end nodes.

Note that according to this definition, a CFG has exactly one entry and one exit node. Let Γ be the set of all control-flow graphs. Below, we define a function $\mathsf{cfg} : L_{G_{simple}}(\mathsf{S}) \longrightarrow \Gamma$ which maps programs to control-flow graphs:

$\mathsf{cfg}(w) = (V, E, in, out)$, where

$$\text{if } w = v\texttt{=}e^p \text{ then } \begin{cases} V = \{w, in, out\}, \\ E = \{(in, \epsilon, w), (w, \epsilon, out)\}, \\ \text{and } in, out \ \text{ are two new nodes,} \end{cases}$$

$$\text{if } w = s_1\texttt{;}s_2 \text{ then } \begin{cases} V = (V_1 - \{out_1\}) \cup (V_2 - \{in_2\}), \\ E = \ (E_1 - \{(v, l, out_1) | v \in V_1\}) \\ \qquad \cup (E_2 - \{(in_2, \epsilon, v) | v \in V_2\}) \\ \qquad \cup \{(v_1, l_1, v_2) | (v_1, l_1, out_1) \in E_1, (in_2, \epsilon, v_2) \in E_2)\} \\ \text{and } in = in_1, out = out_2, \end{cases}$$

where $(V_1, E_1, in_1, out_1) = \mathsf{cfg}(s_1)$ and $(V_2, E_2, in_2, out_2) = \mathsf{cfg}(s_2)$.

An invariant in our construction is that an entry node has only one outgoing edge and no incoming edge, whereas an exit node can have many incoming edges, but has no outgoing edge. To combine two CFGs in a sequence all the incoming edges of the exit node of the first CFG are connected to the target node of the outgoing edge of the entry node of the second CFG:

$$\text{if } w = \texttt{while(}e\texttt{)\{}s\texttt{\}}^p \text{ then } \begin{cases} V = V_0 \cup \{e^p\}, \\ E = \quad (E_0 - (\{(in_0, \epsilon, v) | v \in V_0\} \\ \qquad\qquad \cup\{(v, l, out_0) | v \in V_0\})) \\ \qquad \cup \{(in_0, \epsilon, e^p), (e^p, F, out_0)\} \\ \qquad \cup \{(e^p, T, v) | (in_0, \epsilon, v) \in E_0\} \\ \qquad \cup \{(v, l, e^p) | (v, l, out_0) \in E_0\} \\ \text{and } in = in_0, out = out_0, \end{cases}$$

where $(V_0, E_0, in_0, out_0) = \mathsf{cfg}(s)$,

if $w = \texttt{if(}e\texttt{)\{}s_1\texttt{\}else\{}s_2\texttt{\}}^p$ then

$$\begin{cases} V = (V_1 - \{in_1, out_1\}) \cup (V_2 - \{in_2, out_2\}) \cup \{e^p\}, \\ E = \quad (E_1 - \{(in_1, \epsilon, v_1), (v_2, l, out_1) | v_1, v_2 \in V_1\} \\ \qquad \cup (E_2 - \{(in_2, \epsilon, v_1), (v_2, l, out_2) | v_1, v_2 \in V_2\} \\ \qquad \cup \{(in, \epsilon, e^p)\} \\ \qquad \cup \{(e^p, T, v) | (in_1, \epsilon, v) \in E_1\} \\ \qquad \cup \{(v, l, out) | (v, l, out_1) \in E_1\} \\ \qquad \cup \{(e^p, F, v) | (in_2, \epsilon, v) \in E_2\} \\ \qquad \cup \{(v, l, out) | (v, l, out_2) \in E_2\} \\ \text{and } in, out \text{ are two new nodes,} \end{cases}$$

where $(V_i, E_i, in_i, out_i) = \mathsf{cfg}(s_i)$.

Simple Layout of CFGs We use rectangular boxes and place in and out nodes in the middle of the upper and lower borders. Figures 3.12 and 3.13 show how the box heights for expressions and loops are computed.

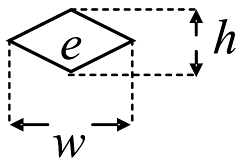


Fig. 3.12. Box heights of a CFG for an expression

For expressions, the height h depends on the font size, and the width also depends on the length of the expression. For a `while` loop, the heights h_1, h_2 and h_3 are fixed, whereas the heights h_e and h_s depend on the heights of the boxes computed for the loop condition e and the body s of the loop. So we can compute $H = h_1 + h_2 + h_3$ beforehand. As a result, we have $h = H + h_e + h_s$.

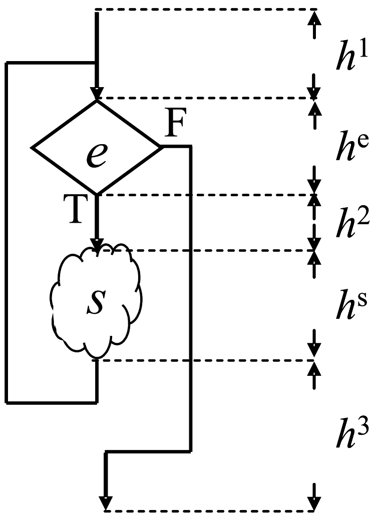


Fig. 3.13. Box heights of a CFG for a loop

More precisely, we can define the computation of the widths and heights of these boxes by a recursive function. We define a function $\mathsf{box} : L_{G_{simple}}(\mathsf{S}) \longrightarrow \mathcal{R} \times \mathcal{R}$ which maps each program to the size of the box required to lay out the control-flow graph of the program:

$\mathsf{box}(e) = (w, h)$ where w, h depend on the font.

$\mathsf{box}(s_1\,\texttt{;}\,s_2) = (W + \mathsf{max}(w_1, w_2), H + h_1 + h_2)$ where $(w_i, h_i) = \mathsf{box}(s_i)$

$\mathsf{box}(\texttt{while(}e\texttt{)\{}s\texttt{\}}) = (W + \mathsf{max}(w_e, w_s), H + h_e + h_s)$ where $(w_e, h_e) = \mathsf{box}(e)$ and $(w_s, h_s) = \mathsf{box}(s)$

Knowing the size of each box, we can easily draw the arrows, as we only have to connect them to the middle of the upper or lower border of a box.

3.2.3 Nassi–Shneiderman Diagrams

To enforce more structured programs, Nassi and Shneiderman introduced nested rectangular diagrams, also known as *structograms* [NS73]:

> Not only does this notation help the programmer to think in an orderly manner, it forces him or her to do so. . . . The absence of any representation of the GOTO or branch statement requires the programmer to work without it: a task which becomes increasingly easy with practice.
>
> [NS73]

The primitive diagrams are shown in Fig. 3.14.

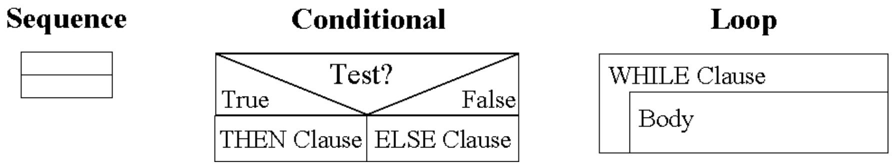


Fig. 3.14. Basic Nassi–Shneiderman diagrams

As an example, consider the Nassi–Shneiderman diagram of the factorial program shown in Fig. 3.15. This consists of an alternative followed by a return statement. In the left part of the alternative we see the loop for $n > 1$; in all other cases, the right part is executed.

```
int fact(n) { if (n>1)
   { nfact=2;
     for(int i=3;i<=n;i++)
       nfact=nfact*i;
   }
  else
   { nfact=1;
   }
  return nfact;
}
```

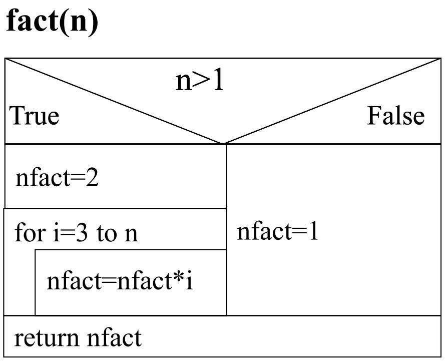


Fig. 3.15. Nassi–Shneiderman diagram of a factorial program

The kinds of control flow that can be modeled with these diagrams are restricted by the fact that rectangles are always disjoint from or fully enclosed by other rectangles. There is no overlap. A similar restriction is true for the control-flow graphs that we constructed above for our simple programming language, but if we were to add jumps to our language this would no longer be the case: $G_{spaghetti} = G_{simple} \cup \{$ $\mathsf{S} \longrightarrow \mathsf{goto\ L}$ $^{\mathsf{P}}$, $\mathsf{S} \longrightarrow \mathsf{L : S}$ $^{\mathsf{P}}$, $\mathsf{L} \longrightarrow$ label of program point $\}$. Actually, with these jumps we can even specify programs that have a nonplanar control-flow graph, i.e. that can not be drawn without edge crossings. Figure 3.16 shows the smallest (with respect to the number of edges) such control-flow graph. In essence, it is a bipartite graph with six nodes, i.e. there are two partitions with three nodes each and each node in one partition is connected to every node in the other partition (see the Kuratowski reduction theorem in [Tho81]). In graph theory, this graph is often denoted by $\mathcal{K}_{3,3}$.

```
L1: if (a1>0) goto L5
         else goto L4
L2: if (a2>0) goto L6
         else goto L5
L3: if (a3>0) goto L4
         else goto L6
L4: if (a4>0) goto L2
L5: if (a5>0) goto L2
L6: if (a6>0) goto L1
```

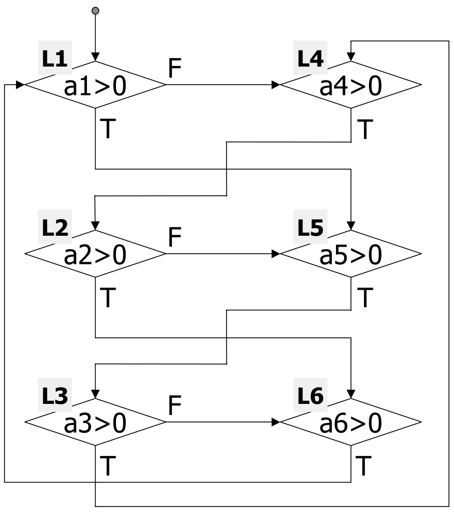


Fig. 3.16. The smallest nonplanar control-flow graph

3.2.4 Control-Structure Diagrams

Both control-flow graphs and Nassi–Shneiderman diagrams deviate from the sequential order of the program parts in the source code. Control-structure Diagrams [CHM98] keep this order, but make the nesting and scope of program constructs more explicit by augmenting the indented program text with a horizontal tree. Vertical lines show the extent of blocks, and vertically stretched oval lines show that of loops. Diamonds indicate the alternatives of conditional statements (see Fig. 3.17). The GRASP tool [Cro] automatically produces control-structure diagrams from program code. There exist versions for Ada 95, C, C++, and Java.

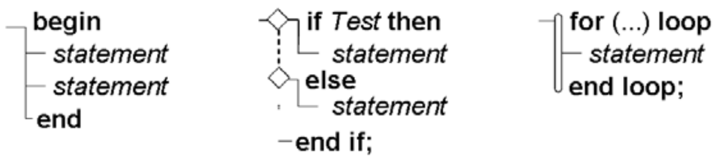
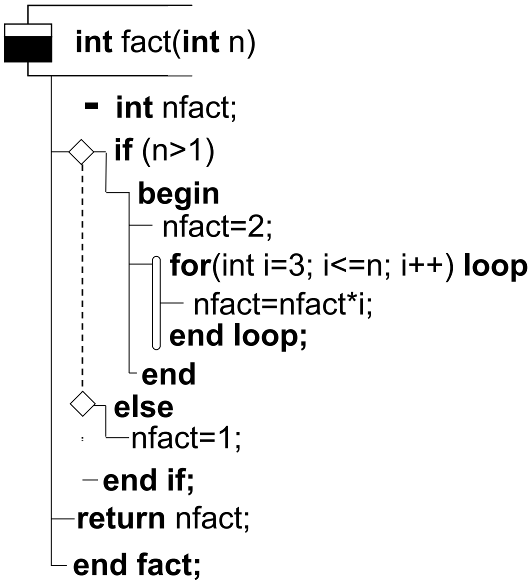


Fig. 3.17. Basic control-structure diagrams: sequence, conditional, and loop

There are many more basic control-structure diagrams that can be used to build program representations from. Here we have only chosen those which we need for our running example, the factorial program. Its control-structure diagram is shown in Fig. 3.18.



```
int fact(int n)
    int nfact;
    if (n>1)
        begin
            nfact=2;
            for(int i=3; i<=n; i++) loop
                nfact=nfact*i;
            end loop;
        end
    else
        nfact=1;
    end if;
    return nfact;
end fact;
```

Fig. 3.18. Control-structure diagram for factorial program

3.3 Visualizing the Results of Program Analyses

In this section, we shall look at static program analyses. First we shall discuss why control-flow analysis, i.e. the computation of the CFG for a given program, is not always as simple as it is for the language that we used in the previous section. Then we shall give an introduction to data-flow analysis and look at two such analyses in more detail. Finally we shall give some examples of systems that visualize such and similar analysis results.

3.3.1 Static Analysis

Static analysis computes properties of a program which hold for all executions of the program [NNH99]. It is important to note that not every property can be computed, because there is no general program that takes another program as its input and decides whether this program finishes after finitely many steps or runs forever.[2] We call those properties which cannot be computed before run time "dynamic". For example, the number of times a program point is executed for a given input is a dynamic property, while the fact that a program point is not executed for any possible input is a static property.

[2] This fact was proven by Alan Turing in 1936 and is known as the undecidability of the halting problem [Tur36].

3.3.2 Control-Flow Analysis

Control-flow analysis computes the control-flow graph of a program. For the sample language defined earlier in this chapter, the algorithm presented was very easy. If we consider real programming languages, the problem becomes much harder. Such programming languages typically have procedural abstraction. Each procedure has its own control-flow graph, but owing to procedure calls within the body of a procedure, these graphs are interconnected. Thus we distinguish intraprocedural and interprocedural control-flow graphs.

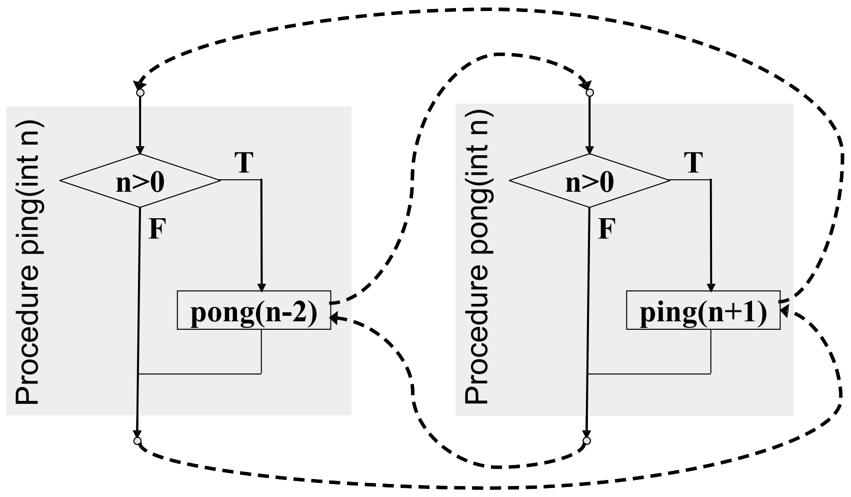


Fig. 3.19. Interprocedural control-flow graph

So, if we have a call of procedure q in the body of procedure p, we draw an arrow from the program point of the call to the entry node of procedure q and an arrow from the exit node of q back to the call. Figure 3.19 shows a control-flow graph of the following simple program:

```
void ping(int n)
 { if (n>0) pong(n-2); }

void pong(int n)
 { if (n>0) ping(n+1); }
```

The gray boxes contain the intraprocedural control-flow graphs of the two procedures. The interprocedural call and return edges are shown by dashed arrows.

Many modern programming languages have function pointers. They are a key feature of higher-order functional languages, but they also exist in C. The problem is that the value of such a function pointer is computed at run time and thus it can point to all functions of the program or at least all functions of a certain type. As a consequence, we have to draw edges from a program point that calls a function via a function pointer to all these functions. A similar problem occurs in object-oriented (OO)languages such as Java. There, we do not have function pointers directly, but references to objects which contain functions or, in OO lingo, methods. What method is called depends on the runtime type of the object referred to. Because of this dynamic dispatch of methods, the interprocedural CFG typically contains edges to all of those methods which might be called on the basis of the static type of the reference. By computing better approximations of the runtime type of the reference [Pro02], the number of possible targets of a method call can be considerably reduced.

3.3.3 Data-Flow Analysis

Data-flow analysis computes information for each program point about the data that will reach this program point during execution. Data-flow analyses provide important information for optimizing compilers [WM95, Lem92]. In general, data-flow analysis works by propagating locally available information over the paths in the control-flow graph. We distinguish two kinds of flow problems on the basis of the direction the information is propagated along the edges:

Forward-flow problems: what can happen before control reaches this program point, for example reaching definitions or available expressions?

Backward-flow problem: What can happen after control leaves this program point, for example live variables, very busy expressions, or reached uses?

Accordingly, for each node v of the control-flow graph, we compute two functions. The function $IN(v)$ yields information about the state before the program point is executed, and the function $OUT(v)$ yields information about the state after the program point is executed.

Available Expressions As an example of a forward-flow problem, we look at the computation of available expressions. A binary expression e_1 *op* e_2 consists of an operation and two expressions e_1 and e_2 which can contain binary expressions themselves. A binary expression is available at a program point p if it has been computed before, i.e. it has to be computed along every path by which p can be reached. Assume that the expression e occurs at program point p and is available on all incoming paths. Let $p_1, \ldots, p_n$ be the program points where the expression was computed before. Then the program can be optimized by inserting an assignment `x=`e to a new temporary variable `x` before each program point $p_1, \ldots, p_n$ and replacing all occurrences of e in $p_1, \ldots, p_n$ and in p by `x`. As a result, the expression is only computed once and we avoid avoid recomputation.

Let (V, E, in, out) be a control-flow graph and let $\mathcal{E}$ be the set of all binary expressions which occur in the program. Furthermore, let us define two functions $GEN, KILL : V \rightarrow \mathcal{P}(\mathcal{E})$ which map program points to sets of binary expressions. The function $GEN()$ yields those binary expressions which are computed at the program point, while $KILL()$ yields the empty set, unless the program point is an assignment. In this case it yields all those expressions in $\mathcal{E}$ that do not contain the variable on the left-hand side of the assignment. The idea is that the values previously computed for these expressions are not valid beyond this program point. The functions are defined as follows:

$$GEN(v) = \begin{cases} \{e' | e' \text{ is a binary subexpression of } e\} & \text{if } v = e \\ \{e' | e' \text{ is a binary subexpression of } e \\ \quad \text{and } x \text{ does not occur in it } \} & \text{if } v = x\texttt{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

$$KILL(v) = \begin{cases} \{e' | e' \in \mathcal{E} \text{ and } x \text{ occurs in } e'\} & \text{if } v = x\texttt{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

The actual data-flow analysis is performed by the following algorithm, which computes the functions $IN()$ and $OUT()$ by iterating over all nodes of the control-flow graph and propagating information forward by using the values of the function $IN()$ to compute those of the function $OUT()$:

Algorithm 1 (available expressions)

```
$IN(in) = \emptyset$
$OUT(in) = \emptyset$
$for\ all\ v \in (V - \{in\})\ do$
   $IN(v) = \mathcal{E}$
   $OUT(v) = (IN(v) - KILL(v)) \cup GEN(v)$
$while\ there\ are\ changes\ do$
   $for\ all\ v \in (V - \{in\})\ do$
           $IN(v) = \bigcap_{(p,l,v) \in E} OUT(p)$
           $OUT(v) = (IN(v) - KILL(v)) \cup GEN(v)$
```

Example: Available Expressions To illustrate the algorithm, we show the steps of the computation of available expressions for the control-flow graph shown in Fig. 3.20.

- *Initialization:*

```
E={y+2, n/2, 2*y, n/2+2*y, n/2-2*y, x-2*y}
IN(in)={}   OUT(in)={}
IN(p1)=E    OUT(p1)=E
IN(p2)=E    OUT(p2)={y+2, n/2, 2*y, n/2+2*y, n/2-2*y}
IN(p3)=E    OUT(p3)={y+2, n/2, 2*y, n/2+2*y, n/2-2*y}
IN(p4)=E    OUT(p4)={n/2}
IN(out)=E   OUT(out)=E
```

```
GEN(in)=
GEN(p1)=y+2
GEN(p2)=n/2, 2*y, n/2+2*y
GEN(p3)=n/2, 2*y, n/2-2*y
GEN(p4)=
GEN(out)=

KILL(in)=
KILL(p1)=
KILL(p2)=x-2*y
KILL(p3)=x-2*y
KILL(p4)=y+2, 2*y, n/2+2*y,
         n/2-2*y, x-2*y
KILL(out)=
```

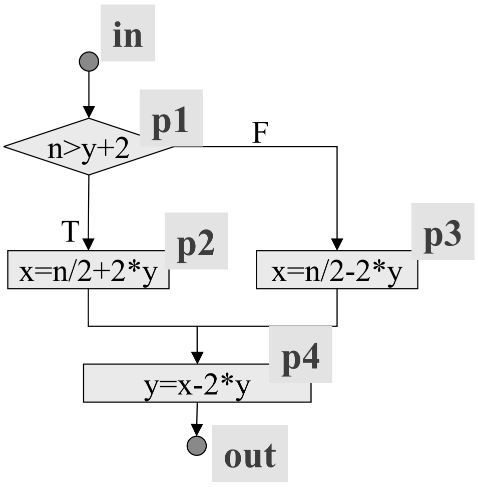


Fig. 3.20. Example: available expressions

- *First iteration:*

```
IN(p1)={}                  OUT(p1)={y+2}
IN(p2)={y+2}               OUT(p2)={y+2, n/2, 2*y, n/2+2*y}
IN(p3)={y+2}               OUT(p3)={y+2, n/2, 2*y, n/2-2*y}
IN(p4)={y+2,n/2, 2*y }     OUT(p4)={n/2}
IN(out)={n/2}              OUT(out)={n/2}
```

- *Second iteration:* in the second iteration, no changes are computed in either of the functions. Thus a fixpoint is reached.

Live Variables As an example of a backward-flow problem, we discuss the computation of live variables. A variable x is live at a program point p

- if there is a path from p to p' and x is used at p', i.e. it occurs in an expression,
- and there is no redefinition of x (assignment to x) along that path.

If a variable is not live at a program point, there is no subsequent access to its value. Thus, an optimizing compiler could produce code that does not store the value of the variable in memory or in a register beyond this program point.

As before, we define the two functions $GEN, KILL : V \to \mathcal{P}(L_G(V))$, which in this case map program points to sets of variables:

$$KILL(v) = \begin{cases} \{x\} & \text{if } v = x\text{=}e \\ \emptyset & \text{otherwise} \end{cases}$$

$$GEN(v) = \begin{cases} \{x' | x' \text{ occurs in } e\} & \text{if } v \in \{x\text{=}e, e\} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that for assignments, the case $x' = x$ is possible. The following algorithm performs the backward data-flow analysis: the values of the function $OUT()$ are computed using those of the function $IN()$:

Algorithm 2 (Live Variables)

```
for all $v \in V$ do
    $IN(v) = GEN(v)$
while there are changes do
    for all $v \in V$ do
            $OUT(v) = \bigcup_{(v,l,s) \in E} IN(s)$
            $IN(v) = (OUT(v) - KILL(v)) \cup GEN(v)$
```

3.3.4 Examples of Visualization of Analysis Results

Some other examples of static analyses are the computation of upper bounds on the stack usage for each program point, as shown in Fig. 1.4; the inference of types for all expressions in a program; and the computation of upper bounds on the worst-case execution times for loops, functions, or complete programs.

In Fig. 3.21 we show a control-flow graph drawn with the aiSee tool [Ang], a graph-drawing tool used by StackAnalyzer [EB02, Ang] for visualization. Each program point is annotated with live variables.

Another tool that my be used to visualize graphs for program analysis is VISTA [Lab]. This combines different graphs in a single view. In the example in Fig. 3.22, control-flow graphs (CFG), data-dependency graphs (DDG) and control-dependency graphs (CDG) are shown. In control-dependency graphs, statements are dependent only on their preceding condition or the entry node. Control-dependency graphs are similar to Jackson diagrams in that they show the hierarchical dependency, but there is no order on the children of a node. Unlike the syntax-directed layout algorithm for control-flow graphs that we sketched in Sect. 3.2.2, the algorithms used by VISTA are not application-specific, but are general graph-drawing algorithms.

By combining, for example, the control-flow graph with the data-dependency graph, one can see what values in what statements influence other values in other statements and along which paths these statements can be reached.

In Sect. 4.4.7, we shall return to static program analysis and discuss how it can be combined with dynamic program visualization.

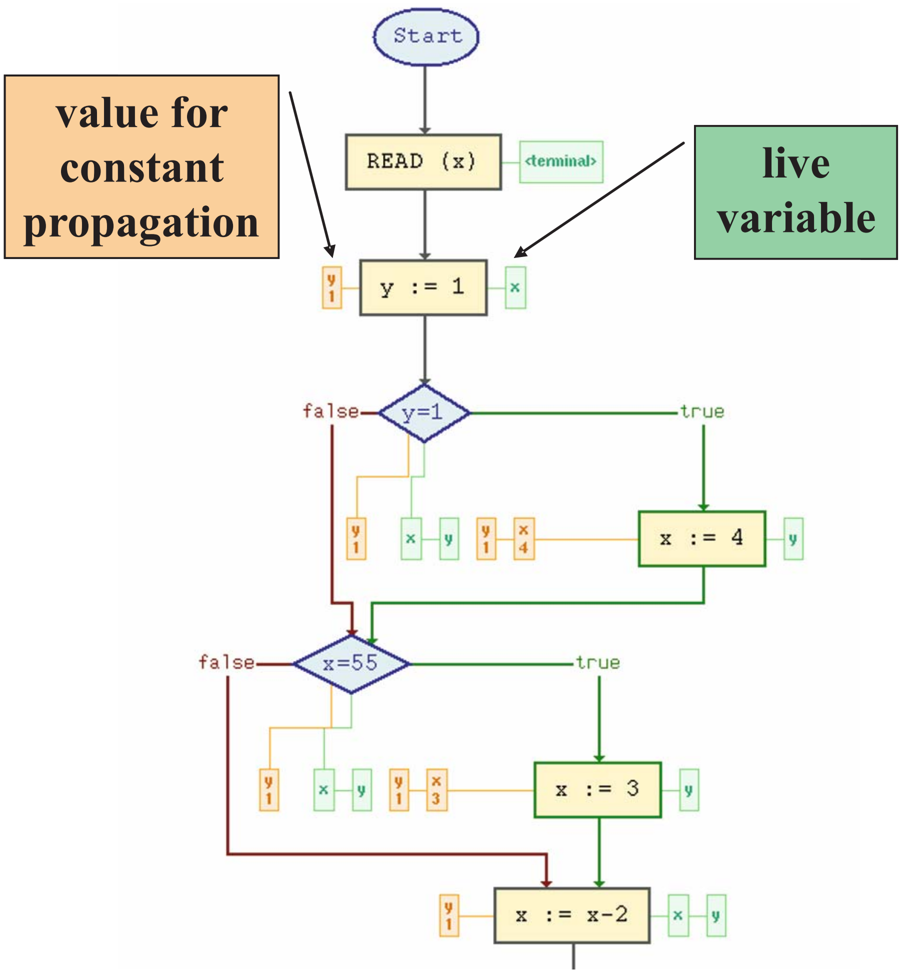


Fig. 3.21. Live variables shown with aiSee

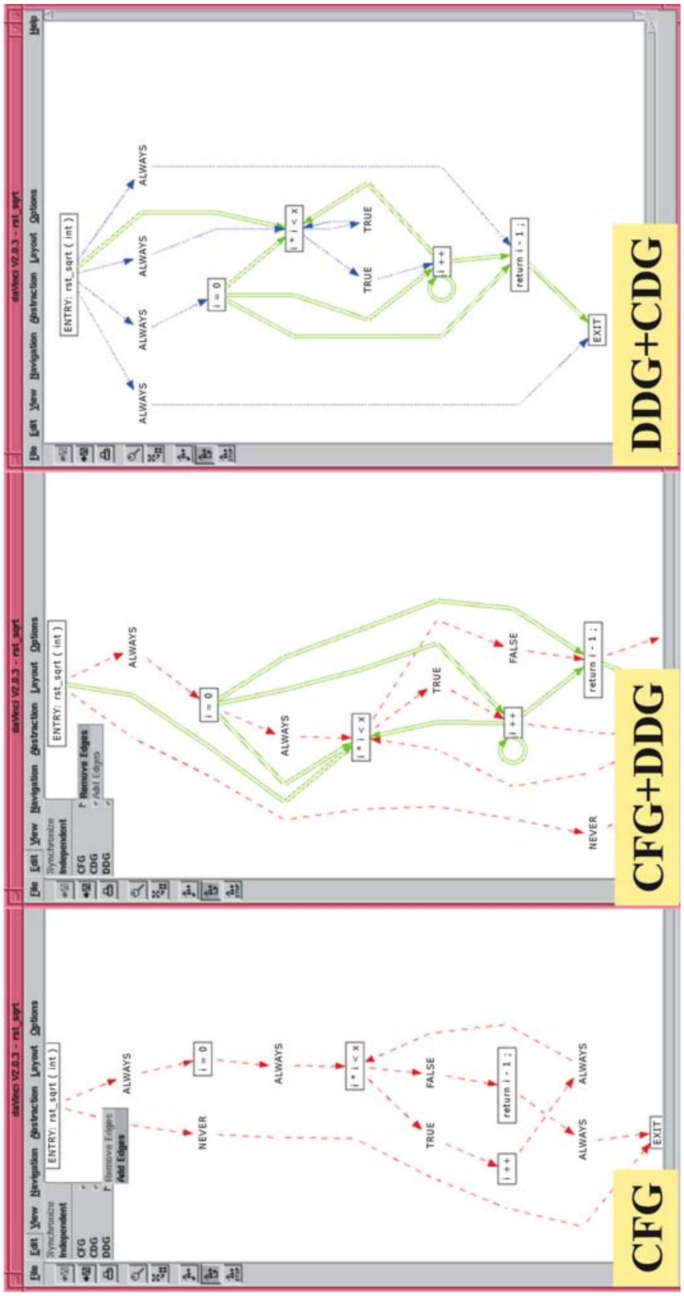


Fig. 3.22. Various kinds of graphs drawn by VISTA

3.4 Visualizing Software Architectures

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem.

[GS93b]

According to an IEEE standard [IEE00] "Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." The keywords in this definition are "structure", "environment" and "principles". As will become apparent in this chapter, visualization so far has concentrated mostly on the structure. Visualizations of software architectures typically deal with the structure at various levels of abstraction. At a higher level, the architecture consists of components with ports, and ports are linked through connectors.

When it comes to actually describing an architecture, there are many aspects, including its gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives [GS93a]. Most of these aspects have both functional and nonfunctional properties and are often described in natural language illustrated with diagrams.

Breaking a system into modules facilitates the design and development of software systems. As David Parnas [Par72] put it, "The existence of the hierarchical structure assures us that we can 'prune' off the upper levels of the tree and start a new tree on the old trunk."

It is widely accepted in software engineering that when one is designing software architectures there should be low coupling between modules and high cohesion within modules [SMC74]. Coupling is "a measure of the strength of the association established by a connection of one module to another", while cohesion is "the degree of connectivity among the elements of a single module".

However, these original notions of coupling and cohesion do not take the direction of the dependencies into account. In particular, in relation to object-oriented design, these original ideas have been extended and new principles and related metrics have been devised to guide the design of software systems.

The first part of this chapter is about types of diagrams that show the structure of an architecture. Then we shall look at some approaches that can be used to extract and visualize architectural information from the source code of a system. Finally, the use of 3D and dynamic software architecture visualization is discussed.

3.4.1 Some Familiar Architectures

We first look at some diagrams of some basic architectures widely used in software systems:

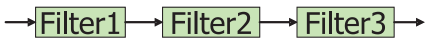


Fig. 3.23. Pipes and filters

Pipes and filters (Fig.3.23): Filters receive a stream of input data and produce a stream of output data. Pipes pass the output data of one filter as input data to the next one. The Infopipes notation extends the pipe metaphor with buffers, pumps, split and merge tees, etc. to design distributed streaming applications [BHK$^+$02].

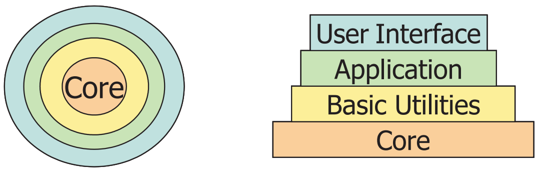


Fig. 3.24. Layered systems

Layered Systems (Fig.3.24): The functionality of a system is organized into several layers. In a purely layered system, the functionality of one layer is implemented by the functionality provided by the layer directly below. Very often, layered systems also allow access to some of the other layers below. One can use both horizontal layers or an onion model. In the first case all layers have the same size, whereas the onion model emphasizes that the core is smaller than the outer layers.

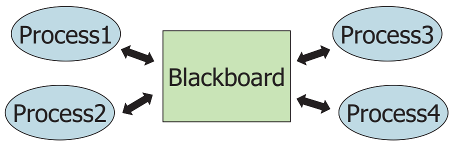


Fig. 3.25. Blackboard

Blackboards (Fig.3.25): In the blackboard architecture, there are multiple units that share data through a blackboard. Typically, the units can read and write to the blackboard. So for example, the blackboard might contain both tasks to be computed and results of previously computed tasks.

A Web search with a search engine such Google for images related to the term *software architecture* reveals a wealth of different styles for drawing architecture diagrams. Most of these use ad hoc visual representations, and the semantics of the colors, nodes, icons, lines, and arrows is often unclear. To remedy this situation somewhat, one can follow general rules for the use of connectors, icons, text, color, etc. On the basis of a study of software architecture diagrams found on the Web, Koning et al. compiled a list of guidelines for drawing architecture diagrams [KDvV02]. But when it comes to building large systems with many developers, a common understanding of the architecture diagrams is key, and standardized graphical notations such as UML promise to be the solution.

3.4.2 The Unified Modeling Language (UML)

UML is a set of graphical notations for modeling software systems. It combines the methods of Booch, Rumbaugh and Jacobsen. Each of these three famous software engineers, sometimes called the “Three Amigos”, had developed a different popular design method consisting of a graphical notation and a process for developing a design. To end what was called a “method war” they decided to merge their methods, and in 1997 they proposed UML to the Object Management Group (OMG) as a standard for modeling software systems.

UML provides a considerable number of different types of diagrams, including use case diagrams, class and object diagrams, behavior diagrams (statechart diagrams and activity diagrams), interaction diagrams (sequence diagrams and collaboration diagrams), implementation diagrams (component diagrams and deployment diagrams), and model-management diagrams (packages, subsystems, and models). In the following, we look only at a very small subset of UML.

In the class diagram in Fig. 3.26, classes and objects are represented by framed boxes; additionally, to mark objects, the object’s name is underlined. An arrow from a class A to a class B means that A inherits from B. A dotted arrow from an object to a class means that the object is an instance of the class. Inside the boxes, properties (attributes and operations) of the class can be declared additionally.

In addition to the inheritance relation, other relations are possible between classes. One important relation is aggregation. “Class A aggregates class B” means that objects of class A can contain objects of class B. Normally this happens because the value of an attribute in A contains an object or many objects of class B. Aggregation is represented by an edge which starts with a diamond on the side of the aggregating class. Additionally, one can determine how many objects will be aggregated (the multiplicity), indicated by numbers or intervals: `1..n`, `n..k`, `1..*`, `*`.

As another example of a UML diagram type, consider the sequence diagram in Fig. 3.27. This shows the time lines of four communicating objects.

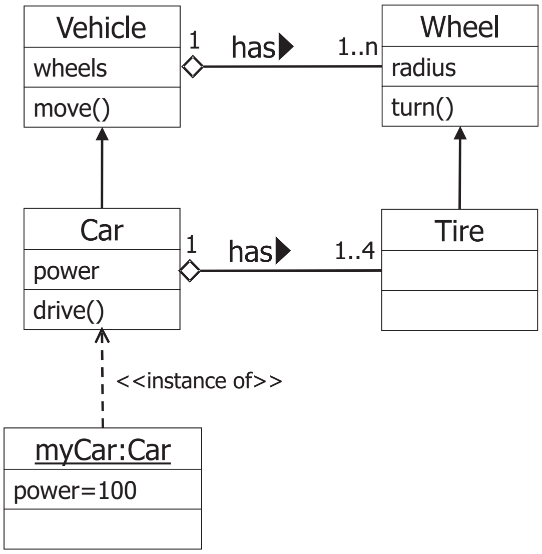


Fig. 3.26. UML class and object diagram

Time proceeds from top to bottom. First, the object `customer` sends the message `select product` to the object `store`. Next, it sends the message `order product`, and at the same time, when it receives this event, the object `store` sends the event `order balance` to the object `customerAccount`. A sequence diagram shows one possible sequence of messages between objects.

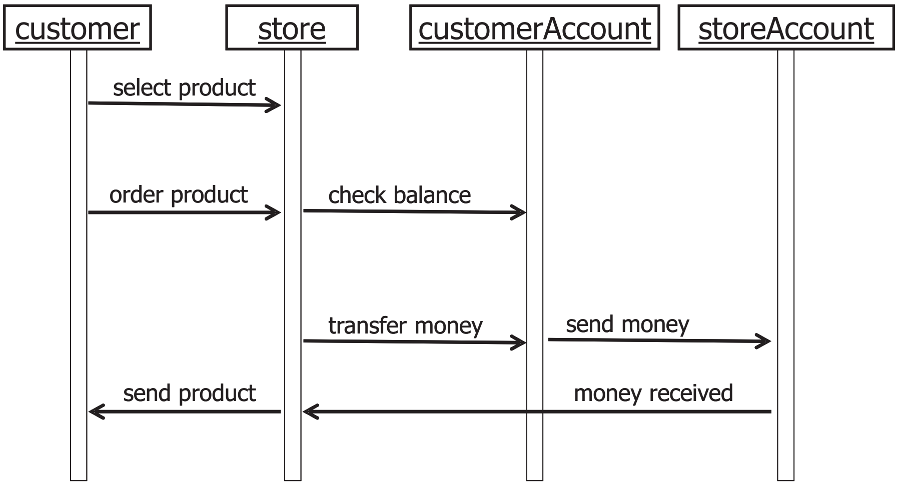


Fig. 3.27. A UML sequence diagram

UML diagrams are composed from a small set of graphical primitives: basically, these consist of text, boxes, lines, and arrows. As a result, designers can easily draw UML diagrams by hand without colored pencils and stencils. Despite the widespread use of this method, the visual efficiency of these diagrams is low. In recent user studies, geons have been used to draw diagrams of software architectures [IW00, IWT01]. The geons are a collection of 24 primitive, viewpoint-invariant 3D objects, which means that they are easy to recognize even when projected into 2D. Several experiments with computer science students showed that the subjects could visually analyze geon diagrams much faster and with more accuracy, and that they could recall them better in comparison with equivalent UML diagrams. Figure 3.28 shows a UML diagram and a geon diagram of a car. A car inherits from "conveyance" and consists of a motor and several wheels. Note that, unlike typical UML diagrams, in the geon diagram the geons of the aggregated classes are also drawn, in a reduced size, within the geon representing the aggregating class.

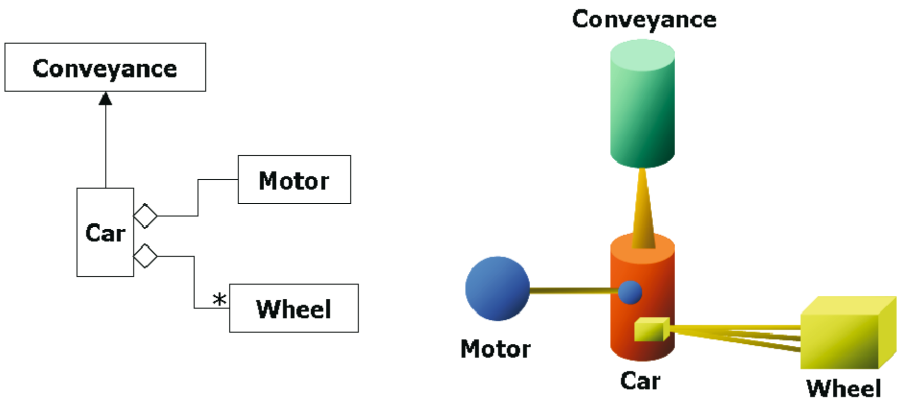


Fig. 3.28. UML versus geon diagrams

3.4.3 Software Metrics

You can't control what you can't measure!

[DeM82]

Not everything that counts can be counted and not everything that is counted counts.

(Albert Einstein)

The IEEE Standard 1061 [IEE98] defines a software quality metric as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." Many metrics[3] have been suggested to assess the quality of software and its development process.

Typical metrics with respect to the static structure of the program code of a software product are its size in terms of lines of code, the number of functions or classes, or its complexity in terms of graph-theoretical measures such as the classical and widely used metric is cyclomatic complexity of McCabe [McC76]. On the basis of the control-flow graph $G = (V, E)$ of a program, the cyclomatic complexity is $CC = |E| - |V| + 2|U|$, where U is the set of unconnected subgraphs of G.

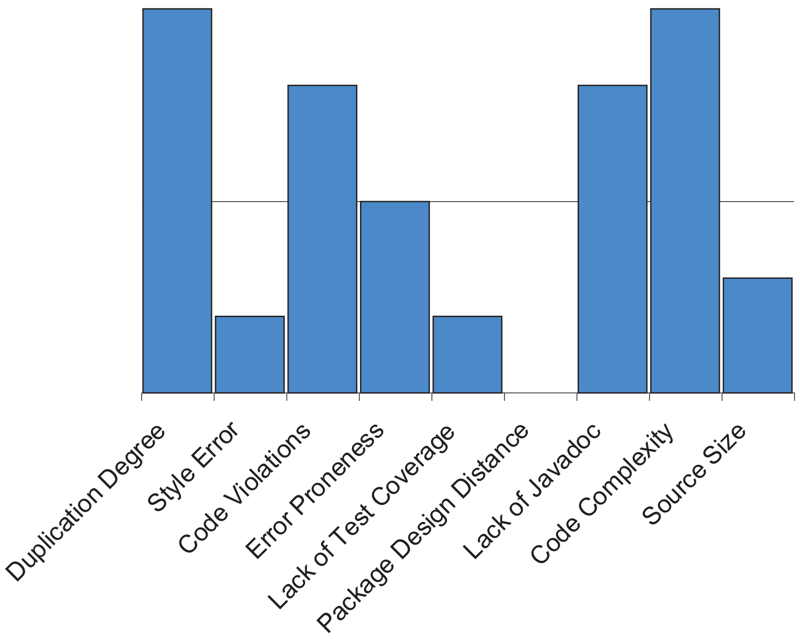


Fig. 3.29. Visualizing multiple software metrics as a bar chart

Other metrics, such as number of successful test runs and test coverage, address the dynamic behavior of the software. Figure 3.29 shows the bar chart for multiple software metrics computed for a software system. The spider diagram in Fig. 3.30 was produced for the same data with the open source Xradar tool [XRa]. Finally, metrics related to the software process measure the effort required in terms of staff, time, and costs, as well as the quality of the process in terms of reported and fixed bugs.

[3] In mathematical measurement theory, the term *metric* is used for a more restricted concept, namely distance functions.

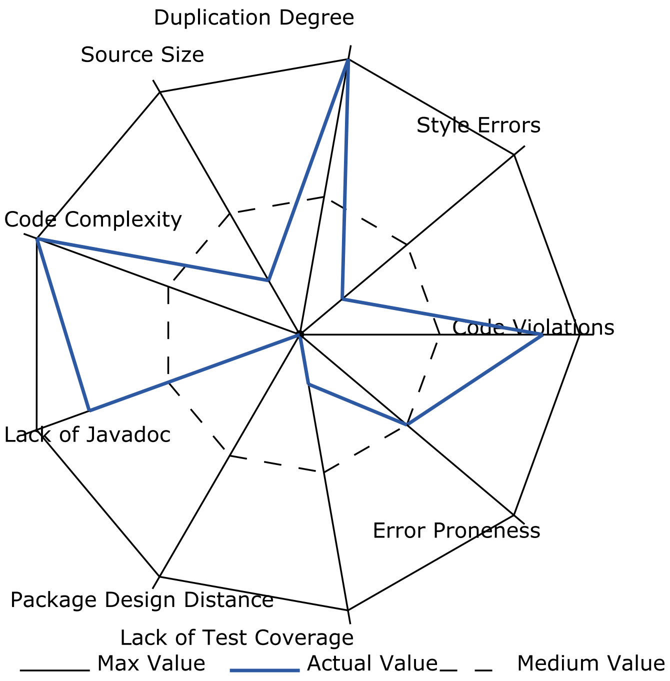


Fig. 3.30. Visualizing multiple software metrics as a spider diagram

Metrics differ not only in what they measure, but also in the scales against which measurement is performed. In general, there are two kinds of scales: categorial and quantitative. Whereas the former use symbolic values, also called categories, the latter use numeric values.

Nominal scales assign categories that can only be tested for equality. As an example of a two-valued nominal scale, consider { `tested`, `untested` }, which could be used to measure whether a function has been tested or not. If we have this metric for each function in a program, we can compute the frequency of each value, i.e. how many functions have been tested or not.

Ordinal scales have a total order on their categories, for example `untested` < `partially tested` < `completely tested`.

Interval scales assign numbers such that the distance between these numbers has a meaning. Complexity measures such as McCabe complexity are examples of interval scales. There is no zero value, and it does not make sense to say that a program is twice as complex as another one if its McCabe value is twice as large.

In contrast, in a ratio scale there has to be an absolute zero that is meaningful, for example the number of programmers that are trying to fix a bug. It makes sense to say that no programmer is working on the bug, or that twice as many programmers have been working on this bug during this month than during last month.

The choice and interpretation of metrics is full of problems [KB04, VW93, ZB89] that are beyond the scope of this book.

In the rest of this chapter, software metrics will be applied to one version of a software system. In Sect. 5.1, we shall also look at the change of software metrics over time.

3.4.4 Software Visualization and Reverse Engineering

> Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction.
>
> [CI90]

The goal of reverse engineering is to recover design abstractions from the implementation on the basis of the source code, the documentation, and domain knowledge. Many reverse-engineering tools present their results in graphical notations that were originally developed for the design of software systems before actually programming them. Thus, there are numerous tools that, given the source code of a program, generate UML class diagrams and some other UML diagram types, for example Borland Together, Rational Rose, ESS-Model, BlueJ, and Fujaba. Fujaba also detects design patterns and extends the UML diagrams by indicating what roles the various classes play in the pattern.

Eichelberger [Eic03] has compiled a list of 15 aesthetic criteria to be used for layout of UML class diagrams. He evaluated the automatic layout of 42 current computer-aided software engineering (CASE) tools [Eic02]. He concluded, that "the tools usually produce horrible results by transforming the layout and implicitly and accidentally changing the semantics of the complete diagram." While tools such as those listed above use mediocre layout algorithms based on hierarchical layout (for example, see Figure 3.31), recently two orthogonal layout algorithms using the shape-metrics approach have been developed [EGK$^+$04]. These algorithms yield better results for UML diagrams that have only a small number of inheritance edges but a large number of associations. Both algorithms have been leveraged by providing plug-ins to major commercial software design tools. Figure 3.32 shows a layout of the same diagram as in the previous example but was produced with one of these plug-ins.

Class Blueprints

Instead of producing UML diagrams from the source code of a system, one can also produce other kinds of diagrams. Class blueprints show the call graphs within classes and those between classes [LD01]. They are one of several visualizations offered by the CodeCrawler tool [Lan].

First of all, the methods of a class are grouped into four layers, and a fifth layer contains the attributes of the class. The initialization layer contains all

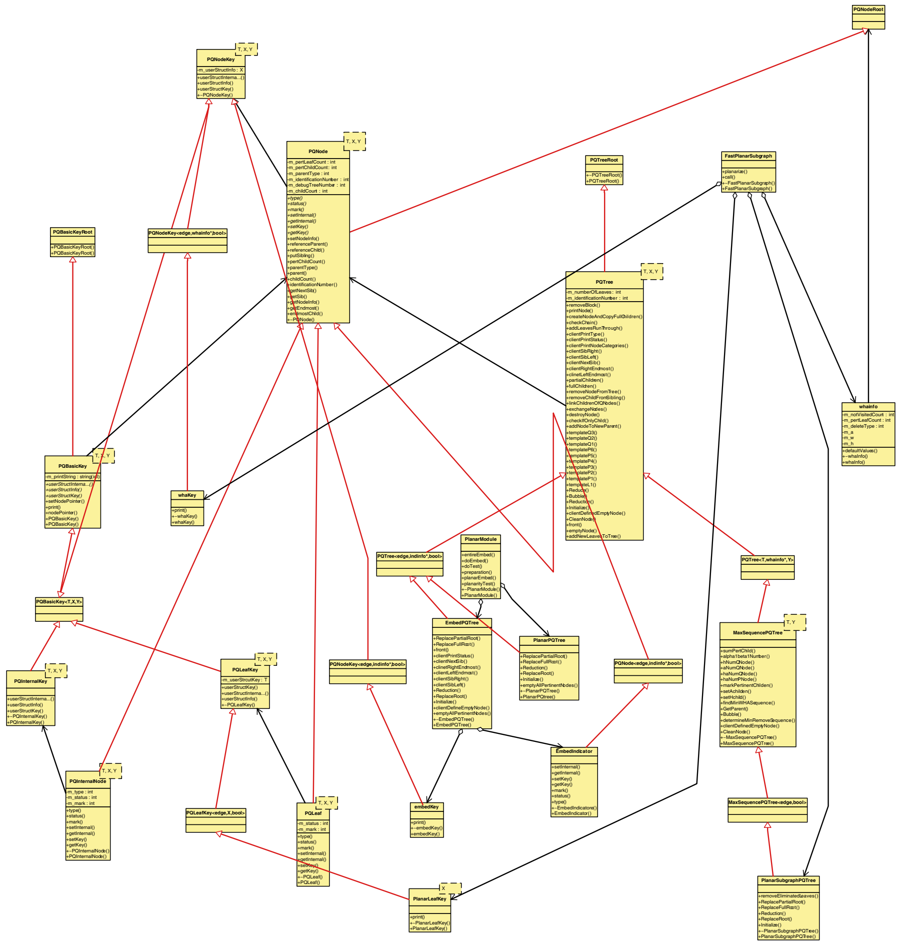


Fig. 3.31. Automatic layout produced by an industrial CASE tool (©2003 ACM)

methods with the substring `init` or `initialize`, and all constructor methods. The interface layer contains all methods invoked from the initialization layer, and all public methods and methods not invoked by other methods in the same class. The implementation layer contains all private methods and methods invoked by other methods of the same class. The accessor layer contains all methods to get and set values of attributes, and, finally, the attributes layer contains all attributes of the class

On the basis of these layers, the static call graph within a class is drawn as shown in Fig. 3.33. Boxes represent the methods, and the sizes of these boxes can represent some metrics, for example the number of lines. In addition, the following color-coding scheme is used: brown boxes represent overridden methods, yellow boxes represent delegating methods, cyan boxes represent

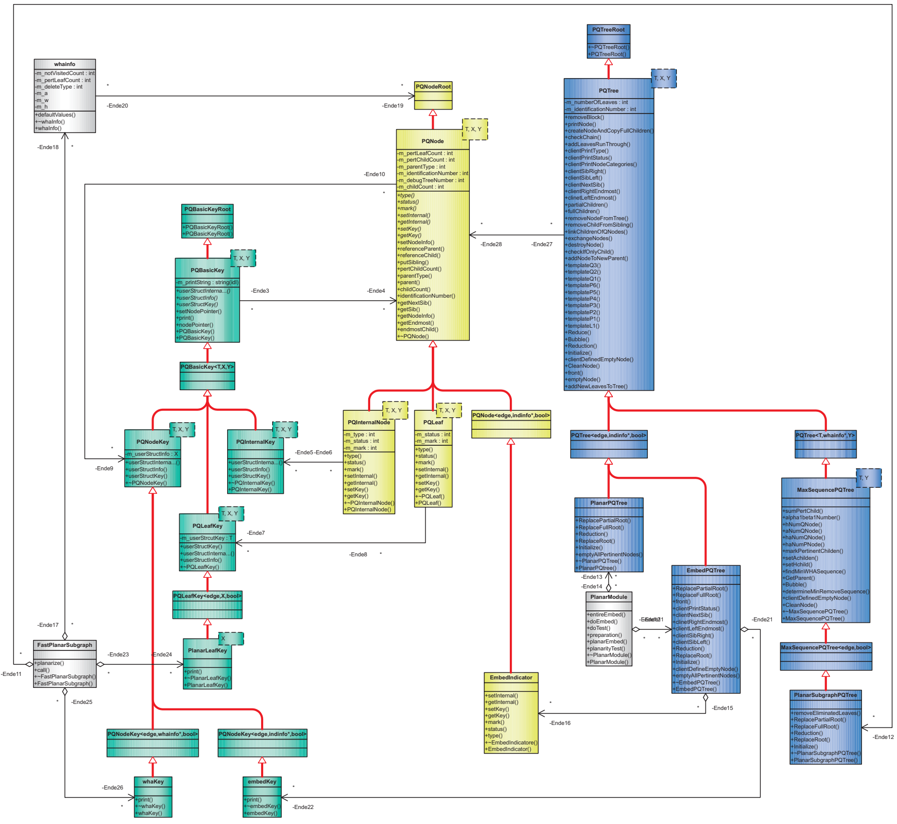


Fig. 3.32. Automatic layout produced by GoVisual (©2003 ACM) [GJK+03b]

abstract methods, and orange boxes represent extending methods. Black arrows indicate invocations of methods, while cyan arrows show invocations of accessor methods or direct access of an attribute

The blueprints of individual classes are then arranged in an inheritance tree and additional arrows show method invocations between classes (see Fig. 3.34).

On the basis of class blueprints, Lanza and Ducasse categorized classes delegator classes or as classes with a large implementation or a wide interface, for example. Taking inheritance into account, they also distinguished definers, overriders, and extenders. Definers are those classes that define mostly new methods, overriders are those that mostly override methods, and extenders are those that mostly extend methods.

In addition, overriders and extenders can be talking, which means that they invoke methods of a superclass, or mute, which means that they do not invoke methods of a superclass.

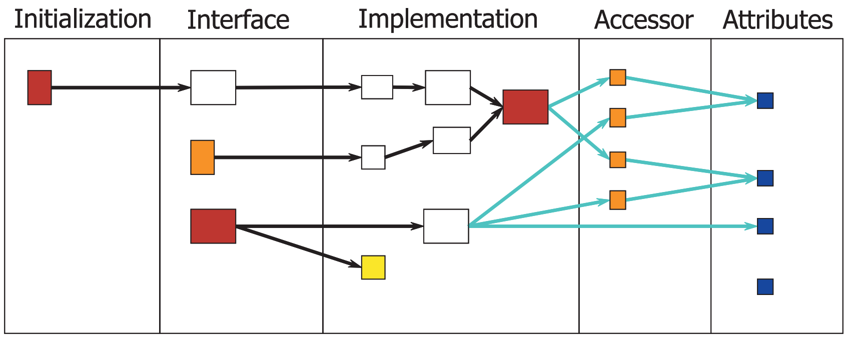


Fig. 3.33. Class blueprint of a single class

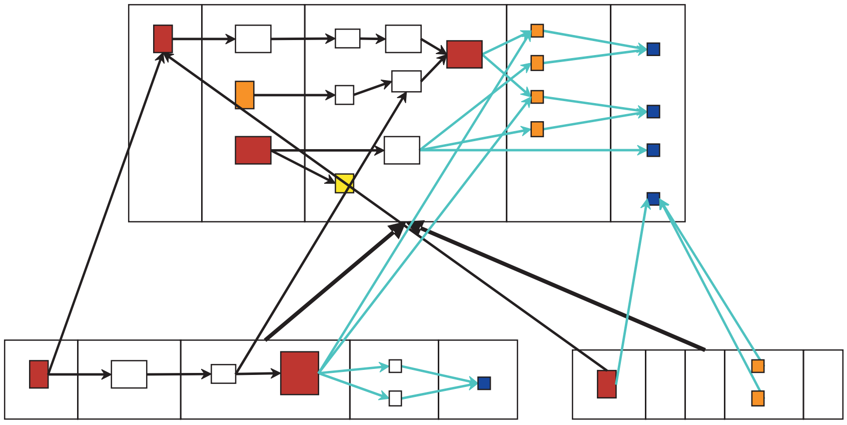


Fig. 3.34. Inheritance and class blueprints

Iterative Analysis and Aggregation

One of the first reverse-engineering tools that provided sophisticated visualizations was RIGI [MOTU93]. It basically parses the source code and builds an initial graph model (called the resource-flow graph). The user can explore this graph and semiautomatically identify subsystems.

This approach has motivated the design of many other visual reverse-engineering tools. Tools such as GOOSE [Goo], Sotograph [Sof], SHriMP [Chib], and VizzAnalyzer [PLL04] work on the class and method levels but the information can be aggregated to form higher levels of abstractions. Graphs, as well as metrics computed for these graphs, are stored in a repository. Graphs can be extracted from the repository and are typically drawn with force-based or energy-based layout methods, which lead to many edge crossings but help to detect clusters. The user can perform various analyses on the graphs, create new graphs, etc., and thus iteratively refine the model as shown in Fig. 3.35.

Their generality is both the strength and the weakness of the above-mentioned reverse-engineering tools as well as many others, because it makes them very difficult to use.

Aggregation is one of the main operations used to form graphs at higher levels of abstraction. Assume we have a graph $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times \mathcal{D} \times V$ is a set of edges, and where $\mathcal{D}$ is a set of labels which carry some information. In other words, if $(v_1, d, v_2) \in E$ then d is the information associated with the edge. The simplest form of aggregation is edge aggregation. Here we combine several edges into one. Assume we have an aggregation function $\alpha : \mathcal{P}(D) \longrightarrow D'$; we obtain an aggregated graph $G^\alpha = (V, E^\alpha)$, where

$$E^\alpha = \{(v, \alpha(\{d_1, \ldots, d_n\}), w) | (v, d_1, w), \ldots, (v, d_n, w) \in E\}$$

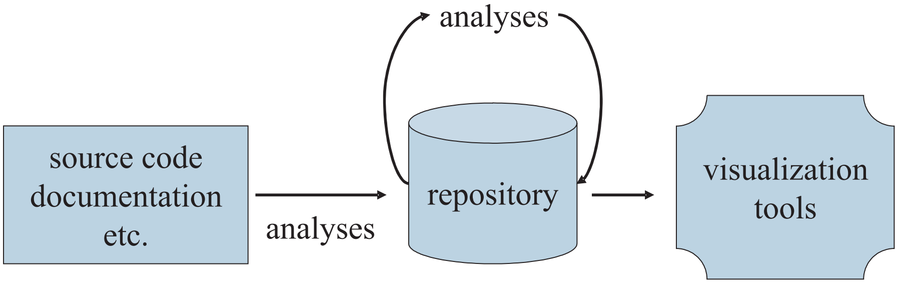


Fig. 3.35. Typical architecture of reverse-engineering tools

Often the information associated with the edges is a weight, i.e. some numeric value, and the aggregation function simply sums these values.

To form subsystems, one partitions the set of nodes and aggregates all edges between different partitions. Assume that $\pi : V \longrightarrow V'$ is a surjective function and $V' \subseteq \mathcal{P}(V)$ is a partition of V, i.e. all sets in V' are disjoint, and $\pi(v) = v' \in V'$ such that $v \in v'$. For simplicity we assume that there is at most one edge between two nodes. The partitioned, aggregated graph is then $G^\alpha_\pi = (V', E^\alpha_\pi)$, where

$$\begin{aligned} E^\alpha_\pi = \{ & (v', \alpha(\{d_1, \ldots, d_n\}), w') | v', w' \in V' \text{ and} \\ & (v_1, d_1, w_1), \ldots, (v_n, d_n, w_n) \in E \text{ where } \pi(v_i) = v' \text{ and } \pi(w_i) = w'\} \end{aligned}$$

Creole

As an example, we take a closer look at Creole [Chia], an open-source plug-in for the Eclipse interactive development environment (IDE) which leverages the SHriMP approach by integrating it into a widely used development environment. SHriMP (Simple Hierarchical Multi-Perspective) visualizes dependencies in hierarchically structured data as nested graphs. While SHriMP requires the results of analyses presented in specific graph exchange formats as input, with Creole, analyzing and visualizing dependencies in a Java package or a file is as simple as dragging it into the Creole window.

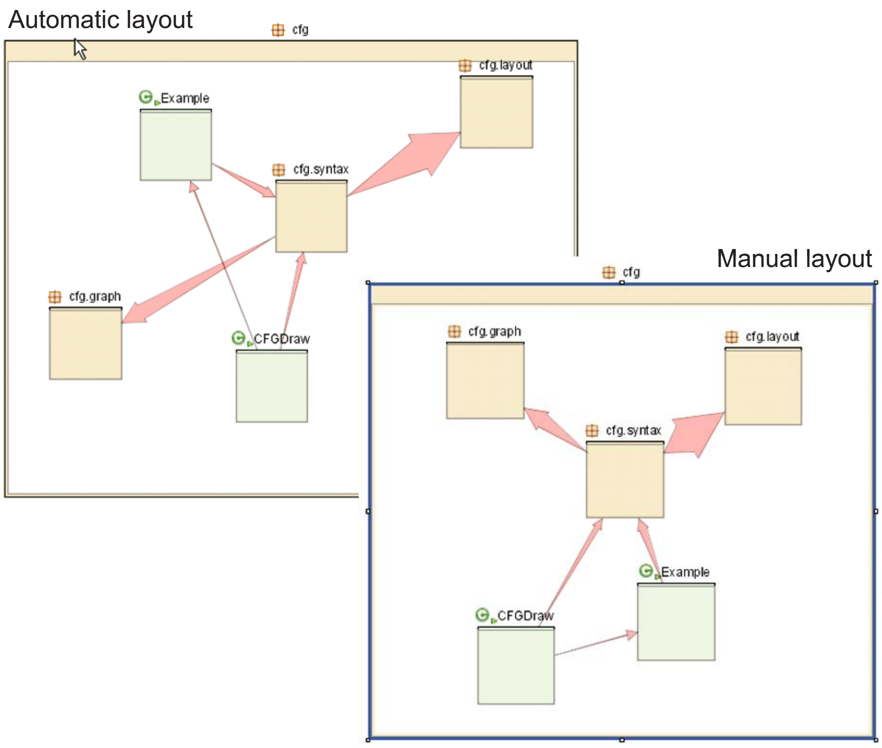


Fig. 3.36. Creole: package dependencies via method calls and field access

In the following, we describe the use of Creole to explore the structure of an implementation of the computation and layout technique for control-flow graphs presented in Sect. 3.2.2.

Assume that V_C is the set of classes (p, c), where p is the name of a package and c is the name of a class in p. The call graph $G_{call} = (V_C, E_{call})$ has edges

$$E_{call} = \{((p,c), 1, (p',c'))|\ \text{a method in class } c \text{ of package } p \text{ invokes}\\ \text{a method in class } c' \text{ of package } p'\}.$$

Analogously, the access graph $G_{access} = (V_C, E_{access})$ has edges

$$E_{access} = \{((p,c), 1, (p',c'))|\ \text{an expression in class } c \text{ of package } p \text{ accesses}\\ \text{a field in class } c' \text{ of package } p'\}.$$

We obtain an aggregated graph of $(V_C, E_{call} \cup E_{access})$ on the package level with the function $\pi((p, c)) = p$ and the aggregation function $\alpha(d_1, \ldots, d_n) = \sum d_i$.

Figure 3.36 shows the dependencies of the top-level package `cfg`, which contains three subpackages and two classes. Here the arrows represent the aggregated method calls and field accesses between classes in each package.

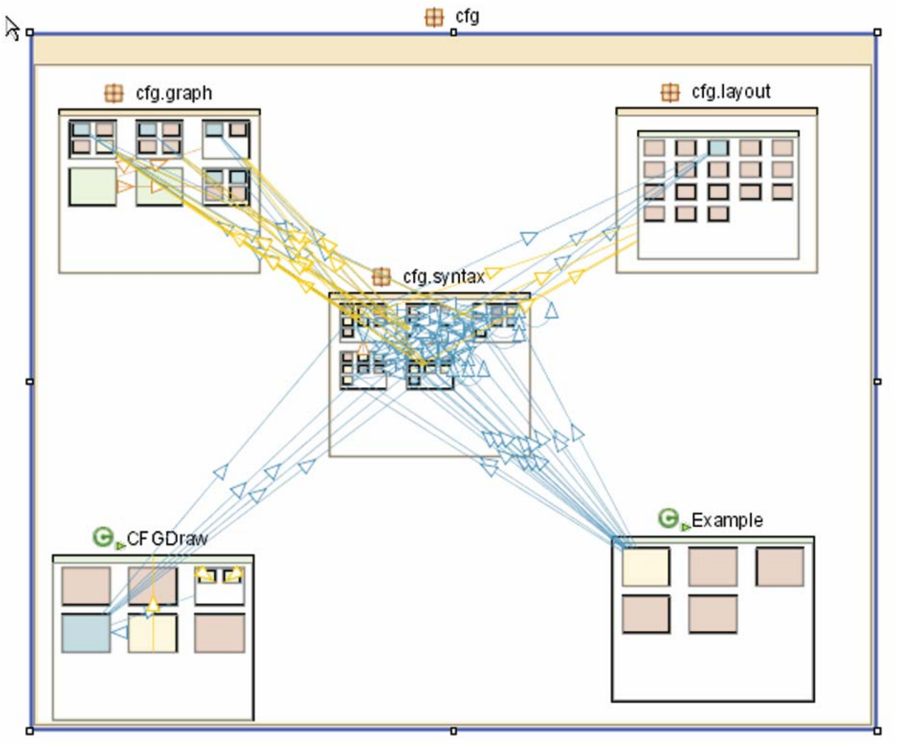


Fig. 3.37. Creole: method calls and object creation

Although the package diagram is based on method calls and field accesses, we can get a better impression of how the packages collaborate by looking at method calls and object creation as shown in Fig. 3.37. The two classes in the highest layer call methods of classes in the middle layer, which in turn call methods in the lowest layer, but, more importantly, create many objects of classes in the lowest layer, indicated by the yellow edges.

Creole allows one to interactively expand packages and classes and select the kinds of nodes and edges to be shown:

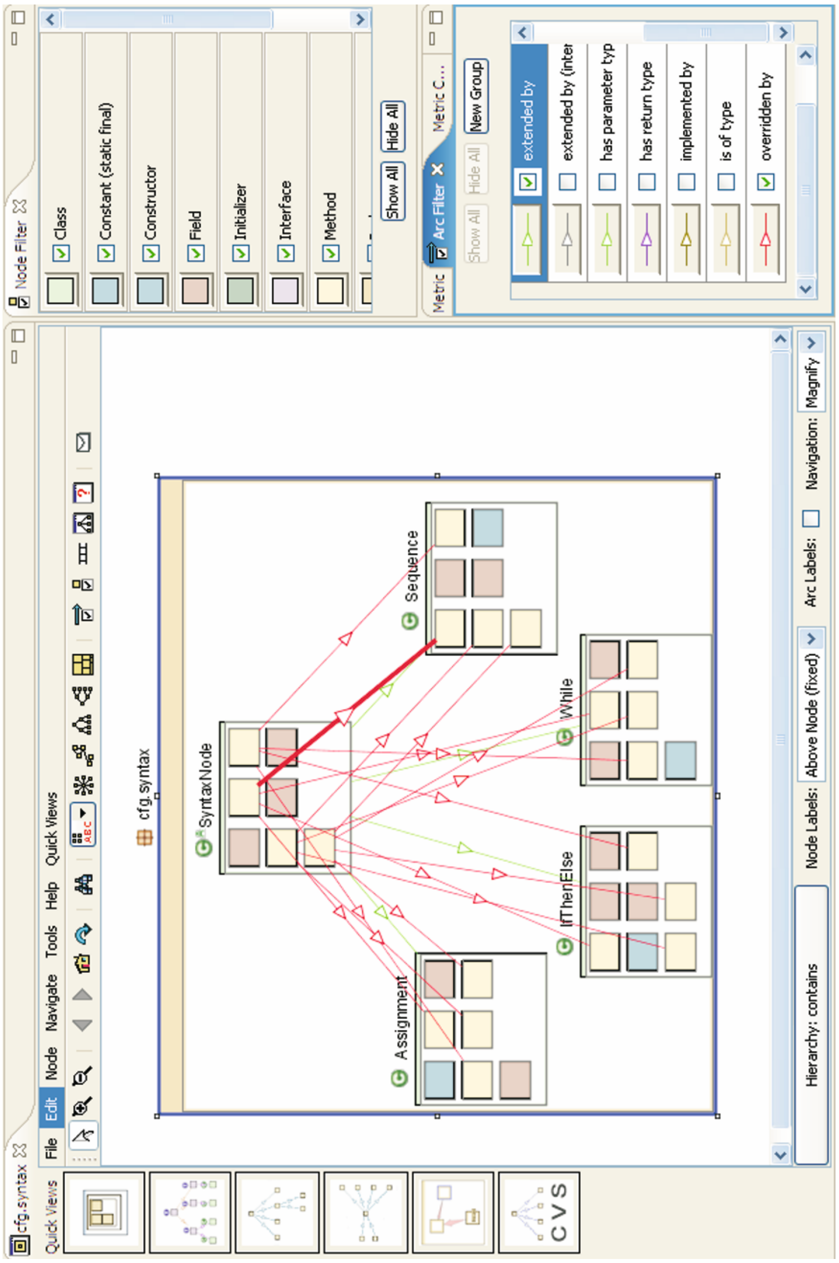


Fig. 3.38. Creole: overriding and extending methods

- A node represents one of the following software artifacts: a class, a constant (static final), a constructor, a field, an initializer, an interface, a method, a package, a package root, or a project.
- An edge represents a relation between two nodes. The kinds of relations depend on the kinds of nodes, i.e. the software artifacts represented by the nodes. The possible relations are: "accesses", "calls", "casts to type", "contains", "creates", "extended by", "extended by (interface)", "has parameter type", "has return type", "implemented by", "is of type", and "overridden by".

In Fig. 3.38 the green edges indicate which methods of class `SyntaxNode` are extended by other classes in the package `cfg.syntax`, while red edges indicate those which have been overridden.

3.4.5 3D and Software Architecture

Many arguments have been put forward speculating that 3D visualizations are superior to two-dimensional ones,[4] including those that we will discuss later in Sect. 4.4.5. Ware and Franck [WF94] have argued that the amount of information that can be perceived in a three-dimensional display exceeds that in a two-dimensional display by a factor of three. Motivated by results such as this and the advance of technology, many researchers have suggested 3D visualizations of software architectures and even the use of virtual environments to allow one to literally walk through these architectures.

Gogolla et al. suggested several scenarios where UML diagrams can benefit from a three-dimensional layout [GRR99, RG00]. They illustrated these scenarios by 3D UML diagrams that they implemented using the Virtual Reality Modeling Language (VRML):

- In a class diagram, important classes are drawn in the foreground and thus have the focus. Moreover, one can have several different perspective views of the same diagram. In each of the perspectives the focus is on different classes, and when the user changes the perspective the nodes of the diagram are moved by smooth animation to their new positions.
- In object diagrams, various shapes can be used to represent objects. Objects of the same class have similar shapes.
- For sequence diagrams, animations show messages represented by small balls that move from the sender to the receiver. The problem of overlapping arrows for simultaneously sent messages and of arrows crossing lifelines can also be solved by 3D layout. Unfortunately, projecting these 3D visualizations onto 2D computer screens leads to other kinds of overlaps and crossings. Figure 3.39 shows a 3D version of the sequence diagram in Fig. 3.27.

[4] Some of these have been empirically invalidated. For example, recent studies show that perspective effects in monocular static computer displays do not significantly influence the effectiveness of spatial memory [CM01, CM02].

- Several diagrams can be combined in space. For example, a class diagram may be shown in the background and a related sequence diagram in the foreground.

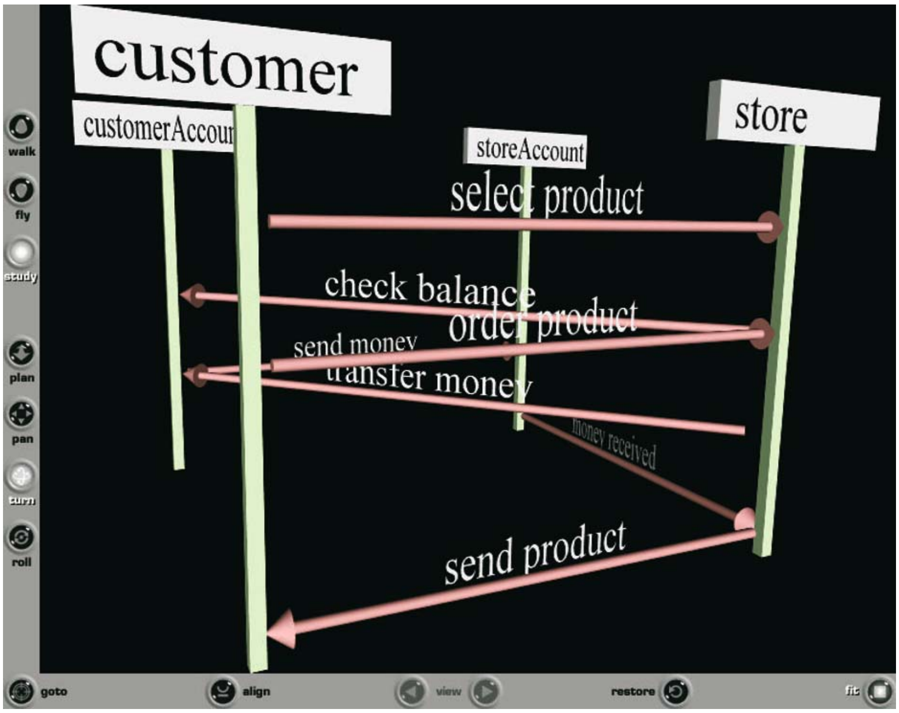


Fig. 3.39. Three-dimensional sequence diagram

X3D-UML is an approach to automatically producing 3D UML diagrams from source code [MHvS05]. First the source code is converted into an XML representation, and then XSLT scripts extract the relevant information from this representation and produce X3D models (X3D stands for “extensible 3D”, the XML-based successor to VRML).

Imsovision [MLMD01] is a system that shows a stereoscopic 3D visualization of classes, their properties, and their dependencies and aggregations in a virtual-reality environment, namely a cave. As the cave is a room where the visualization is rear-projected onto its walls, the user can enter the room and interact with the 3D scene. In Imsovision, classes are represented by platforms, methods by columns, and attributes by spheres put on top of the platforms. The platforms of subclasses are placed next to their superclasses. Dependencies and aggregations are shown as flat edges. In addition, various properties are represented by colors.

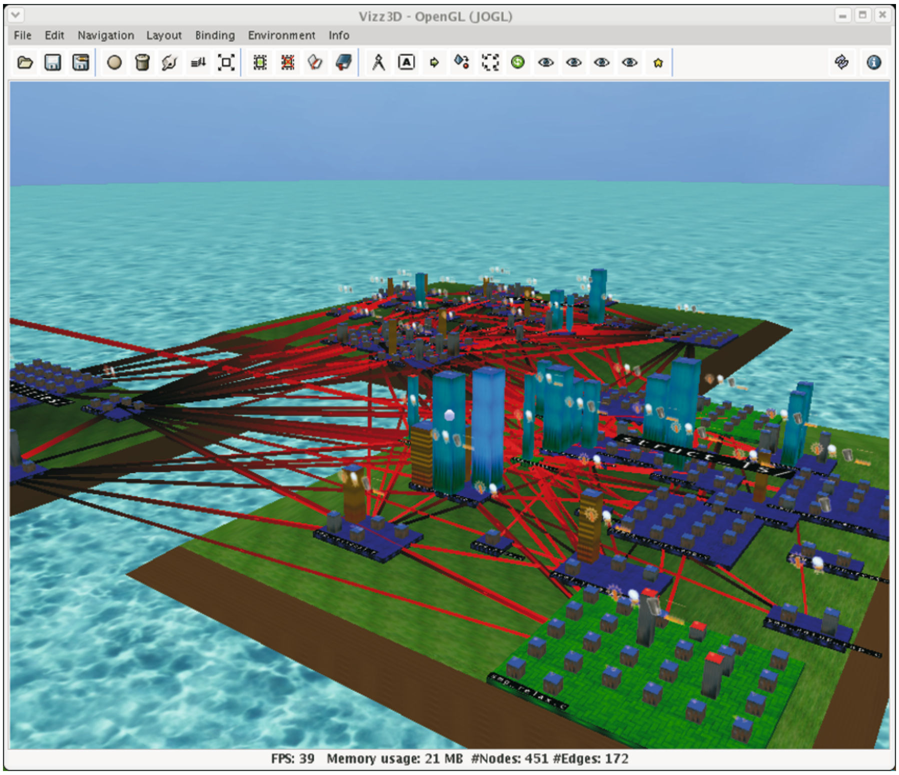

Vizz3D - OpenGL (JOGL)
File Edit Navigation Layout Binding Environment Info
FPS: 39 Memory usage: 21 MB #Nodes: 451 #Edges: 172

Fig. 3.40. 3D architecture visualization produced with Vizz3D

A very similar approach, called Software Landscapes was recently developed by Balzer et al. [BNDL04]. This representation uses more realistic metaphors, such as islands instead of platforms and skyscrapers instead of simple columns.

In the Software World [KM99] visualization, the world represents the software system as a whole, a country represents a package, a city represents a file, a district within a city represents a class, and a building represents a method. The size of a building indicates the number of lines of code, the doors represent its parameters, the windows represent the variables declared in the method, and the color of the building indicates whether the method is private or public.

Panas et al. have suggested that dynamic information should also be included in such a visualization [PBG03], for example hot-execution areas can be surrounded by fire, two-directional calls can be represented by streets, and unidirectional calls can be represented by water (rivers). Information is sent via boats or cars between classes or packages.

Figure 3.40 shows the first steps towards automatically generating such a visualization using the open-source tool Vizz3D [Viz]. Houses represent functions, and edges indicate function calls. The size of a house corresponds to the number of lines of code. Similarly to information pyramids (see Sect. 2.3.3), the platforms represent directories or files containing functions. Icons show additional information, for example a garbage can indicates dead code, a wheel indicates access to global variables, and a lock indicates security issues. Flames used as icons and as textures of houses indicate that the McCabe's cyclomatic complexity exceeds a threshold.

3.5 Summary

Graphical representations of program code have been used since the early days of computer science. Whereas Nassi–Shneiderman diagrams encode relations between program parts by containment, all other representations described in this chapter do so by linkage, i.e. the program parts are represented by nodes and the relations by edges between these nodes.

Except for control-structure diagrams, no graphical representation preserves the linear order of the underlying source code, and this makes it hard for the programmer to map graphical elements to parts of the program code. Fortunately, the sequential order in the code becomes less important for representations at higher levels of abstraction such as call graphs. Whatever the graphical representation of the program structure is, it can be enriched using information from static program analyses and thus help to debug or improve the program code.

While UML is widely used for the design of software architectures, nonstandard graph-based notations prevail for recovering various aspects of the software architecture of a given system. Automatic and semiautomatic aggregation of nodes and edges in these graphs is an important technique for forming architecture diagrams at higher levels of abstraction.

Recently, three-dimensional visualizations of software architectures have taken real-world metaphors literally. The resulting visualizations are very appealing and may help in the future to convey architectural information to nonexperts.

Exercises

Exercise 1: Software is big! Assume that a line of code (LOC) has a height of 0.5 cm when printed out on a sheet of paper. How many kilometers would the printed program code of Windows 3.1 (3 million LOC, 1990), Windows XP (50 million LOC, 2002) and Debian Linux 3.1 (210 million LOC, 2005) cover?

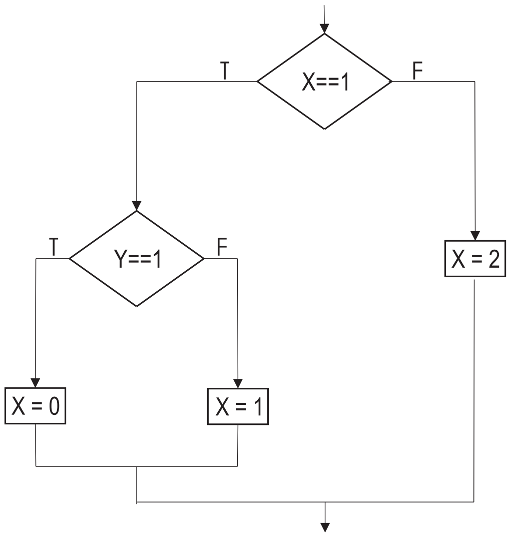


Fig. 3.41. Control-flow graph for Exercise 2

Exercise 2: Write a program for which we would get the control-flow graph shown in Fig. 3.41.

Exercise 3: Draw the control-flow graph and the Nassi–Shneiderman diagram of the following program:

```
x=0;
odd=0;
even=0;
while (x<100)
  { if (x%2==0)
      even=even+1;
    else
      odd=odd+1;
    x=x+1; }
```

Exercise 4: Give all computation rules (informal and formal) for `cfg()` and `box()` of the `do{`S`}while(`E`)` statement.

Exercise 5: Compute the live variables for the control-flow graph in Fig. 3.20 and annotate the nodes of the graph.

Exercise 6: Search for architecture diagrams on the Web and identify familiar architectures therein.

Exercise 7: Below you will find excerpts from the Java source code of a program that controls a pet door. The pet door detects animals wearing a collar key. The collar key electronically transmits an ID. The pet door opens and closes automatically for registered pets. For this program draw three different diagrams:

- Draw an architecture diagram using icons and metaphors related to the application. You can draw the diagram by hand, but you can also produce it with your computer.
- Draw the UML class diagram with aggregations.
- Draw a class blueprint (no color coding required).

```
public class Pet
{ int collarKey;  String name; }

public class Pets
{ final int maxPets=10;
  Pet[] list = new Pet[maxPets];
  public boolean contains(int k) { ... }
  public Pet get(int k) { ... }
  public void add(Pet p) { ... }
  public void remove(int k) { ... }
}

public class Door
{ boolean isOpen;
  public void isOpen() { return isOpen; }
  public void open() { isOpen=true; }
  public void close() { isOpen=false; }
}

public class PetDoor extends Door
{ Pets currentPets, registeredPets;
  PetDoor(Pets regPets)
   { registeredPets=regPets;
     currentPets=new Pets();
     isOpen=false; }

  public void open() { if (!isOpen) { super.open(); } }

  public void collarKeySignalReceived(int k)
   { if (registeredPets.contains(k))
       { Pet p=registeredPets.get(k);
         open();
         currentPets.add(p); } }

  public void collarKeySignalLost(int k)
   { if (currentPets.contains(k))
      { currentPets.remove(k);
        close(); } }
}
```

Exercise 8: Apply the Creole tool to one of your own software projects. Try to arrange the packages in the package-level diagram in a meaningful way. Expand a package and look at the inheritance, overriding, and extending edges. Can you obtain a visualization similar to the class blueprint visualization by choosing the appropriate options in the `Node Filter`?

4 Dynamic Program Visualization

In the previous chapter the visualization of the structure of software at various levels of abstraction was discussed. Now we shall look at the behavior of programs, i.e. what actually happens at run time. Dynamic program visualization shows the behavior of a program for a given input, i.e. what instructions are executed and how the program state changes (see Fig. 4.1).

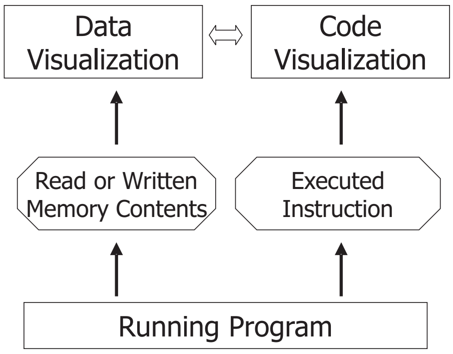


Fig. 4.1. Dynamic program visualization

Ideally, data and code visualization are combined such that the user sees how the execution of an instruction reads and modifies the memory.

As examples of dynamic program visualization, we shall look in more detail at dynamic architecture visualization, algorithm animation, and visual debugging and testing.

4.1 Dynamic Data Acquisition

There are many different ways to collect data during the execution of a program. Dynamic data acquisition usually slows down the execution of the program and might even change its behavior. So, the choice of the acquisition method depends on its invasiveness and on the kind of data that has to be collected.

4.1.1 How Is Runtime Data Collected?

Most often the program is instrumented, i.e. additional instructions are added to the program code. Instrumentation can be done on the level of both the source code and the machine code. Instrumentation code can be added before or after each instruction of the original code, at the start and end of each loop or each iteration of the loop, at each method call, at the entry and exit of each method, or at certain program points explicitly specified by the user. In simple cases, instrumentation code can be inserted by hand, but for real programs instrumenting the code should be done automatically.

To capture when and how data is accessed and changed, data structures can be replaced by new ones that capture changes or invoke external routines (demons) whenever the data structure is accessed. Instead of changing a program, one can also have a parallel thread or process that looks for changes in the program memory. For distributed programs, messages can be recorded. Finally, the behavior can also be observed by running it on a virtual machine or interpreter.

A serious problem, at least for embedded systems or distributed programs, is that instrumentation and also all other approaches to collecting data are invasive, i.e. they actually change the timing of the execution. Thus, for example, deadlocks might not occur in the instrumented program but might occur in the original one.

4.1.2 What Runtime Data Is Collected?

For code visualization, we need the program position; this might be the actual line of code, if we use an interpreter, or the value of the program counter or the address of an invoked method, if we capture at the machine level. For compiled programs, the mapping of addresses in the compiled program to lines of the source code is often a problem. Some compilers allow one to store additional information for debugging in the compiled program, but if the compiler does optimizations such as sharing code for available expressions, an instruction in the compiled code can actually correspond to several different parts of the source code.

For data visualization, typically the values of program variables and the parameters of method calls or, at a lower level, the values of registers are recorded. For some purposes parts or even the whole contents of the program heap might be extracted, as discussed in Sect. 4.5.3. For distributed programs, not only the messages but also the local time when they were sent and received at the various computers should be recorded to allow one to reconstruct the causal order of the messages.

Note that not only the amount of instrumentation code but also the amount of data that is captured, and how it is stored, increase the run time of the program.

4.1.3 Dynamic Data Acquisition in Java

To illustrate some of the above-mentioned approaches, we look at various ways to gather runtime information in Java.

Enclosing Method Calls If the source code is available, one can enclose instructions in additional code to notify a tracing object when the instruction is executed. For example method calls can be modified in order to record when the method is called and when its execution is finished:

```
trace.beforeMethodCall("withdraw(int amount)");
account.withdraw(100);
trace.afterMethodCall("withdraw(int amount)");
```

This approach is very flexible, as one can enclose different kinds of instructions, and not every occurrence of an instruction need be enclosed. The instrumentation of the source code can be automated by using aspect-oriented programming tools such as AspectJ or InjectJ.

Instrumenting Method Bodies To trace all method calls of a particular method, the above method would require one to modify all method calls, which would lead to a large increase in code size and is only possible if the source code of all classes that contain calls to the method is available. Instead of modifying the method calls, we can also change the method body by adding instrumentation code at the start of the method body and before each return statement in the method body:

```
void withdraw(int amount)
{ trace.startOfMethod("withdraw(int amount)");
  balance=balance-amount;
  trace.beforeReturnFromMethod("withdraw(int amount)");
  return; }
```

Method Stubs As the body of a method can be very complex or the source code might not be available for transformation, one can also create method stubs that forward the method call to the original method, and all calls to the original method in the available source code have to be replaced by calls to the method stub.

```
void trace_withdraw(int amount)
{ trace.startOfMethod("withdraw(int amount)");
  withdraw(amount)
  trace.beforeReturnFromMethod("withdraw(int amount)");
  return; }
```

Byte Code Injection Instead of transforming the source code, the instrumentations discussed above can be added to the byte code using Java APIs such as JikesBT [IBMa] or BCEL [Fou]. This approach is useful in particular when the source code is not available, which is often the case for third-party libraries.

Extended JVMs There have been various extensions of the Java Virtual Machine (JVM) that allow one to access runtime information without changing the application program. In Java 1.5, the Java Virtual Machine Tool Interface (JVMTI)[1] was introduced to inspect the state and control the execution of an application at run time. For example, it allows a client program to register with the JVM to receive events whenever the JVM is initialized or shut down, objects are allocated or released, methods are called or returned from, threads are started or stopped, classes are loaded or unloaded, garbage collection is started or finished, monitors of synchronized methods or objects are entered or exited, or a thread is waiting to enter a monitor.

4.2 Visualizing Dynamics

Below, we discuss three general approaches to visualizing changes of information over time: accumulation, spatial projection, and animation. These approaches apply to both dynamic data visualization and dynamic code visualization.

4.2.1 Fundamental Techniques

First, information can be accumulated over time. For example, dynamic data visualization may simply show the average value of a variable, or dynamic code visualization may show just the number of times an instruction has been executed. Reducing the changes to a single value seems like a bad choice as we lose much information, but it is very useful if we want to get an overview of the changes of many items.

If we focus on a few items, we can plot their values along a time axis, i.e. we use a spatial representation of temporal information. In the resulting diagram, we can look for trends and phases during the time period.

Instead of mapping time onto an axis in 2D or 3D space, animated computer graphics enable us to represent time by time. An animation is a sequence of images which are shown one after another. Each image represents the program state at some point during the execution. One problem with animations is that, at each moment, we see only a single image and have to to rely on our memory to remember what happened before. For better inspection and for

[1] This provides most of the features of the JVM Profiler Interface (JVMPI) and the JVM Debug Interface (JVMDI) and is meant to replace these.

use in textbooks, the images of an animation can be shown next to each other as in a comic strip. Next we shall illustrate the three approaches by means of an example.

4.2.2 A First Example

As an example, consider the following program, which does a simple simulation of annealing by computing the value of each cell in a matrix from the average of the value of the cell and the values of its neighboring cells:

```
int m[][]= new int[5][5], m2[][] = new int[5][5];
int old[][]=m, new[][]=m2;
m[2][2]=40; // put a high value in the middle of the matrix
            // and all other cells have the initial value 0

for (int t=0;t<4; t++)
 { for(int i=0; i<m.length; i++)
    { for(int j=0; j<m[i].length; j++)
       { int s=old[i][j];
         int neighbors=1;
         if (i>0) { s=s+old[i-1][j]; neighbors++;}
         if (i<old.length-1) { s=s+old[i+1][j]; neighbors++;}
         if (j>0) { s=s+old[i][j-1]; neighbors++;}
         if (j<old[i].length-1)
            { s=s+old[i][j+1]; neighbors++;}
         if ((i>0)&&(j>0)) { s=s+old[i-1][j-1]; neighbors++;}
         if ((i>0)&&(j<old[i].length-1))
            { s=s+old[i-1][j+1]; neighbors++; }
         if ((i<old.length-1)&&(j>0))
            { s=s+old[i+1][j-1]; neighbors++;}
         if ((i<old.length-1)&&(j<old[i].length-1))
            { s=s+old[i+1][j+1]; neighbors++;}
         new[i][j]=(int) Math.round(s/(double)neighbors);
       }
    }
 }
```

A code visualization using accumulation could indicate simply that the bodies of the first four `if` statements are executed 80 times, and those of the remaining four `if` statements only 64 times. Table 4.1 shows a data visualization: the values of the matrix are shown after each iteration. The last entry shows the result of accumulating all matrix values.

In Fig. 4.2 the values of three matrix elements are plotted along a time axis. For the middle element `a[2][2]` the value drops dramatically after the first iteration: for the outermost element `a[0][0]`, the value slowly increases. The values of all three elements converge to 2.

Table 4.1. Values of the matrix during the execution

Initial Value	First Iteration	Second Iteration	Third Iteration	Fourth Iteration	Average
0 0 0 0 0	0 0 0 0 0	1 1 2 1 1	1 2 2 2 1	2 2 2 2 2	1 1 1 1 1
0 0 0 0 0	0 4 4 4 0	1 2 3 2 1	2 2 2 2 2	2 2 2 2 2	1 2 2 2 1
0 0 40 0 0	0 4 4 4 0	2 3 4 3 2	2 2 3 2 2	2 2 2 2 2	1 2 11 2 1
0 0 0 0 0	0 4 4 4 0	1 2 3 2 1	2 2 2 2 2	2 2 2 2 2	1 2 2 2 1
0 0 0 0 0	0 0 0 0 0	1 1 2 1 1	1 2 2 2 1	2 2 2 2 2	1 1 1 1 1

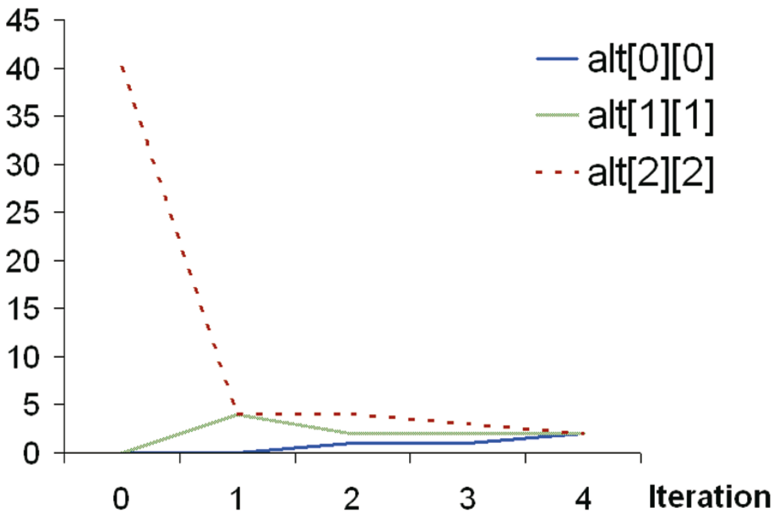


Fig. 4.2. Values of three matrix elements along a time axis

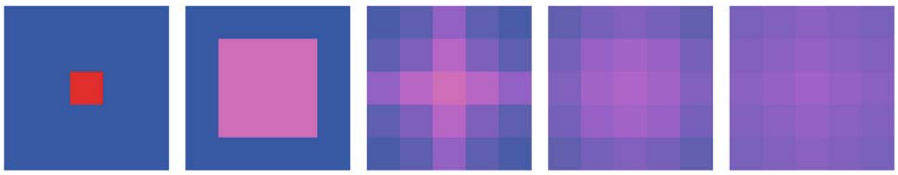


Fig. 4.3. Animation as a comic strip

Finally, Fig. 4.3 shows the sequential images in an animation. The animation was produced using floating-point numbers instead of integers for better precision, and color-coding to represent the values of the matrix elements.

4.3 Dynamic Architecture Visualization

The actual behavior of a running software system can be visualized at the level of its architecture at least in two ways. First, the architecture diagrams used for its design can be augmented with run time information. Second, on the basis of run time information behavior diagrams can be computed.

4.3.1 Augmenting Static Diagrams

SoftArch is a visual design tool for software architectures [GH03]. From the architecture diagrams drawn by the software architect, Java classes are generated, which can be used as a starting point for the implementation. The classes generated are automatically instrumented such that they allow one to capture particular method calls and events at run time. This runtime information can be used to analyze the actual implementation of the architecture. SoftArch aggregates this information and shows it at the level of the design elements of the architecture. While the implementation runs, SoftArch continues to capture information and adapts the visualization. It thus extends the static architecture visualization with dynamic information.

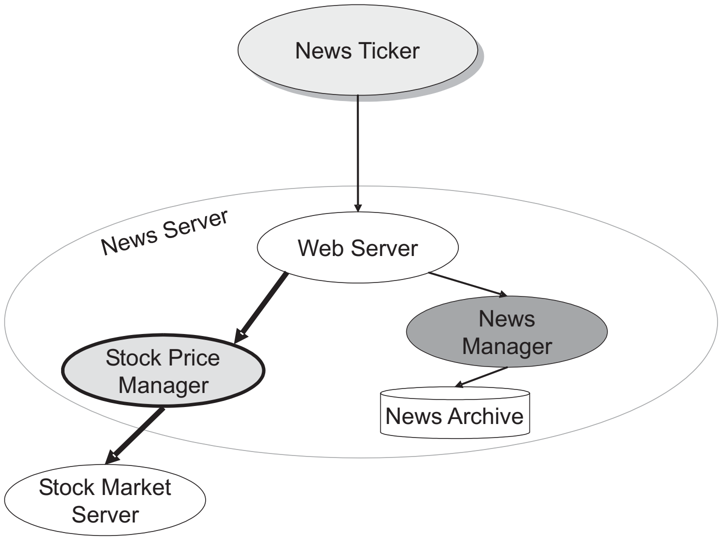


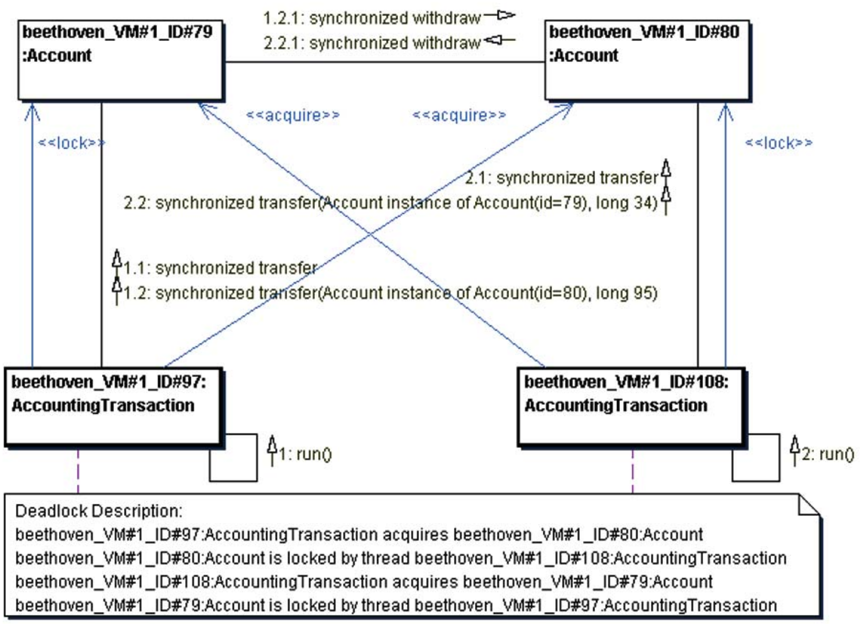
Fig. 4.4. Dynamic visualization of software architecture

Figure 4.4 shows a dynamic software architecture diagram similar to ones produced by SoftArch. In this example, the dynamic information visualized is the relative number of method calls and the event propagation. The thickness of the arrows indicates the amount of communication between components, the thickness of the border of a component indicates the amount of incoming communication, and the background color indicates the amount of internal communication.

4.3.2 Generating Behavior Diagrams

The SHIMBA [SKM01] environment for dynamic architecture visualization combines and extends the RIGI tool [MOTU93] for static visualization and the SCED tool [KST98] for dynamic visualization. SHIMBA uses information from the static visualization for focusing and abstraction in the dynamic visualization as follows.

In the static visualization, the user selects software artifacts such as classes and methods of the subject system. Then the subject system is executed, and runtime information is captured for the selected artifacts. On the basis of this runtime information, SHIMBA produces sequence diagrams. On the basis of the abstractions in the static visualization, i.e. the grouping of software artifacts into higher-level components, SHIMBA can also abstract the sequence diagrams and thus show the interaction among higher-level components.


1.2.1: synchronized withdraw
2.2.1: synchronized withdraw
beethoven_VM#1_ID#79 :Account
beethoven_VM#1_ID#80 :Account
<<acquire>>
<<acquire>>
<<lock>>
<<lock>>
2.1: synchronized transfer
2.2: synchronized transfer(Account instance of Account(id=79), long 34)
1.1: synchronized transfer
1.2: synchronized transfer(Account instance of Account(id=80), long 95)
beethoven_VM#1_ID#97: AccountingTransaction
beethoven_VM#1_ID#108: AccountingTransaction
1: run()
2: run()

Deadlock Description:
beethoven_VM#1_ID#97:AccountingTransaction acquires beethoven_VM#1_ID#80:Account
beethoven_VM#1_ID#80:Account is locked by thread beethoven_VM#1_ID#108:AccountingTransaction
beethoven_VM#1_ID#108:AccountingTransaction acquires beethoven_VM#1_ID#79:Account
beethoven_VM#1_ID#79:Account is locked by thread beethoven_VM#1_ID#97:AccountingTransaction

Fig. 4.5. JavaVis: collaboration diagram showing a deadlock situation

JaVis [Meh02] uses the Java Debug Interface (JDI) to trace the execution of concurrent programs and to detect deadlock situations. It then accesses the Together UML tool through an API to draw sequence and collaboration diagrams of deadlocks. Figure 4.5 shows such a collaboration diagram.

4.4 Algorithm Animation

In this section, we look at algorithm animation as an example of dynamic program visualization. As mentioned in Sect. 1.5, many authors characterize algorithm animation as being at a higher level of abstraction than program visualization. This distinction is very blurry. Most of the systems presented in this section have been designed to visualize single algorithms, instead of complete programs, and they have been developed mainly for educational purposes.

In this section we shall give a short history of algorithm animation, look at some examples, and then discuss some design issues. Next, some of the principal architectures of algorithm animation systems are characterized. Finally, we discuss an approach to the visualization of the abstract execution of algorithms.

4.4.1 What Is It About?

Algorithm animation is the visualization of the behavior of an algorithm. The term "animation" stems from the verb "to animate", which means "to bring to life". We refrain from defining the term "algorithm"[2] here. There are just too many formal and informal definitions, and we leave the dispute to others ([Mos01, Gur00]). No doubt the Church-Turing thesis, which states that every algorithm can be computed by a Turing machine [Tur36], constitutes a fundamental insight in computer science, but we certainly do not want to visualize the execution of all kinds of algorithms on Turing machines. Rather, we prefer computational models that are closer to the problem to be solved.

In all cases the execution of an algorithm by a real or mathematical machine leads to a sequence of states. Each step of the algorithm results in a transition from one state to another. Algorithm animations map each state into a visual representation (image) and usually show the transitions as animations between these images (see Fig. 4.6).

If we take a closer look at the implementation of algorithm animation systems, we often find that there is an intermediate layer. The state is mapped onto visual models, i.e. graphical objects or geometric data, which are then rendered to produce the images (see Fig. 4.7). Using this intermediate layer,

[2] The term "algorithm" is named after Abu Ja'far Muhammad ibn Musa Al-Khwarizmi, who wrote around the year 840 a treatise on algebra and a treatise on arithmetical calculation with Hindu–Arabic numerals. The Arabic text is lost but a Latin translation, *Algoritmi de numero Indorum* (Al-Khwarizmi on the Hindu numerals), gave rise to the word "algorithm" derived from his name in the title.

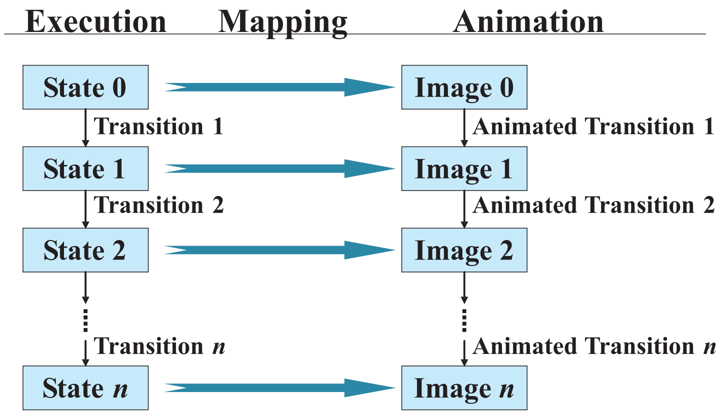


Fig. 4.6. Algorithm animation: mapping states to images

animations can be performed on the image level as before, but the real advantage of this approach is that the animations can be performed by continuous transformation of a model to a subsequent model. The challenge of algorithm animation is to find the right models, i.e. appropriate graphical abstractions for states and transitions between states.

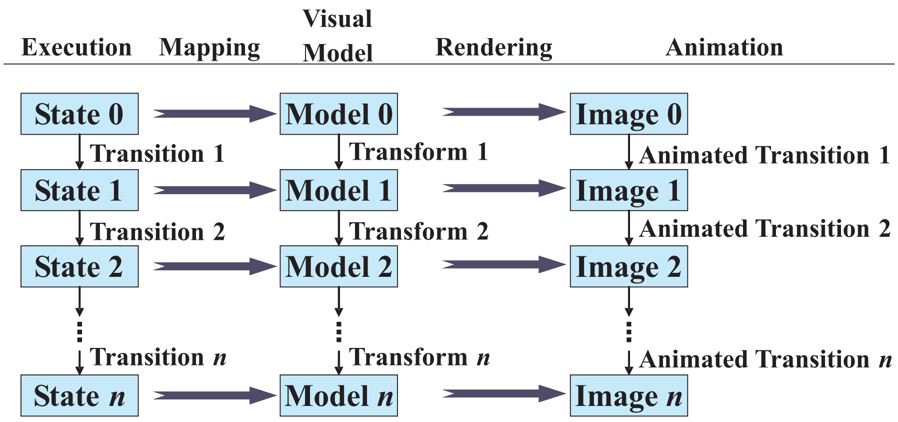


Fig. 4.7. Algorithm animation: mapping states to models

4.4.2 Why Do People Animate Algorithms?

Different people have different motivations for animating algorithms. These motivations also place different requirements on the animations and the way they are produced.

Understanding and teaching: Teachers visualize algorithms to explain them to their students.
Design: Developers visualize algorithms to better communicate the ideas to other experts.
Optimization: Developers visualize algorithms to better understand how they work and find possibilities to enhance them.
Debugging: Programmers use visualizations to find faults in their programs.

4.4.3 A Short History of Algorithm Animation

Allegedly, the first algorithm animation ever produced was a movie about list processing with the language L6 [Kno66]. In the subsequent work, educational promise was the main motivation for the production of algorithm animations [Hop74, Bae73]. But a real impetus was provided to the field by the video *Sorting Out Sorting* [Bae81] presented at the ACM SIGGRAPH conference in 1981, showing a race among nine 9 different sorting algorithms. Each value of the list was represented by a dot in a matrix, as shown in Fig. 4.8.

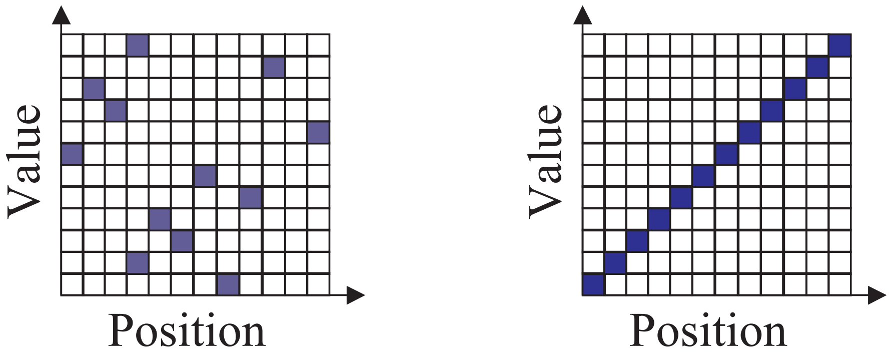


Fig. 4.8. Matrix view: unsorted and sorted data

On the basis of experience with algorithm animations that had been developed from scratch, two seminal algorithm animation tools were developed at Brown University to ease the development of such animations: BALSA[3], by Marc Brown [BS84] (later at MIT and DEC), and TANGO[4], by John Stasko [Sta90a] (later at Georgia Tech). In the following years, at their new institutions Brown and Stasko developed other systems, driven to some extent by the technological advancement in other areas, in particular 3D computer graphics and networked computers (see Table 4.2).

[3] Brown University Algorithm Simulator and Animator.
[4] Transition-based Animation Generation.

Table 4.2. Offspring of BALSA and TANGO

Marc Brown (and Marc Najork)	John Stasko
BALSA (1985)	
BALSA-II (1988)	XTANGO / TANGO (1989)
Zeus (1992)	POLKA + SAMBA (front end)
Anim3D, Zeus3D (1993)	Polka3D (1992)
CAT (Web-based, 1996), JCAT (1997)	

BALSA introduced the concept of interesting events (see Sect. 4.4.6) and related to this, the use of several views of the same state. Later, in BALSA-II [Bro88], step and stop points were added. Finally, in CAT[5] [BN96] and its successor JCAT [BR96], the views were distributed over several computers. TANGO [Sta90a] implemented the path-transition paradigm [Sta90b] to enable smooth, continuous, simultaneous animations of state transitions. TANGO is actually an interpreter for animation commands. The idea was to implement algorithms with an arbitrary programming language and have them produce commands in the SAMBA language as textual output. To this end, the program usually would print these at interesting program points, either to standard output or into a file. This output was then fed post-mortem into the TANGO interpreter to produce the animation. To animate parallel algorithms a library called POLKA[6] was developed [SK93, Sta] and, using POLKA an animation interpreter called SAMBA [Sta97] enabled post-mortem visualizations similar to TANGO. Traces had already been used by PVM[7] [Sun90] to visualize parallel programs postmortem. In this case traces were collected at each computer, and merged and visualized later.

Some seminal papers about the classical animation tools are contained in an anthology on software visualization [SDBP98] published in 1998. Today there exist many more algorithm animation systems that have been developed by other researchers, for example Animal, CATAI, Daphne, GANIMAL, Gasp, GeoWin, Jawaa, Jeliot, LEONARDO, and Mocha, to name a few. In 2002 an overview of the more recent developments in the field was published in a state-of-the-art survey of software visualization [KS02].

4.4.4 Some Animations Produced by X-Tango

In the following we look at some animations produced with X-Tango. We start by discussing animations of algorithms for binary trees, linked lists, bin packing and, the n-queens problem.

A binary tree (Fig. 4.9) is a tree where each node has a maximum of two children. A binary tree is natural if the left child is smaller than the right

[5] Collaborative Active Textbook.
[6] Parallel program-focused Object-oriented Low Key Animation.
[7] Parallel Virtual Machine.

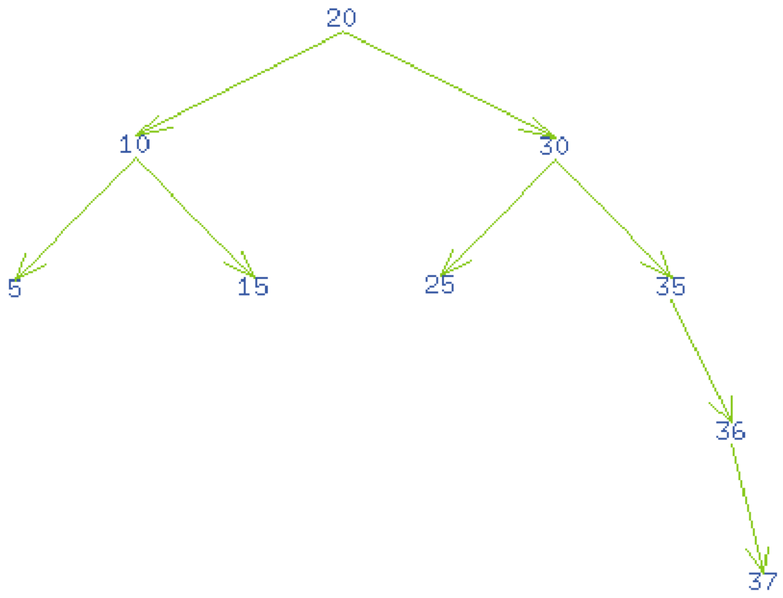


Fig. 4.9. Binary tree

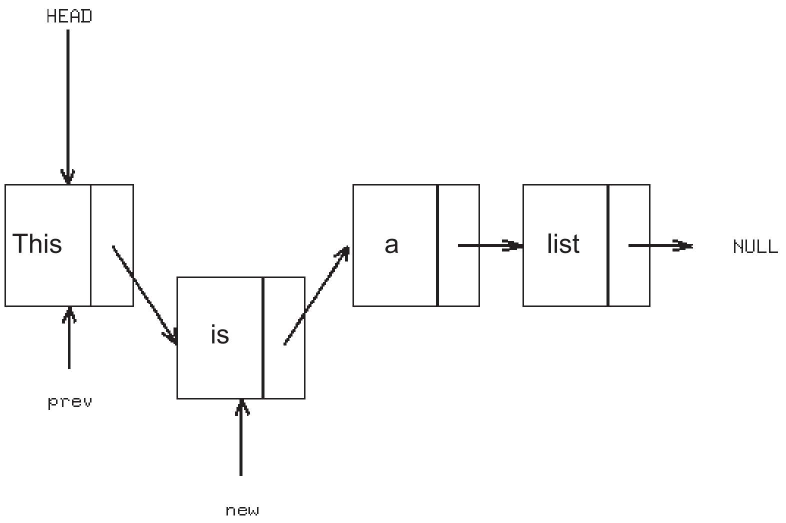


Fig. 4.10. Linked list

child and both children are smaller than their parent node. Typical algorithm animations show the insertion and deletion of nodes.

In a linked list (Fig. 4.10), each node has a pointer to the next element. The pointer to the first element of the list is the head pointer. The next pointer from the last element of the list is undefined (null). Typical algorithm animations show the insertion or deletion of elements at the head or at the end, or at any other position in the list. In the example, the newly inserted element is emphasized by a vertical disalignment with respect to the other elements in the list.

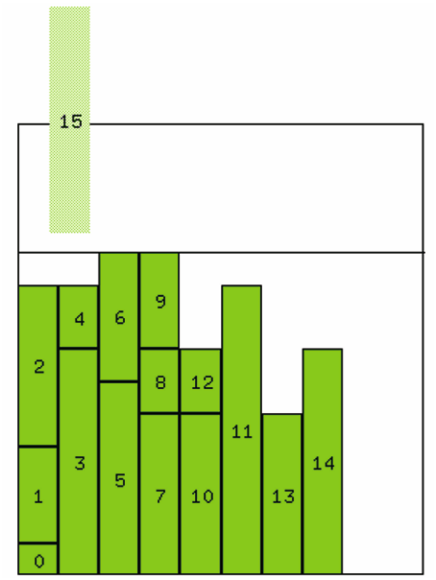


Fig. 4.11. The bin packing problem

In the bin-packing problem (Fig. 4.11), a container with a fixed height has to be filled with boxes of different sizes, such that a minimal amount of horizontal space is used. Boxes can be stacked vertically on top of each other as long as the height of each stack does not exceed the height of the container. There is an online and an offline version of this problem. For the online problem, each box must be placed before the next one is received, i.e. the number of boxes and the size of each of these boxes are not known beforehand. For the offline problem, the number and sizes of the boxes are known before packing. Typical algorithm animations show the next-fit, first-fit and best-fit strategies.

For the n-queens (Fig. 4.12), problem n queens must be placed on an $n \times n$ chess board such that they cannot capture each other. A typical algorithm places the ith queen on the first square of the ith row. If the queen can be

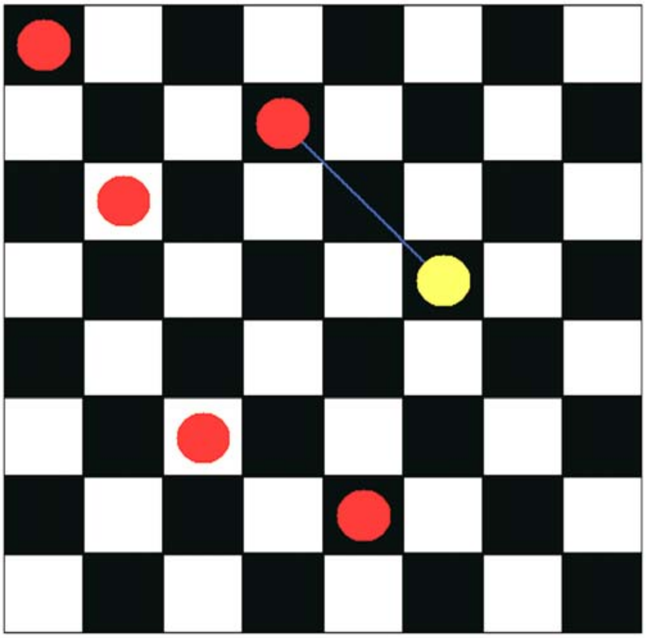


Fig. 4.12. The n-queens problem

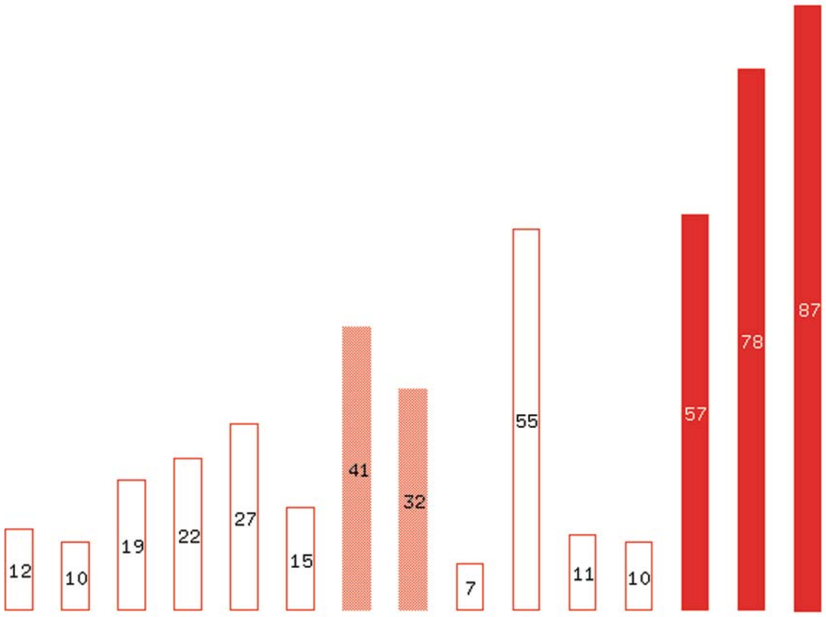


Fig. 4.13. Bubble sort algorithm

captured by one of the previously placed queens, the queen is moved to the next square in the row. Otherwise, the algorithm tries to place the $(i + 1)$th queen recursively. If this fails, the queen is moved to the next square, and so on.

Next we discuss three animations of sorting algorithms: the bubble sort, quicksort and heapsort. Bubble sort (Fig. 4.13) performs a pairwise comparison of the elements in a sequence from left to right and swaps them if the first element is larger than the second. At the end of the first iteration, the largest element is at the end of the sequence. The process is repeated for the rest of the elements until no more elements are swapped. The animation shows how the large elements move to the end of the list.

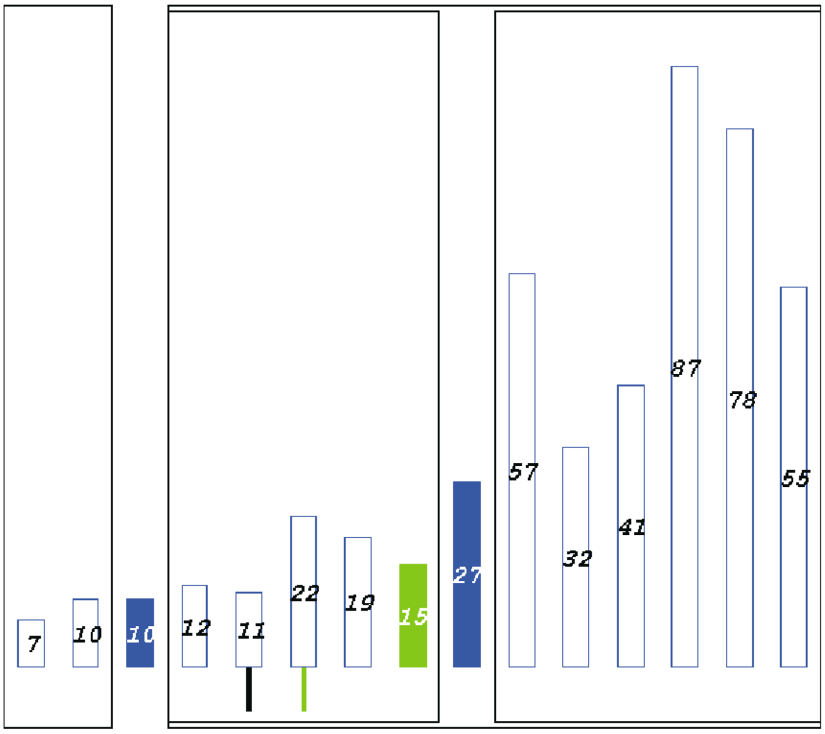


Fig. 4.14. Quicksort algorithm

Quicksort (Fig. 4.14) selects an element and splits the list such that the left part contains all elements of smaller or equal value, and the right part all larger elements. Then these two parts are recursively sorted in the same way. In the animation, elements are displayed as vertical bars, the recursively constructed parts are indicated by nested boxes, and the current pivot element, i.e. the element where the list is split, is colored blue (black in the grayscale image).

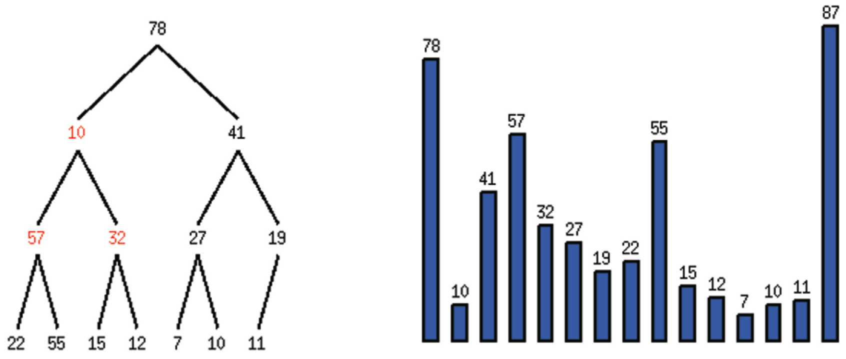


Fig. 4.15. Heapsort algorithm

A heap is a binary tree where the value of each node is larger than that of each of its children. The heapsort algorithm (Fig. 4.15) constructs a binary tree and establishes this heap property such that the largest element is at the root of the tree. To get the second largest element, the root is replaced by the rightmost leaf of the tree and the tree is traversed to establish the heap property again. The process is repeated until the tree is empty. The heapsort animation in the example uses two views of the same data structure, here the list to be sorted. One view shows the list as a tree diagram, while the other shows it as a bar diagram. In the tree view one can see how the heap property is established, while the bar view indicates how "sorted" the list is.

4.4.5 3D for Algorithm Animation

As with other 3D visualizations, the use of the third dimension is typically motivated by one or more of the following reasons:

Aesthetics: Three-dimensional graphics rendered with photorealistic rendering techniques are appealing to many people.

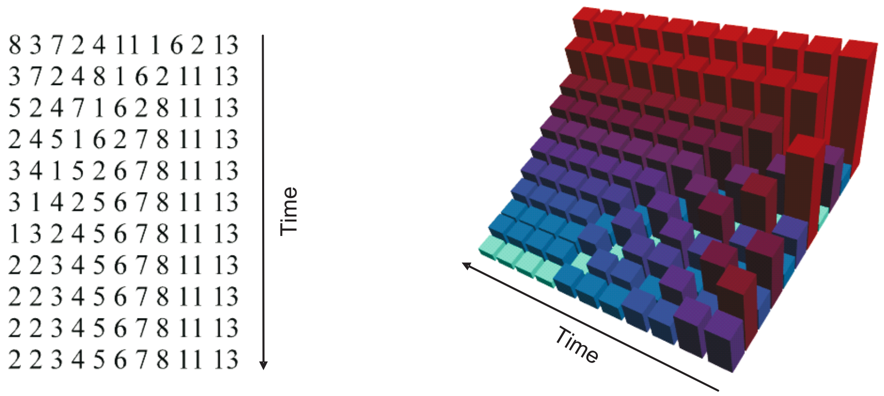
Evolution: Humans are used to three dimensions. It has been argued that the human visual system has been adapted to the real world – and this is a three-dimensional world.

Dimensionality: The third dimension can be used to add additional information to an originally two-dimensional representation. There are two notable instances of this:

- *Multiple views:* The same object can be shown in different ways by placing different, typically two-dimensional, views of the object in the 3D space.
- *History:* The third dimension can be used as a time axis. Along this time axis, the states of an object at different points in time can be shown.

Inherence: In some domains data structures or algorithms are inherently three-dimensional. For example, algorithms in 3D geometry such as triangulation work with three-dimensional data.

Figure 4.16 shows a screen dump of an animation of the bubble sort algorithm implemented with VRML [CB97] and JavaScript. The third dimension is used to show the history of the sort, i.e. the partially sorted sequences are shown along the Z axis.



```
8 3 7 2 4 11 1 6 2 13
3 7 2 4 8 1 6 2 11 13
5 2 4 7 1 6 2 8 11 13
2 4 5 1 6 2 7 8 11 13
3 4 1 5 2 6 7 8 11 13
3 1 4 2 5 6 7 8 11 13
1 3 2 4 5 6 7 8 11 13
2 2 3 4 5 6 7 8 11 13
2 2 3 4 5 6 7 8 11 13
2 2 3 4 5 6 7 8 11 13
2 2 3 4 5 6 7 8 11 13
```

Fig. 4.16. Three-dimensional animation of bubble sort

As another example, look at the 3D animation of the shortest path algorithm (single source shortest path, SSSP) in Fig. 4.17, which was produced with Zeus3D [BN93]. In this animation the third dimension is used to display additional information, here the cost. The graph is drawn in the XY plane, and the Z axis indicates, for each node, the cost of getting from the source node to this node. Consequently, the source node has a cost of 0. At the end of the algorithm, the shortest-paths tree is shown, where for each node, the shortest path is the ascending path with the lowest height.

Design Issues Looking at the examples, it is quite illuminating to see how they address the following design issues:

How are invariants visualized? In the 3D shortest-path computation, the invariant is shown as follows: along each path, the columns have increasing height.

How does focusing work? The active parts of the data structure can be drawn in a certain color, a pointer can be placed next to them, they can move to a certain location on the screen, or their size can be increased (zooming).

How is recursion displayed? The depth of a recursion and the various functions invoked can be displayed by use of frames as in the quicksort example, or by use of colors or even sound.

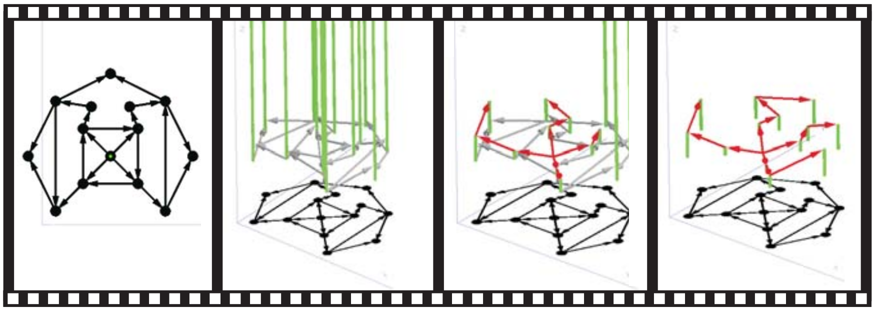


Fig. 4.17. 3D animation of SSSP (©1993 ACM)

4.4.6 Architectures of Algorithm Animation Tools

Algorithm animation tools have been designed with very different goals in mind. A tool cannot be easy to use, comprehensible, and powerful at the same time. What can actually be done with an algorithm animation system and what cannot be done, depends heavily on how the algorithms and animations are coupled or separated. The following architectures have been used to implement various algorithm animation systems and other kinds of dynamic program visualizations:

Ad hoc: The animation of a single algorithm is implemented without using a tool at all, but by implementing everything from scratch.

Libraries: To implement the animation of a single algorithm, libraries containing graphical abstractions, control elements, etc. are used. Such a library might, for example, come with a VCR-like GUI with start, stop, and play buttons.

Special datatypes: The algorithm is programmed with datatypes which have built-in visualizations. Thus the animation is just a by-product when the program is run.

Postmortem: The algorithm and visualization tool are two separate applications. When the algorithm is executed, a trace or animation plan is produced and later visualized by a separate component.

Interesting events: The algorithm is annotated at essential program points with interesting events. During execution, these events are sent to one or more views. The approach usually applies the MVC[8] design pattern.

Declarative: The annotations and algorithm are separated. There are two approaches. The first is *state mapping*, where a demon watches state changes and updates the visualization of the state accordingly. The second approach uses *constraints-based systems*. These work in a similar way, but the program to be visualized is itself written in a constraints-based language.

[8] Model-View-Controller

Semantics-directed: The algorithm is executed by a visual interpreter or debugger which produces the visualizations automatically, but usually on a low level of abstraction.

Both the declarative and the semantics-directed approach are usually non-invasive, i.e. the program code does not need to be changed to get a visualization of the program. In the following we compare the declarative and the interesting-events approach. The examples are taken from a recent paper [DFS02].

Example: Interesting Events in POLKA Using POLKA, programs are annotated at essential program points with calls to methods. These annotations are called interesting events. Whenever one of these program points is reached during the execution of the program, the method sends information about the current program state to all views. The code of our example is

```
void main() {
 bsort.SendAlgoEvt("Input",n,v);
 for(j=n; j>0; j--)
    for(i=1; i<j; i++)
        if (v[i]>v[i+1])
           { int temp= v[i];
              v[i]=v[i+1];
              v[i+1]=temp;
              bsort.SendAlgoEvt("Exchange",i,i+1); } }
```

In this example an event called `Input` is sent to the view referenced by `bsort` at the start of the program. Within the nested loops, an event called `Exchange` is sent whenever two list elements are swapped. Note that in this example there are no other events sent. In particular, the view is not informed whenever two elements are compared, and thus it cannot visualize how the execution of the program proceeds.

Example: Declarations in LEONARDO LEONARDO [DF] integrates both a C and a Pascal compiler together with a source code editor. It uses a virtual machine with invertible instructions to visualize a program and provide undo/redo functions for every execution step.

The visualization is separated from the program by writing the visualization declarations as comments in the program code. The declarations are written in a kind of logic programming language called ALPHA. This describes a set of visual objects. An ALPHA program consists of a sequence of predicates that define these objects and their relationships. Each predicate is defined by a head–body pair, where the head specifies the name and formal parameters while the body specifies the computation. The code of our example is

```
void main() {
 for(j=n; j>0; j--)
    for(i=1; i<j; i++)
        if (v[i]>v[i+1])
           { int temp= v[i];
              v[i]=v[i+1];
              v[i+1]=temp; } }

/**  View(Out 1);
     Rectangle(Out ID, Out X, Out Y, Out L Out H, 1);
     For N: InRange(N,0,n-1)
         Assign X=20+20*N   Y=20  L=15  H=15*v[N]  ID=N;  **/
```

In this example, program variables such as `n` and `v` are written in lower case, while variables of the animation language such as `N`, `H` and `ID` are written in upper case. For example, the height of the `N`th rectangle depends on the value of the program variable `v` according to the following equation: `H=15*v[N]`. Thus, whenever the value of the `N`th element of the array `v` changes, the height of the rectangle with an `ID` equal to `N` is adapted.

4.4.7 Abstract Algorithm Animation

Focusing is a big problem in algorithm animation. What part of the data structure should be displayed on the screen? Not all data structures need to be displayed at all times, and often the amount of data makes it impossible to show all data simultaneously. So we have to manage the screen estate carefully.

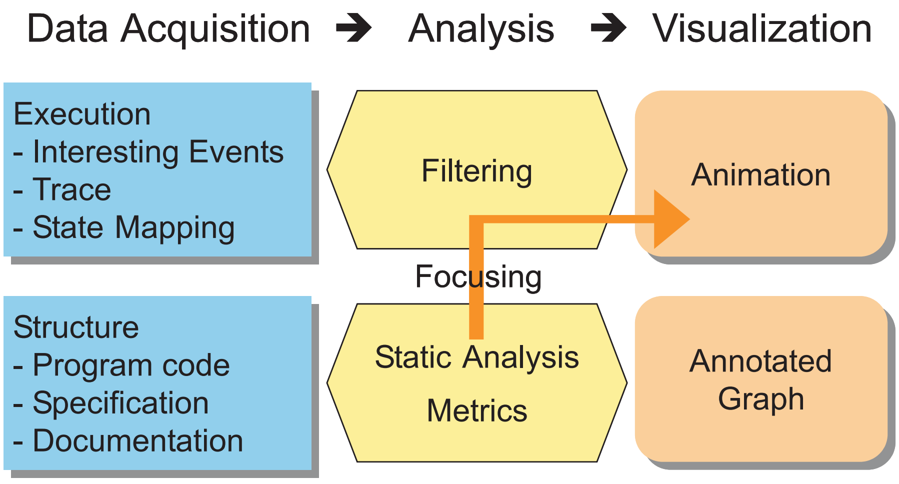


Fig. 4.18. Using program analysis to focus algorithm animations

Figure 4.18 shows how the visualization pipelines of static and dynamic program visualization could be combined to this end. Despite its potential, the use of static program analysis to focus algorithm animations is largely unexplored.

Usually, at each program point only a small fraction of the data can be accessed. One solution to determining this fraction before run time is to use static program analyses which compute information about the accessible data structures at each program point. Such an analysis has been developed by Sagiv, Reps, and Wilhelm [SRW96, SRW99] and is called shape analysis. It computes an abstract representation of linked data structures, which focus on the active parts of these structures. For each program point, it yields a finite set of shape graphs.

Braune and Wilhelm have suggested that abstract execution should be animated on the basis of shape graphs [BW00]. They call the resulting animations "algorithm explanations" to emphasize that they show the invariants of the data structures at each program point. Recently, this approach has been extended to also show nonstructural invariants [WMS02].

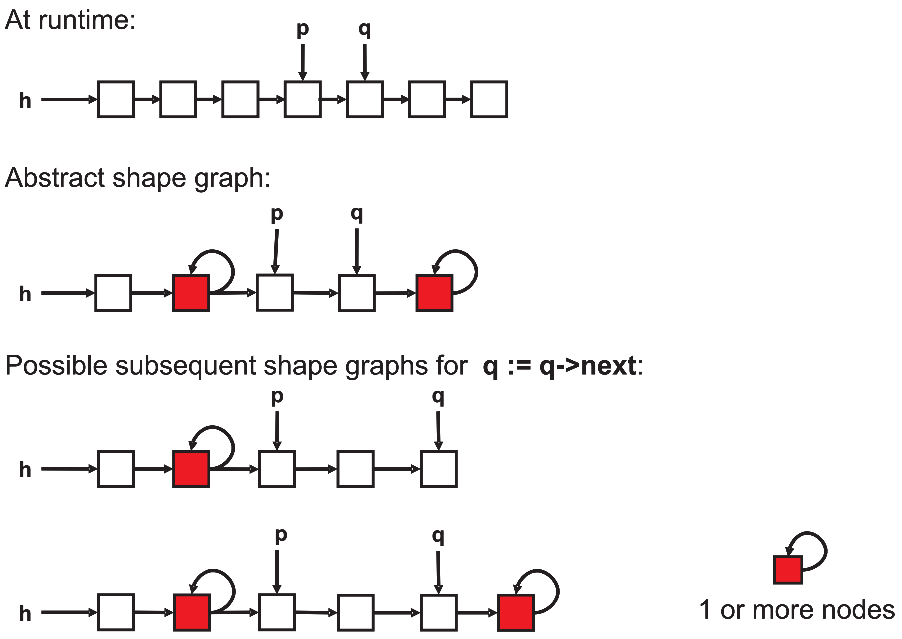


Fig. 4.19. Abstract execution of `q := q->next`

In Fig. 4.19, the abstract execution of a program point is shown. The abstract shape graph before the execution shows that the pointer `q` points to an element immediately after the element pointed to by `p`. Furthermore, there

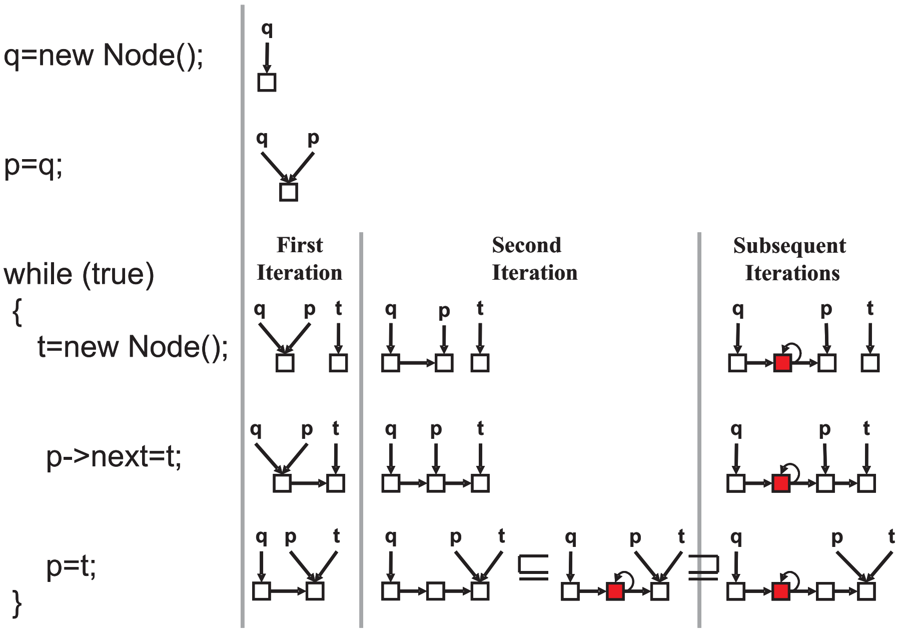


Fig. 4.20. Abstract execution of a simple program

is at least one element after the element pointed to by `q`. If we now execute the assignment `q:=q->next`, then there are two possible subsequent abstract shape graphs. The first shows the case where there was exactly one element more in the list, while the second abstract shape graph represents the case where there were at least two more elements.

Thus an abstract shape graph represents a set of concrete shape graphs, i.e. pointer structures that can occur during the execution of a program. Let γ be a function that yields, for a given abstract shape graph, the set of concrete shape graphs that it represents. A transition from an abstract state as_1 to an abstract state as_2 is only legal[9] if a transition from a concrete state cs_1 to a concrete state cs_2 exists, where cs_1 is represented by as_1 and cs_2 by as_2, or, formally, $cs_1 \in \gamma(as_1)$ and $cs_1 \in \gamma(as_1)$. Visual abstract execution must show only legal transitions. Figure 4.20 shows the abstract visual execution of a simple loop that creates an endless list. To the right of each program point we see shape graphs that describe the various possible program states after the instruction at that program point has been executed. For each program point within the loop, there are three such cases: one for the first, one for the second, and one for all subsequent iterations. So, for each program point in the loop, the possible states are described by a set of three shape graphs, which show the structural invariants that hold after the program point has

[9] With respect to a given program.

been executed. Thus the abstract visual execution can also be seen as a visual proof of partial program correctness. Note that the abstract shape graphs for the second and subsequent iterations in the last row are subsumed by the one shown between them. An abstract shape graph as_2 subsumes an abstract shape graph as_1, written $as_1 \sqsubseteq as_2$, if all concrete shape graphs represented by as_1 are also represented by as_2, or, formally, $\gamma(as_1) \subseteq \gamma(as_2)$.

Abstract algorithm animation is still in its infancy. Even for small programs, the set of shape graphs for each program point becomes very large. A promising solution seems to be to partition these sets such that each partition contains shape graphs that represent similar cases and thus do not have to be visualized separately [JSW05].

4.4.8 Learning Scenarios

When algorithm animations are used in education, the learner typically has to perform one of the following three tasks:

View animation of algorithm: In the simplest case, there are ready-made animations with fixed inputs.[10] Using existing algorithm animation tools, animations which allow user input are also possible.

Read algorithm and view animation: As before, one can use fixed animations, as well as those produced to allow user input, but in addition the learner is asked to read the description of the algorithm beforehand or step by step while the algorithm is running.

Implement algorithm and create your own animation: Easy-to-use algorithm animation tools enable the student to implement both the algorithm and its animation. As a side effect, the animation helps students to debug their implementations of the algorithm.

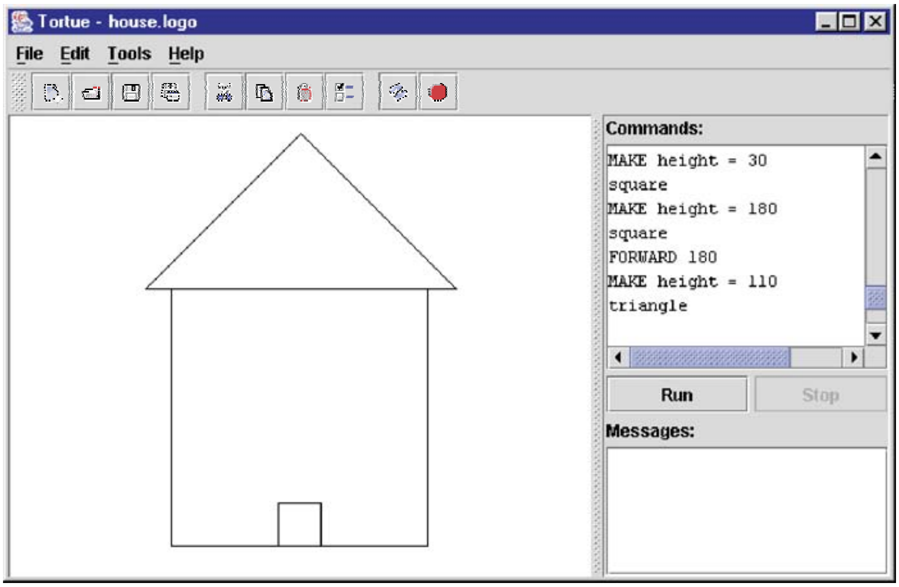
These tasks differ in the level of learner involvement. While for the first two it is usually low, implementing both the algorithm and the animation is more demanding. [11]

In order to achieve higher learner involvement without requiring programming of animations, some more sophisticated scenarios have been developed. Two of these are described below.

Exploring the Functional Structure Faltin has suggested an approach that supports exploratory learning by only providing the building blocks of an algorithm, so that the learner actively reinvents parts of an algorithm

[10] Actually, to produce such animations, neither an algorithm animation system nor an implementation of the algorithm is needed. The author of such an animation can use standard graphics tools to produce the individual frames of the animation.

[11] The engagement taxonomy suggested in a recent workshop report [NRA$^+$03] contains some more tasks: no viewing, viewing, responding, changing, constructing, and presenting.



```
MAKE height = 30
square
MAKE height = 180
square
FORWARD 180
MAKE height = 110
triangle
```

Fig. 4.21. Composing steps

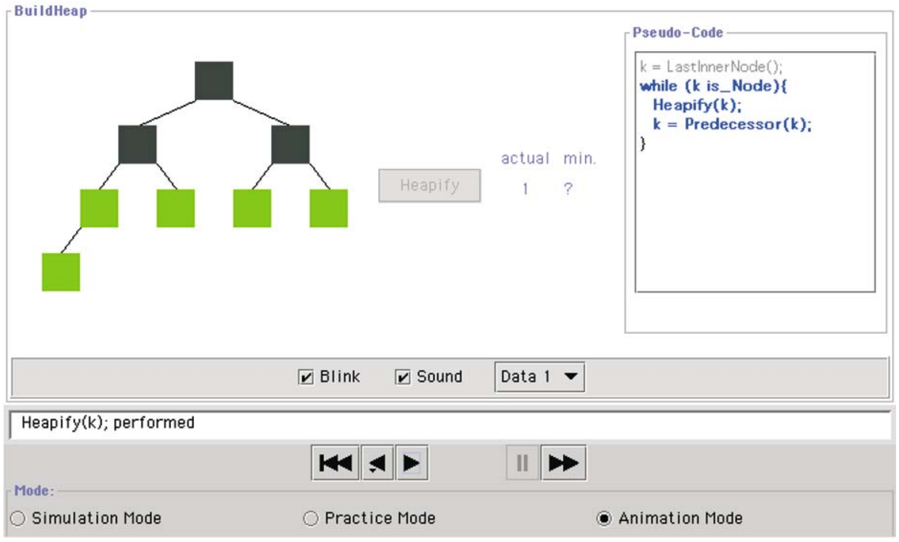
[Fal02]. To this end, the algorithm is structured into many small functions. The task of the student is to find the steps of a function for a specific data input. To facilitate this task, additional constraints delimit the exploration space, and simulations of each step are provided. The student can test these steps and compose them.

As an example, assume that we provide the students with the procedures `square` and `triangle` in the LOGO programming language. First the students can test these procedures by changing the value of the variable `height`, and then their task is to compose these procedures to draw a house (see Fig. 4.21).

Faltin used the same basic approach to have students explore the functional structure of a more complex algorithm, the binomial heap. One task was to establish the heap property, i.e. that the value of the parent node is less then or equal to those of its children, by comparing and swapping (see Fig. 4.22). Note that the student does not see the values of each node, but has to find out about them by comparing them.

Visualized Path Testing Instead of having the students reconstruct the algorithm from parts, one can gain higher student involvement by having the students find input data for the algorithm that satisfies certain criteria, and provide visualization tools such that the students can check these criteria.

As an example of this approach, we take a closer look at program coverage as a criterion, and a related visualization tool. Program coverage data is gathered by running a program with a test suite, i.e. a set of input data, and



```
k = LastInnerNode();
while (k is_Node){
  Heapify(k);
  k = Predecessor(k);
}
```

Heapify(k); performed

Mode: ○ Simulation Mode ○ Practice Mode ● Animation Mode

Fig. 4.22. Interactively establishing the heap property

keeping track of which parts are actually executed. Often this information is used as a metric, for example 20% of all statements may have been executed. In contrast, coverage criteria are satisfied only if all parts have been executed, i.e. if the percentage is 100%. Coverage criteria differ in what they consider to be a program part: statements, branches, or paths.

Statement coverage is satisfied when every (non-control-flow) statement is executed at least once with the test set. Branch coverage is satisfied when every edge of the flow graph of the program is applied at least once with the test set. Finally, path coverage is satisfied when the test set contains a test case for every possible control path in the flow graph of the program. The problem with path coverage is that the number of paths is exponential with respect to the number of branches.

In the learning scenario considered by Korhonen et al., the task was to find a minimal test set satisfying a certain coverage condition [KST02]. This task was supported by a visualization tool. As an example, consider the following Java program:

```
if ((a<4) || (b>5))
 { b=b+1; }
if (b<2)
 { a=a+7; }
```

The following test sets of pairs (a, b) satisfy the above coverage criteria: the set $\{(0,0)\}$ satisfies statement coverage, the set $\{(0,0),(4,2)\}$ satisfies branch coverage, and the set $\{(0,0),(0,1),(4,1),(4,2)\}$ satisfies path coverage. The four possible paths are shown in Fig. 4.23.

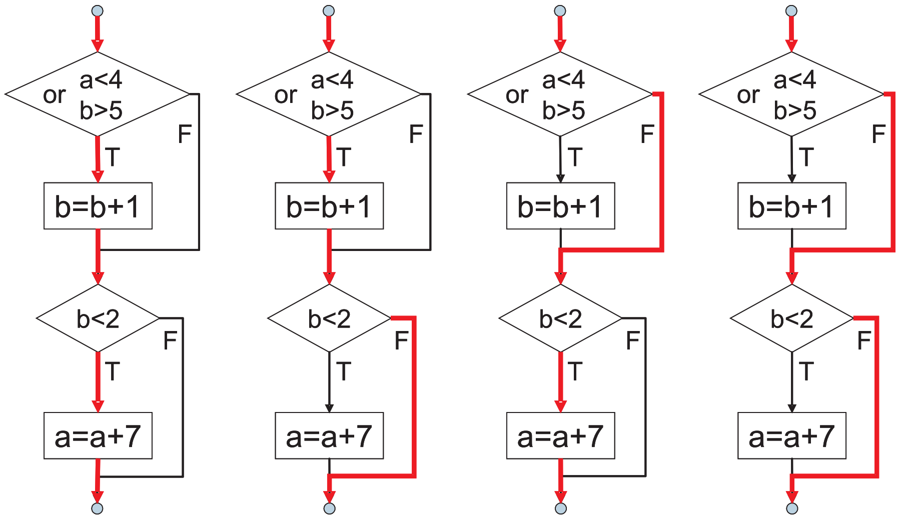


Fig. 4.23. Paths in the control-flow graph of the example program

In simple cases students can perform path testing just by trial and error, but even then finding minimal sets requires argument as to why there is no smaller set. But usually the search space is so large that only systematic thinking will work. The student has to think about the different alternatives at each branch and, formally or at least intuitively, accumulate the conditions along each path to find input values that satisfy these conditions.

4.4.9 A Brief Introduction to SAMBA

SAMBA is an interactive animation interpreter that can be used with any programming language to produce algorithm animations. It was designed to be very easy to learn and to use. Today, there exist versions for Unix and Windows, as well as a Java applet. SAMBA works in batch mode. The actual algorithm is written in some programming language. When the algorithm is executed, it writes, with some form of `print` statements, animation code to a standard output device or a file. The animation code consists of ASCII commands – one command per line. SAMBA reads this code and performs the corresponding animation actions. Each command consists of multiple entries. For example, consider the following SAMBA command:

```
line li  0.1  0.1  0.4  0.2  green thin
```

The first field defines the type of the command, while the remaining fields contain parameters such as identifiers or coordinates. In the above example, a thin green line is created, which is bound to the identifier `li`. The line starts at position $x = 0.1$ and $y = 0.1$ and has a width 0.4 and a height

0.2, which means that its end points are $x' = 0.5$ and $y' = 0.3$. In order to make animations independent of the resolution of the computer display and the window size, the coordinate system runs from 0.0 to 1.0 both horizontally and vertically and is mapped to the real coordinate system of the output window by SAMBA.

SAMBA provides commands to create objects, to modify objects, and to create and alter views:

```
comment CREATE OBJECTS
rectangle box  0.1  0.1  0.2  0.2  blue outline
circle 27  0.8  0.7  0.1  red solid
comment MODIFY OBJECTS
move li  0.5  0.6
color 27  blue
jump 27  0.3  0.4
comment CREATE AND ALTER VIEWS
viewdef MainView 600 600
view MainView
```

Finally, SAMBA also allows explicit concurrency of animation commands with `{` and `}`. All commands in braces will be performed concurrently. In the following excerpts from the animation code for a swapping operation, two nodes and two labels in separate views (windows) change their position simultaneously:

```
view TreeView
move leftnode 0.3 0.3
move rightnode 0.1 0.1
view TableView
move leftlabel 0.1 0.4
move rightlabel 0.1 0.2
```

The following Java implementation of Insertion Sort was annotated with print statements to produce SAMBA animation code. The code added to the original sorting algorithm is printed in bold face:

```
public class JSAMBAExample
{ static int A[] = new int[]5,7,3,9,2,9,3,6,8,4;
  static float width=0.08f, height=0.05f;

  public static void main(String argv[])
  { int i,k,x;
    init();
    for (k=1;k<10;k++) {
      println("moverelative line "+width+" "+0.0f);
      x=A[k];
      println("moverelative "+k+" "+0.0f+" "+0.5f);
      i=k-1;
      println("move tri "+(i+1.5f)*width+" "+height/2);
      while (i>=0 && A[i]>x) {
        println("moverelative "+i+" "+width+" "+0.0f);
        println("swapid "+i+" "+(i+1));
        println("moverelative tri "+(-width)+" "+0.0f);
        A[i+1]=A[i];
        i--; }
    A[i+1]=x;
    println("move "+(i+1)+" "
                   +(width*(i+2)+0.005f)+" "+height);
   }
  }

  static void init()
    { float xpos, ysize;
      float ypos=(float) height, xsize=width-0.01f;
      for (int i=0;i<10;i++) {
        xpos=width*(i+1)+0.005f;
        ysize=height*A[i];
        println("rectangle "+i+" "+xpos+" "+ypos+" "
                          +xsize+" "+ysize+" blue solid"); }
     println("triangle tri "+width*1.25f+" "+height/3+" "
                       +width*1.75f +" "+height/3+" "
                       +width*1.5f+" "+height/3*2
                       +" green solid");
     println("pointline line "+width*2+" "+0+" "
                              +width*2+" "+height*12
                              +" red thin");
    }

  static void println(String s) { System.out.println(s); }
}
```

The method `init()` creates rectangles that represent the various array cells (see Fig. 4.24). The height of each rectangle represents the value stored in the array cell. Furthermore, a triangle and a vertical line are created. During the animation, the vertical line is always placed to the right of the `k`th element, i.e. the one which is currently to be inserted. The triangle shows which element it is currently compared with. All expressions to compute coordinates are completely evaluated when the Java program is executed, and only numbers are printed. Thus the animation code does not contain any variable names, only numbers.

For more details on SAMBA, see John Stasko's technical report [Sta97]. More examples and download information can be found online [Joh, Sol].

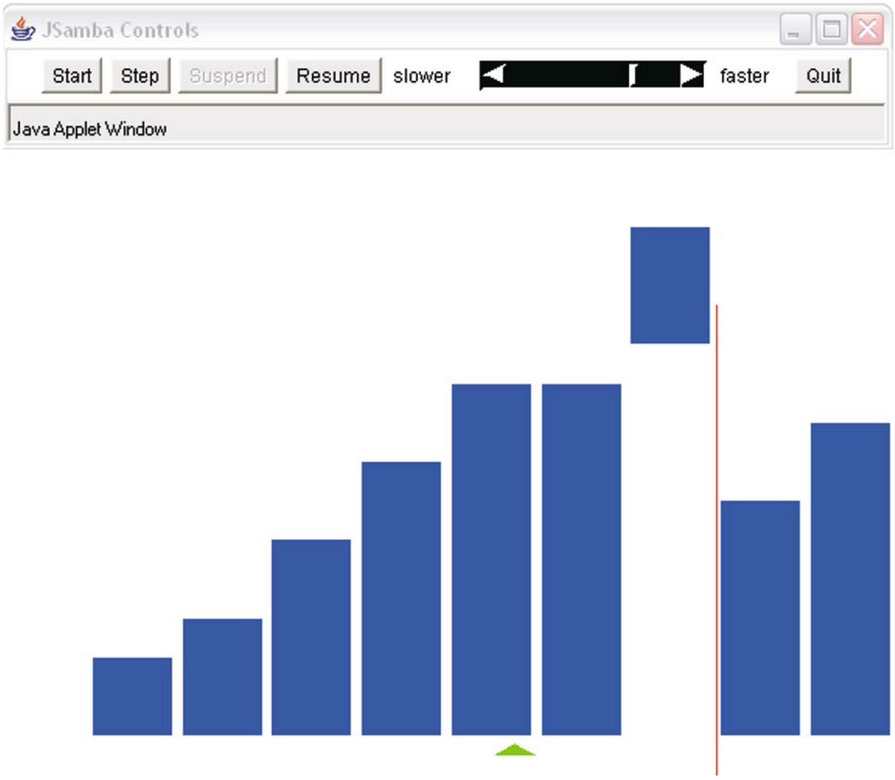
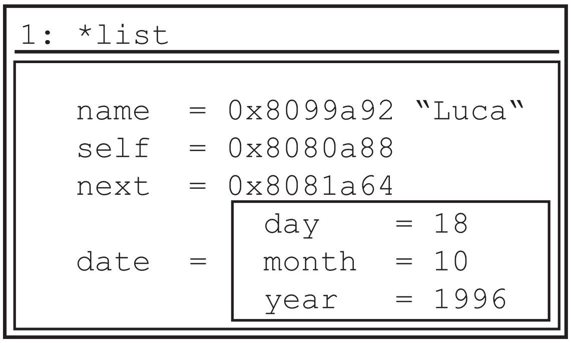


Fig. 4.24. JSAMBA animation of a sorting algorithm

4.5 Visual Debugging – Inspecting the Program State

The goal of debugging is to detect the existence of errors in a program, to locate their position or cause, and, finally, to fix them. In the following, we shall focus on visualization techniques that help to locate errors in programs. In principle, there are two kinds of visualizations: those which show program states or memory and those which display program code and test results. In this section, we shall look at incremental interactive unfolding of the program state and automatic traversal of the program state to produce more abstract representations. For large data structures, methods to focus on changes and to group similar objects to detect reference patterns are discussed. In the next section, we will discuss how debugging can be supported by running a program with a set of test cases and visualizing information about what parts of the program have been executed in failed and passed runs.

Even the program state of a simple C or Java program that prints “Hello World” on the screen consists of several thousand bytes because of the many data structures required by the runtime system. For multimedia applications,



```
1: *list
   name  = 0x8099a92 "Luca"
   self  = 0x8080a88
   next  = 0x8081a64
                day    = 18
   date  =      month  = 10
                year   = 1996
```

Fig. 4.25. Nested boxes as drawn by DDD

the amount of runtime data can easily exceed several gigabytes. Displaying all this data is impossible. The programmer needs techniques to select only certain parts of the program state or to somehow aggregate the data such that it fits onto the screen.

4.5.1 Interactive Visual Unfolding

Some common subtasks of debugging are, first, to identify the statements involved and then to reduce the problem further by selecting statements which might contain faults. Then one has to come up with hypotheses about the faults, and set program variables to a specific state [DPS97].

As an example of an interactive visual debugger that supports these tasks, we look at the Data Display Debugger (DDD), which allows one to visualize program states [Zel01, ZL96]. Programs are executed in a defined environment, and execution stops at situations specified at conditional break points. The user can then inspect and modify the program state, and continue execution. In a textual debugger such as the GNU `gdb`, the user enters a command such as `display *list` and the debugger prints information about the program state, for example the data structure referenced by the pointer `*list`:

```
(gdb) display *list
*list = { name = 0x8099a92 "Luca",
  self = 0x8080a88,  next = 0x8081a64,
  date = { day = 18, month = 10, year = 1996}}
(gdb) _
```

GNU `gdb` uses some of the techniques discussed in Sect. 3.1.1 for pretty printing, but the output is still difficult to read. Note that the attribute `date` refers to another data structure, which is indicated by nested braces. To see the data structure referenced by the attribute `next`, the user has to enter another command, and that data structure will be printed as before.

DDD can be used as an extension to command line debugger such as GNU `gdb`. It shows data structures as nested boxes (see Fig. 4.25). To follow a pointer in a data structure, the user clicks on it and its value is unfolded, i.e. the referenced data structure is drawn in another box and an arrow points from the attribute to this box (see Fig. 4.26). Thus, in DDD, data structures are incrementally unfolded by user interaction. Note that the user selects which parts are unfolded and which are not. DDD will also detect that two pointers refer to the same node and draw this node only once (see Fig. 4.27).

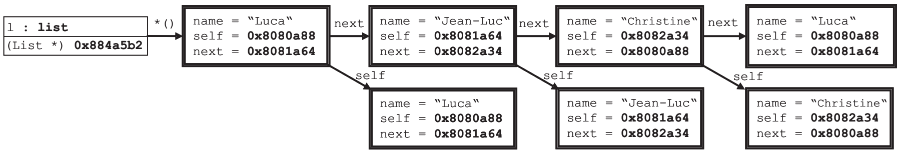


Fig. 4.26. Unfolding of linked data structures by DDD

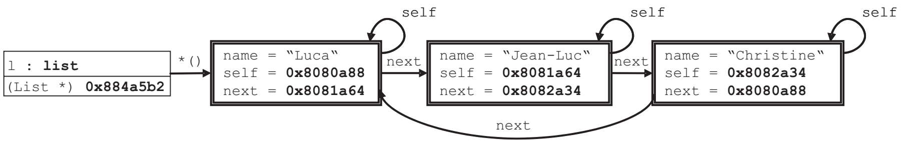


Fig. 4.27. Alias detection by DDD

4.5.2 Traversal-Based Visualization

In DDD, data structures are unfolded step by step. For a large data structure, for example a long linked list, this becomes a tedious task. In a visual debugging prototype developed at Princeton [KA98], the whole data structure is unfolded at once with the help of visualization rules. More precisely, the system traverses linked data structures. At each node, it matches the data found with the patterns of the visualization rules. If a rule matches, it produces a set of visualization objects – the visual model. This model is then rendered by a separate component (see Fig. 4.28).

Visualization rules are defined in a textual notation (see Fig. 4.29). The pattern on the left-hand side of the rule, i.e. on the left side of the colon, checks whether attributes of the current data structure have a certain value. If this is the case, the code on the right-hand side creates new visual objects and controls where to continue the traversal of the data structure.

In the example, the data structure of the program is an instance of the class `Op` in Java:

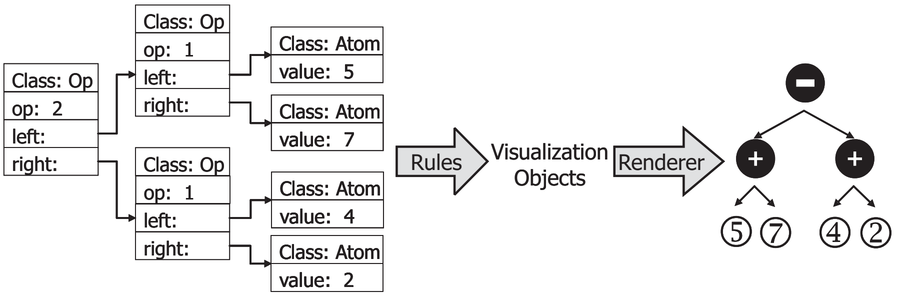


Fig. 4.28. Applying visualization patterns to data structures

```
class Op {
  final static int PLUS=1;
  final static int MINUS=2;
  int op;
  Expr left;
  Expr right;
  ...
 }
```

The visualization objects for trees are instances of the Java classes `TreeNode` and `TreeEdge`:

```
class TreeNode { String icon; ... }
class TreeEdge { TreeNode from; TreeNode to; }
```

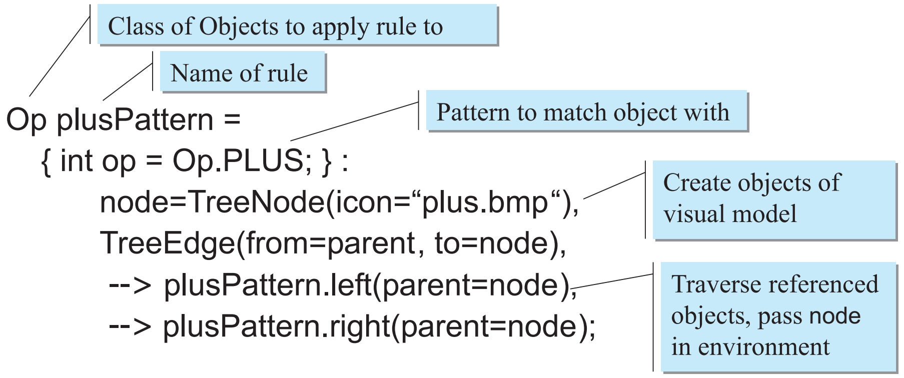
Given the above classes, the rule in Fig. 4.29 will create a new `TreeNode` with an icon for the plus sign, and a `TreeEdge` from the node bound in the environment to the variable `pattern` to the new node. Then it will continue traversal with the left and right children, with the variable `parent` bound to the new node.

What if the data structures to be visualized are really large? By large, we mean that they consist of several hundreds or thousands of elements. Interactive unfolding would take too long, and visualization rules would produce excessively large graphs. In the following sections, we discuss two approaches to handling large data structures. First, one can focus on modified parts.[12] Second, one can group elements to form collections of data with similar structures.

4.5.3 Memory Graphs and Memory Slices

Memory graphs represent the memory of a program. Nodes correspond to memory content, and arrows indicate possible access paths. Memory graphs

[12] Here the focusing is based on runtime information, whereas in abstract algorithm animation static information is used (see Sect. 4.4.7).



```
Op plusPattern =
   { int op = Op.PLUS; } :
         node=TreeNode(icon="plus.bmp"),
         TreeEdge(from=parent, to=node),
          --> plusPattern.left(parent=node),
          --> plusPattern.right(parent=node);
```

Class of Objects to apply rule to
Name of rule
Pattern to match object with
Create objects of visual model
Traverse referenced objects, pass node in environment

Fig. 4.29. Visualization rule

are computed by unfolding all accessible data structures in the program. In [ZZ02], Zimmermann and Zeller computed memory graphs for C programs, where all common data structures, such as structs, unions, arrays, and pointers are properly represented.

In the following we define memory graphs for Java.[13] A memory graph is a tuple $G = (V, E, root)$, where V is a set of nodes, E is a set of edges, and $root$ the root of the memory graph. Let V_s be the set of all static variables, let V_l be the set of all local variables, and let V_a be the set of all arguments of methods. [14] We define the edges and nodes of a memory graph as follows:

- $(root, v) \in E$ for all $v \in V_s \cup V_l \cup V_a$,
- if $(v_1, v_2) \in E$ then $\begin{cases} (v_2, o) \in E & \text{if } v_2 \text{ is a variable refering to an object or array } o, \\ (v_2, \texttt{null}) \in E & \text{if } v_2 \text{ is a reference variable with value } \texttt{null}, \\ (v_2, v2.v) \in E & \text{if } v_2 \text{ is an object and } v \text{ a static or non-static object variable of } v_2, \\ (v_2, e_i) \in E & \text{if } v_2 \text{ is an array and } e_i \text{ its elements of either primitive or reference type,} \end{cases}$
- $(o.v, o') \in E$ if $(v_1, o.v) \in E$ where v is an object variable referring to an object or array o,
- V is the set of all nodes occurring in E.

Figure 4.30 shows a part of the memory graph of a Java application. Note that one of the `Address` objects is referenced by two different objects.

[13] A very similar concept called Java object graphs [PNB04] has been introduced by Potanin et al.

[14] To be complete, one would also have to consider the set of references registered through the Java Native Interface (JNI) for the start set.

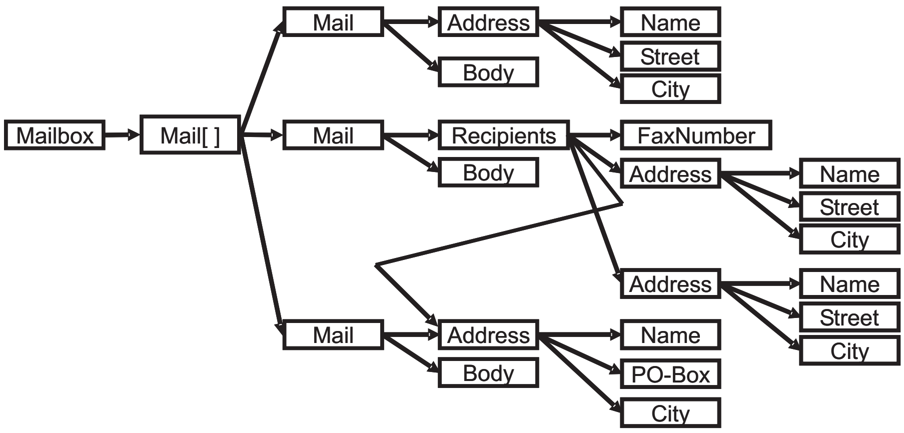


Fig. 4.30. Example: Part of a memory graph (forward memory slice) of a Java application

Given a memory graph, we can compute the forward memory slice for an object o. This contains all access paths which start at the object o. More precisely, we define the forward memory slice $S_F(o)$ as follows:

$$S_F(o) = (V, \{(v_1, v_2)|\ o \rightarrow^* v_1 \text{ and } (v_1, v_2) \in E\})$$

Analogously, we define the backward memory slice as follows:

$$S_B(o) = (V, \{(v_1, v_2)|\ v_2 \rightarrow^* o \text{ and } (v_1, v_2) \in E\})$$

This contains all access paths which lead to the object o. One interesting application of these memory slices is to find possible paths by which an object o_2 is reachable from an object o_1. This can be done by computing the memory chop $S_F(o_1) \cap S_B(o_2)$ which contains only those edges which occur in both slices.

Memory graphs tend to be large; for example the memory graph of the GNU compiler has about 40 000 nodes. It is possible to fit such a graph on the screen and navigate in it with, for example, a hyperbolic graph viewer. But the sheer size of the graphs renders finding interesting parts almost impossible. So one needs to have automatic support to find those parts. Zimmermann and Zeller achieved this by computing graph differences between two memory graphs. Actually, they computed the greatest common subgraph, and the remaining edges and nodes were those that were different between the two graphs.

Now, how does this help with debugging? Assume that you have identified a statement or method at which your program crashes. By looking at the differences between the memory graphs before and after the execution of that statement or method, one can, for example, detect far-reaching side effects, which might not have been explicit from reading the source code.

If we ignore what the nodes stand for, memory graphs abstract from concrete values and addresses and just present the structure, and are subject to standard graph operations such as tests for cyclicity or isomorphic subgraphs. As a consequence, it is even possible to compute the graph differences between memory graphs resulting from different runs of the same program. Looking at the differences between memory graphs which have been captured at the same break point for a successful and a failing run of a program can help to identify those parts of the program state that caused the error.

4.5.4 Reference Patterns

Instead of intersecting memory graphs, the visual Java debugger Jinsight [IBMb, PJM$^+$02] developed at IBM Research groups objects of the same class together to form what are called reference patterns. Reference patterns are trees. In a forward reference pattern, each node represents all objects of a single class which are *referenced by* at least one of the objects represented by the parent of that node. In a backward reference pattern, each node represents all objects of a single class which *refer to* at least one of the objects represented by the parent of that node. Reference patterns were originally developed to find memory leaks in Java programs [dPS99]. A similar approach can be used to extract execution patterns from a dynamic call graph [PLVW98].

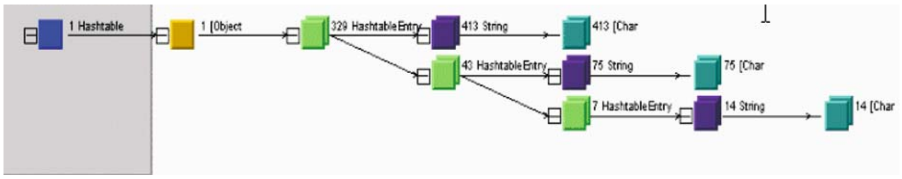


Fig. 4.31. Reference pattern extracted for a hash table

Figure 4.31 shows a forward reference pattern of depth 6 as it is displayed by Jinsight. The node to the left represents only one `Hashtable` object; it refers to an `Object` array which contains 329 objects of class `HashtableEntry`. These 329 objects refer to 413 objects of class `String`, and 43 objects of class `HashtableEntry`. In total, the pattern gives a summary of about 900 objects, assuming that there are no shared objects.

More precisely, a reference pattern is a tuple $(V, E, root)$, where the nodes in V have the form (O, t, d), and

- O is a set of objects of the same type t;
- d is the depth of this node, i.e. the length of the path from the root $root$ to this node.

Given a forward memory slice $S_F(o) = (V, E, root)$, we define a forward reference pattern $R_F(o) = (V', E', root')$ of the object o as follows:

- $root' = (\{o\}, t, 0)$, where t is the type of the object o,
- $root' \in V$,
- if $(O, t, d) \in V'$

$$\begin{cases} ((O,t,d),(O_i,t_i,d+1)) \in E' & \text{if } t \text{ is an object type and} \\ & O_i = \{o' | (o, o.v), (o.v, o') \in E,\ o \in O, \\ & v \text{ is an object variable,} \\ & \text{and the type of } o' \text{ is } t_i\}, \\ ((O,t,d),(O',t',d+1)) \in E' & \text{if } t \text{ is an array type and} \\ & O' = \{o' | (o, o') \in E,\ o \in O, \\ & \text{and the type of } o' \text{ is } t'\}. \end{cases}$$

Analogously, given a backward memory slice $S_B(o) = (V, E, root)$, we define a backward reference pattern $R_B(o) = (V', E', root')$ of the object o as follows:

- $root' = (\{o\}, t, 0)$ where t is the type of the object o,
- $root' \in V$,
- if $(O, t, d) \in V'$

$$\begin{cases} ((O,t,d),(O_i,t_i,d+1)) \in E' & \text{where } O_i = \{o' | (o', o'.v), (o'.v, o) \in E, \\ & o \in O,\ v \text{ is an object variable,} \\ & \text{and the type of } o' \text{ is } t_i\}, \\ ((O,t,d'),(O',t',d+1)) \in E' & \text{where } O' = \{o' | (o', o) \in E,\ o \in O, \\ & \text{and the type of } o' \text{ is an array type } t'\}. \end{cases}$$

Because of cycles, reference patterns can become infinite, and thus in practice we only compute them up to some given depth.

Jinsight can also show sharing of objects. For every object that is represented by several nodes in the reference pattern, an arrow is drawn from these nodes to one of them, for example the one that was visited first when the memory graph was traversed. Figure 4.32 shows a reference pattern extracted from a forward memory slice of a Java application.

4.6 Visual Testing – Detecting Possibly Buggy Program Code

Detecting those parts of the program state that have incorrect values does not necessarily mean that the programmer knows what part of the program code led to these wrong values and needs to be fixed. In this section, we look at techniques that can highlight possibly buggy program parts. These techniques are based on the following heuristic: parts of a program that are only or mostly executed when the program produces an error are more likely to contain a bug.

4.6.1 Dynamic Program Slices

In the previous sections we have seen that one can regard the data stored in memory as a huge graph and focus on certain subgraphs, which we called mem-

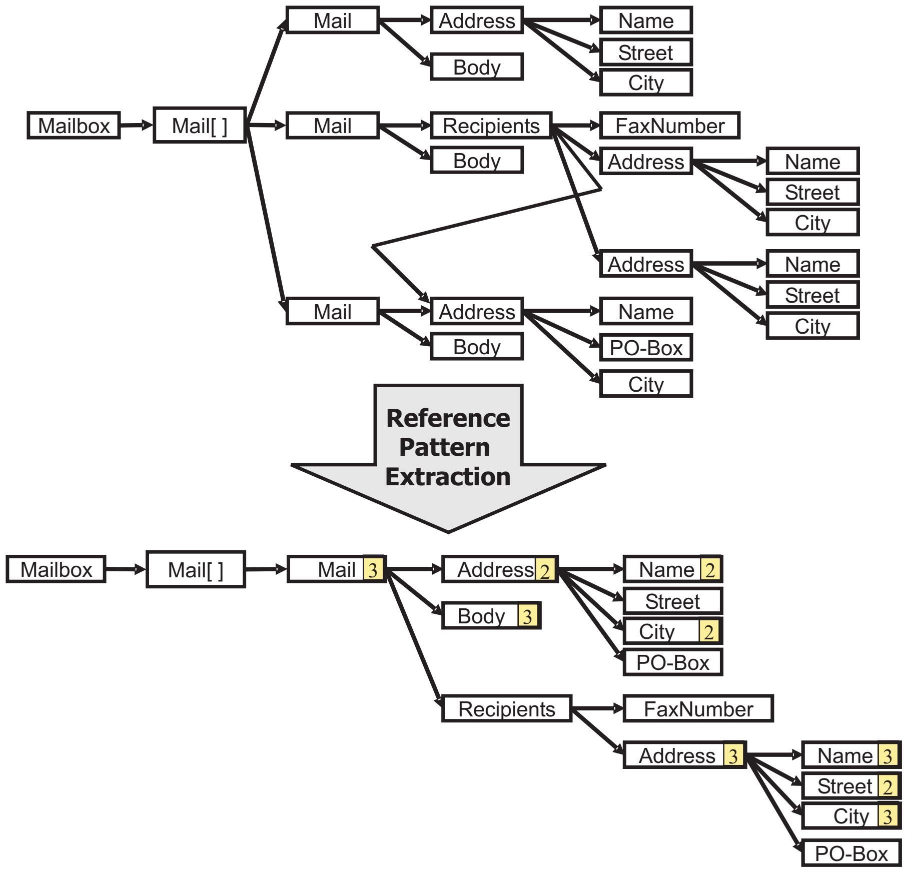


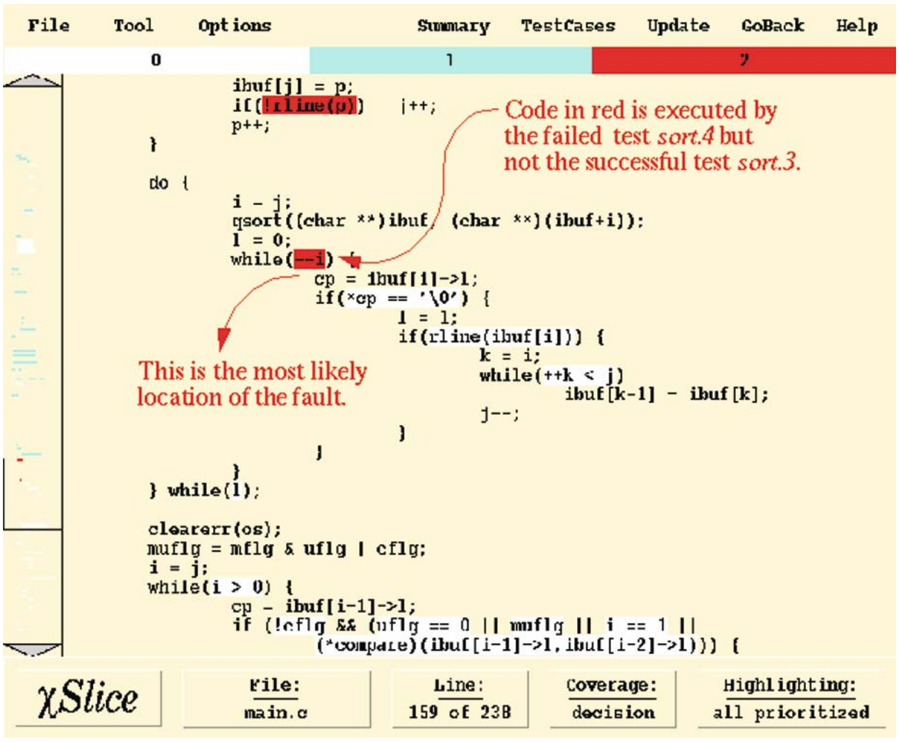
Fig. 4.32. Example: Linked data (forward memory slice without variables) and its forward reference pattern

ory slices. We can use a very similar technique to focus on certain subgraphs of a program. Remember that we can represent a program by its control-flow graph.

A static slice is the set of all program points that may affect the value of a particular output or an instance of a variable at a certain program point. Static slices are computed by static program analyses similar to those presented in Section 3.3.3.

In the following we shall look at slices computed for real runs of a program instead. An execution slice is the set of all program points executed for a given input. A dynamic slice is the set of all program points that, for a given input, actually affect a program point or an instance of a variable at a certain program point. As a consequence, a dynamic slice is a subset of an execution slice.

An important operation on dynamic slices is dicing. A dice is the set difference $A - B$ between two slices A and B. For example, X-Slice [Xsl] is a slicing and dicing tool for C programs. Here a dice contains those program points which were executed for a failing test case but not for a successful test case. In Fig. 4.33[15], the program points of the dice are shown in red in the program code.


File
Tool
Options
Summary
TestCases
Update
GoBack
Help
0
1
2
Code in red is executed by the failed test *sort.4* but not the successful test *sort.3*.
This is the most likely location of the fault.
χSlice
File:
main.c
Line:
159 of 238
Coverage:
decision
Highlighting:
all prioritized

```
                ibuf[j] = p;
                if(!rline(p))    j++;
                p++;
        }

        do {
                i - j;
                qsort((char **)ibuf, (char **)(ibuf+i));
                l = 0;
                while(--i) {
                        cp = ibuf[i]->l;
                        if(*cp == '\0') {
                                l = 1;
                                if(rline(ibuf[i])) {
                                        k = i;
                                        while(++k < j)
                                                ibuf[k-1] - ibuf[k];
                                        j--;
                                }
                        }
                }
        } while(l);

        clearerr(os);
        muflg = mflg & uflg | cflg;
        i = j;
        while(i > 0) {
                cp - ibuf[i-1]->l;
                if (!cflg && (uflg == 0 || muflg || i == 1 ||
                        (*compare)(ibuf[i-1]->l, ibuf[i-2]->l))) {
```

Fig. 4.33. Example of the use of X-Slice

4.6.2 Visualizing Test Case Results

Testing is the process of executing a program with the intent of finding errors.

[Mye79]

Testing is done by running a program with some input data and checking whether it produces the expected output or behavior. Test cases describe how to test a whole program, or modules or functions thereof. A test case identifies

[15] Reprint courtesy of Cleanscape Software Int'l

the state before the test is executed, the module or function to be tested, parameter values for the test, and the expected outcome. For simplicity, in the following we shall consider only test cases for a whole program. Let G be the grammar of a programming language. We model the execution of a program by a function $\mathsf{run}()$:

$$\begin{array}{ll} \mathsf{run} : L_G(S) \times I \to O & \\ O : \text{output values or final states} & \textbf{(Program execution)} \\ I : \text{input values or start states} & \end{array}$$

For our purposes, a test case (in, out) consists only of the input values and the expected outcome. We call a set of test cases a test suite $T \subseteq I \times O$. Now we can run a program s for each of the inputs in our test suite and look to see whether each run yields the expected outcome. As a result, we partition the test cases into a set of passing and a set of failing test cases:

$$\begin{array}{l} \mathsf{passed}(s,T) = \{(in, out) \in T | \mathsf{run}(s, in) = out\} \\ \mathsf{failed}(s,T) = \{(in, out) \in T | \mathsf{run}(s, in) \neq out\} \end{array} \qquad \textbf{(Actual test runs)}$$

To get from the failure to the error, i.e. from failing runs to wrong parts of the program (also called bugs or defects), we need information about individual program points in the program. For this purpose, we compute execution slices, i.e. the program points covered by an actual run:

$$\mathsf{coverage}(s, in) = \{p | \begin{array}{l}\text{program point } p \text{ is visited when} \\ \text{executing program } s \text{ with input } in\}\end{array} \qquad \textbf{(Coverage)}$$

Taking into account the failure or success of each test case, we get failing and successful execution slices:

$$\begin{array}{l} \mathsf{passed}(p, s, T) = \{(in, out) \in \mathsf{passed}(s,T) | p \in \mathsf{coverage}(s, in)\} \\ \mathsf{failed}(p, s, T) = \{(in, out) \in \mathsf{failed}(s,T) | p \in \mathsf{coverage}(s, in)\} \end{array}$$

Note that these sets are usually not disjoint, because program points such as those of the initialization of a program are executed by both failing and successful runs.

In Sect. 4.6.1, dicing was used to detect program points that might contain an error. For dicing, we need two slices, and a program point is either only in one of the two slices or in both. Now we have two sets of slices, and a program point can be in some of the slices in each of these sets. So we could use a three-valued approach, i.e. the program point is executed only by passed test cases, failing test cases or both, and consequently we could use three different colors, when visualizing the program points of the program.

In the system Tarantula [JHS02, Tar], the relative numbers of failing executions of program points are used as a basis for coloring the program points:[16]

[16] Gammatella, a successor tool of Tarantula, gathers information about errors and problems with already deployed systems instead of test cases [JOH04].

	year	1800	1998	2000	2002	2004	2006	2008	passed	failed	%passed	%failed	hue	brightness
1	int isLeapYear(int year)													
2	{ if (y%4==0)	1	1	1	1	1	1	1	4	3	1,00	1,00	0,50	1,00
3	if (y%100==0)	1	0	1	0	1	0	1	1	3	0,25	1,00	0,20	1,00
4	if (y%400==0)	1	0	1	0	0	0	0	1	1	0,25	0,33	0,43	0,33
5	return true;	0	0	1	0	0	0	0	1	0	0,25	0,00	1,00	0,25
6	else return true;	1	0	0	0	0	0	0	0	1	0,00	0,33	0,00	0,33
7	return false; }	0	1	0	1	1	1	1	3	2	0,75	0,67	0,53	0,75
	Result:	true	false	true	false	false	false	false						
	Passed:	no	yes	yes	yes	no	yes	no						

Fig. 4.34. Color coding of test case results per program point

$$\%\mathsf{passed}(p,s,T) = \frac{|\mathsf{passed}(p,s,T)|}{|\mathsf{passed}(s,T)|}, \qquad \%\mathsf{failed}(p,s,T) = \frac{|\mathsf{failed}(p,s,T)|}{|\mathsf{failed}(s,T)|}.$$

The system uses a color-coding scheme where the hue indicates the relative success rate of each statement, assuming a continuous color scale from 0 (red) through 1/2 (yellow) to 1 (green):

$$\mathsf{hue}(p,s,T) = \frac{\%\mathsf{passed}(p,s,T)}{\%\mathsf{passed}(p,s,T) + \%\mathsf{failed}(p,s,T)}. \qquad \textbf{(Hue)}$$

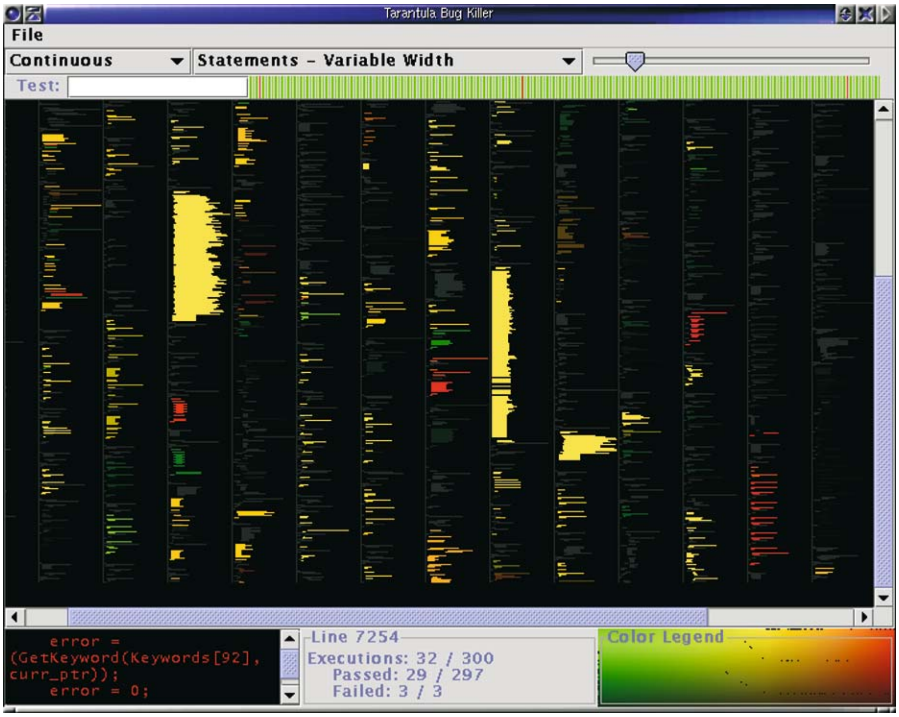
In addition to the hue, brightness is used to indicate the coverage of the program point; in fact, the maximum of the two percentages is chosen:

$$\mathsf{bright}(p,s,T) = max(\%\mathsf{passed}(p,s,T), \%\mathsf{failed}(p,s,T)). \qquad \textbf{(Brightness)}$$

Consider the example shown in Fig. 4.34. This shows the visualization of test cases for a buggy implementation of a function to determine leap years. Recall that a leap years are every year divisible by four except for those years which are both divisible by 100 and not divisible by 400. Line 5 is colored green because it is executed only in one successful run, while lines 3 and 6 are colored red because they are executed mostly in failing runs. The indentation suggests that line 6 is the `else` branch of line 3, but as the conditional in Java is right-to-left associative, line 6 is actually the `else` branch of the `if` statement in line 4. So the bug in the program is actually an instance of the dangling-else problem.

To display the test case results for large programs, Tarantula uses the line representation originally introduced by the SeeSoft system [ESJ92] (see also Sects. 1.4.3 and 5.1.1). Tarantula represents each character of the source code as a pixel. The hue and brightness of these pixels are computed as described above. The results for 300 test cases are shown in Fig. 4.35. Note that only a small fraction of the program is colored red.

In trying to produce our own examples to illustrate the Tarantula approach, we found that it does not work very well for loops or recursive functions – in other words, for the majority of programs. In the presence of loops and recursion, program points are visited several times for the same test case,



```
    error =
(GetKeyword(Keywords[92],
curr_ptr));
    error = 0;
```

```
Line 7254
Executions: 32 / 300
    Passed: 29 / 297
    Failed: 3 / 3
```

Fig. 4.35. Screenshot of Tarantula

but Tarantula would only take into account the fact that they have been visited at all. We can achieve better results if we use the absolute number of visits instead:

$$\begin{aligned}\mathsf{COVERAGE}(s, in) &= \{\,(p, n) | \text{program point } p \text{ is visited } n \text{ times}\\ &\qquad \text{when executing program } s \text{ with input } in\},\\ \mathsf{PASSED}(p, s, T) &= \{(p, n) |\, (in, out) \in \mathsf{passed}(s, T)\\ &\qquad \text{and } (p, n) \in \mathsf{COVERAGE}(s, in)\},\\ \%\mathsf{PASSED}(p, s, T) &= \frac{\sum_{(p,n)\in\mathsf{PASSED}(p,s,T)} n}{\sum_{(p_0,n)\in\mathsf{PASSED}(p_0,s,T)} n},\end{aligned}$$

where p_0 is the first program point of the program s. The functions $\mathsf{FAILED}()$ and $\%\mathsf{FAILED}()$ are defined analogously. Consequently, hue and brightness are now computed with these functions.

Figures 4.36 and 4.37 show test case visualizations for a buggy implementation of a function to compute the power k^n based on the observation that $k^n = k^{n/2} * k^{n/2}$ if n is even and $k^n = k * k^{n/2} * k^{n/2}$ if n is odd. We chose the ascending powers $2^0, \ldots, 2^6$ as test cases. Three computations produced wrong results. Using the first color-coding scheme, as shown in Fig. 4.36, lines 4 to

7 are colored slightly orange. Using our second scheme, as shown in Fig. 4.37, line 6 is colored green, indicating that it is unlikely to contain the bug, while the faulty return statement in line 7 is colored red.

	k	2	2	2	2	2	2	2	passed	failed	%passed	%failed	hue	brightness
	n	0	1	2	3	4	5	6						
1	int power(int k,int n)													
2	{ if (n==0)	1	1	1	1	1	1	1	4	3	1,00	1,00	0,50	1,00
3	return 1;	1	1	1	1	1	1	1	4	3	1,00	1,00	0,50	1,00
4	else { int t=power(k, n/2);	0	1	1	1	1	1	1	3	3	0,75	1,00	0,43	1,00
5	if ((n%2)==0)	0	1	1	1	1	1	1	3	3	0,75	1,00	0,43	1,00
6	return t*t;	0	0	1	0	1	1	1	2	2	0,50	0,67	0,43	0,67
7	else return k*t; } }	0	1	1	1	1	1	1	3	3	0,75	1,00	0,43	1,00
	Result:	1	2	4	4	16	8	16						
	Passed:	yes	yes	yes	no	yes	no	no						

Fig. 4.36. A buggy recursive program: Tarantula color-coding scheme

	k	2	2	2	2	2	2	2	PASSED	FAILED	%PASSED	%FAILED	HUE	BRIGHTNESS
	n	0	1	2	3	4	5	6						
1	int power(int k,int n)													
2	{ if (n==0)	1	2	3	3	4	4	4	10	11	1,00	1,00	0,50	1,00
3	return 1;	1	1	1	1	1	1	1	4	3	0,40	0,27	0,59	0,40
4	else { int t=power(k, n/2);	0	1	2	2	3	3	3	6	8	0,60	0,73	0,45	0,73
5	if ((n%2)==0)	0	1	2	2	3	3	3	6	8	0,60	0,73	0,45	0,73
6	return t*t;	0	0	1	0	2	1	1	3	2	0,30	0,18	0,62	0,30
7	else return k*t; } }	0	1	1	2	1	2	2	3	6	0,30	0,55	0,35	0,55
	Result:	1	2	4	4	16	8	16						
	Passed:	yes	yes	yes	no	yes	no	no						

Fig. 4.37. A buggy recursive program: alternative color-coding scheme based on total number of visits

4.6.3 Web Service Flow Patterns

Web services are currently praised as the solution to the development of distributed applications by allowing one to compose services in a standardized way [W3C].

As more Web services become available, applications get larger and more complex. The Web Services Navigator [IBMc, PKM06] is a visualization tool for understanding, debugging and analyzing the performance of these complex applications. To this end, execution traces in the form of Web service transactions are collected. A Web service transaction is a tree of messages and invocations that is initiated by a client. Thus, a transaction captures the flow of one service invoking one or more other services, which in turn may invoke other services and so on.

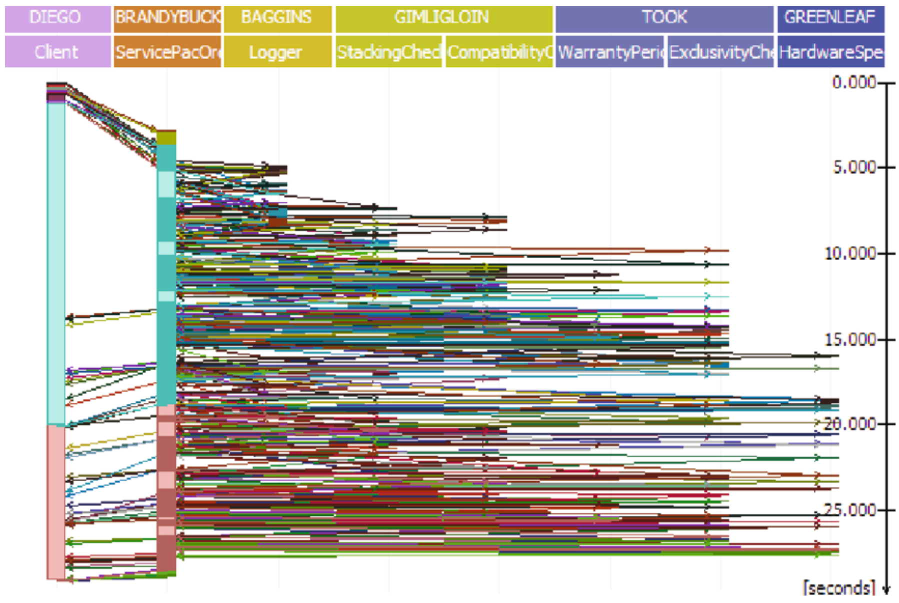
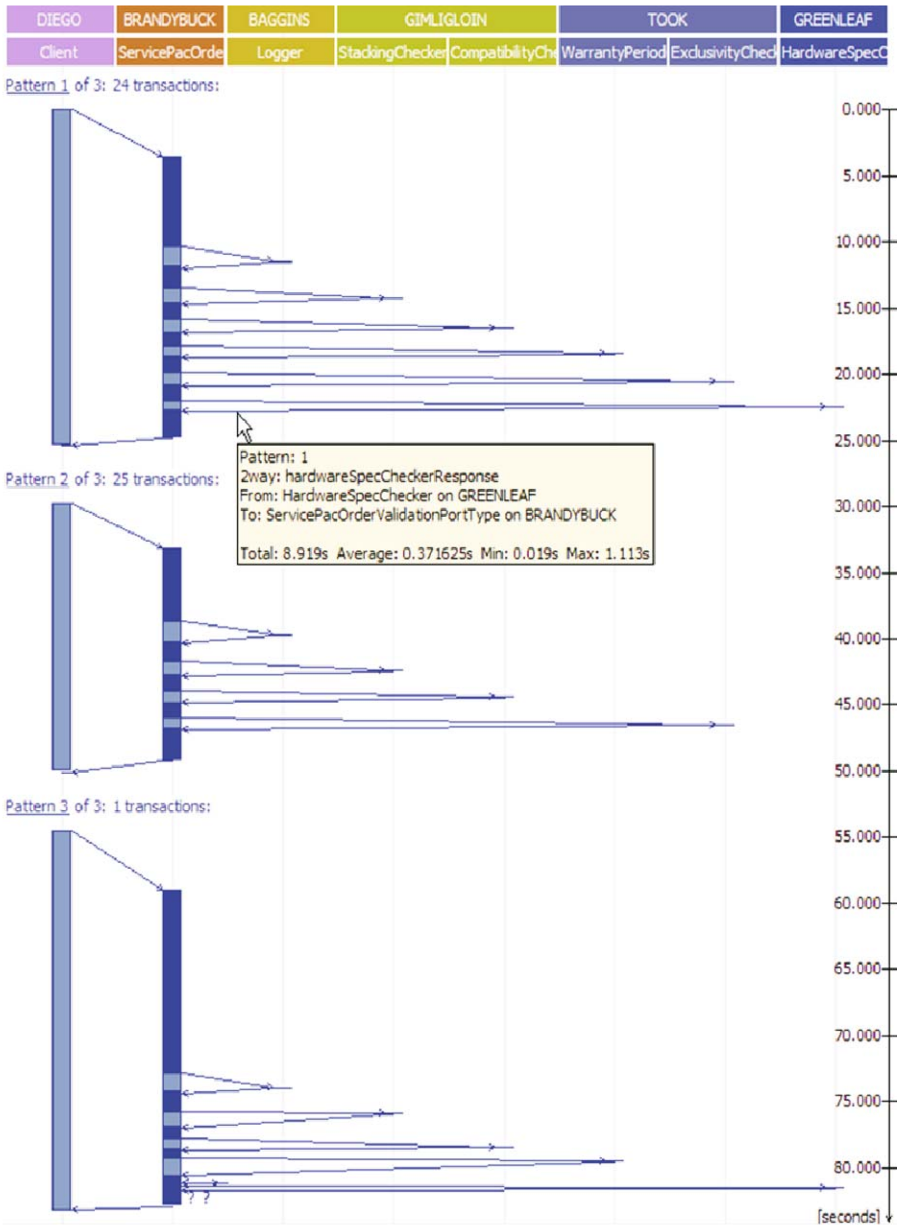


Fig. 4.38. Traces of 50 Web service transactions (Reprint courtesy of International Business Machines Corporation, copyright 2006 ©International Business Machines Corporation and ACM)

Transactions can be visualized using a sequence diagram-like representation (see Sect. 3.4.2), where vertical lines represent the different services involved. To test or tune the performance of a complex Web service, one typically needs to collect a large number of transactions. As shown in Fig. 4.38, simply drawing all transactions in a single diagram does not reveal any new insight other than revealing the fact that it looks like a mess.

By partitioning the transactions into groups of isomorphic tree shapes, and then further subdividing these groups based on matching node and edge at-



Pattern: 1
2way: hardwareSpecCheckerResponse
From: HardwareSpecChecker on GREENLEAF
To: ServicePacOrderValidationPortType on BRANDYBUCK

Total: 8.919s Average: 0.371625s Min: 0.019s Max: 1.113s

Fig. 4.39. Flow patterns computed for the 50 transactions (Reprint courtesy of International Business Machines Corporation, copyright 2006 ©International Business Machines Corporation and ACM)

tributes, the Web Services Navigator computes what we call Web service flow patterns. Figure 4.39 shows the three flow patterns computed for the 50 transactions shown earlier: 24 of the transactions match the first pattern, 25 the second, and only one matches the third pattern. In fact, this last pattern represents an erroneous situation, where a request was sent but no response was received. This is indicated by the two question marks shown at the lower end of the pattern. In addition, the last pattern differs from the other patterns in the timing of the events: the second Web service `ServicePackOrderValidation` takes more time before it calls another service.

4.7 Summary

While most dynamic program visualization techniques have been developed for algorithm animation and educational purposes, many of the techniques can be used for other purposes as well. For example, dynamic visualizations of the software architecture show the dynamic behavior of a software system at the level of its components by gathering information at run time and enriching the static architecture diagram or producing behavior diagrams such as sequence diagrams.

We have discussed several more or less invasive dynamic acquisition methods which are mostly based on instrumenting the source, intermediate, or binary code. While invasiveness is not usually an issue for algorithm animation, it is critical for analyzing, debugging, or optimizing software such as embedded systems or parallel programs.

In general, dynamic information can be shown by accumulating information over time, by spatial projection on the time axis, or by animation, i.e. the time dimension of the data is represented by time when the user views the visualization. Algorithm animation applies this latter technique.

Abstract algorithm animation uses static program analysis instead of dynamic data and is able to show all possible executions of a program by executing the program with abstract values instead of real data. Abstract values represent possibly infinite sets of real data that have some common properties, for example the set of all positive numbers or the set of all data structures that have the same shape.

Combining both static and dynamic analysis is an active field of research which will certainly enable new kinds of program visualizations.

Visual debugging tools display either the program state or the program code. While the program state is often shown as a graph, the program code is typically shown as color-coded text. Graphical representations of program code such as those discussed in Chap. 3 are presumably not widely used because programmers are too familiar with the textual representation of their programs.

As the program state tends to be very large, visual debuggers usually let the programmer interactively unfold those parts of the program state he or she is interested in. We have also discussed some more automatic approaches, that can be used to deal with large program states: differences between memory graphs can be help to identify possibly wrong program states, and reference patterns represent several objects of the same class by a single node.

Just like the program state, the source code of real applications tends to be very large. To automatically detect buggy program parts, the coverage information from several failing and successful program runs can be combined to compute for each line of code a heuristic probability that it contains the bug. This probability is then visualized using a continuous color scale.

Finally, to visualize large amounts of execution traces, one can compute groups of traces with similar shape. Partitions which contain one or few traces are likely to represent problematic situations.

Exercises

Exercise 1: Look for algorithm animations on the Web. You may want to use the following URLs as starting points: `http://www.cs.hope.edu/~alganim/ccaa/`, `http://www2.hig.no/~algmet/animate.html`, and `http://www.animal.ahrgr.de/Anims/animations.php3`. How have the developers of these animations addressed the design issues discussed in Sect. 4.4.5?

Exercise 2: The goal of this exercise is to produce an algorithm animation of the SIT problem (scheduling of independent tasks): Given m machines and n jobs of lengths $a_1, \ldots, a_n$, allocate the jobs to the machines such that the maximal span (the span of the longest-working machine) is minimal.

There are exist, for example, the following three different algorithms to solve the problem:

1. Heuristic: allocate the jobs in order $a_1, \ldots, a_n$. Each job is allocated to the machine with the present minimal span.
2. Heuristic: sort the jobs by length in descending order. Then use algorithm 1.
3. Use a branch-and-bound algorithm (or just a complete search) to compute the optimal solution.

Implement at least two algorithms in your favorite programming language and annotate them with print statements to produce a SAMBA trace. The animation should be a good "visual explanation" of the algorithm. To make things easier you can assume $m = 3$. If this is too easy, try to solve the more challenging problem for arbitrary $m > 1$.

For this exercise JSAMBA, a reimplementation of SAMBA in Java, is recommended. The trace can be passed to JSAMBA using cut and paste into the input field of the JSAMBA applet.

Exercise 3: Instead of SAMBA, one can also use SVG (Scalable Vector Graphics). Write a converter that translates SAMBA animation plans into SVG code. For circles, texts, and rectangles, as well as for animations that move these objects to given coordinates, the translation is not too difficult. The following examples should give you some ideas about how the translation should work. Note that the origin of the SAMBA coordinate system is in the lower left corner, whereas in SVG it is in the upper right corner. Assuming an SVG window of size $w \times h$, a SAMBA coordinate (x, y), where $x, y \in [0, 1]$, corresponds to the SVG coordinate $(x * w, h - y * h)$. The examples are

JSamba: `rectangle 3 0.1 0.9 0.1 0.1 blue solid`

SVG:
```
<rect id="object_3"
      x="10" y="0" width="10" height="10"
      fill="blue" opacity="1.0"/>
```

JSamba: `moverelative 3 0.05 -0.4`

SVG:
```
<animate id="step_2x" xlink:href="#object_3"
         attributeName="x" attributeType="XML"
         begin="step_1x.end" dur="5s" fill="freeze"
         additive="sum" accumulate="sum"
         from="0" to="5" />
<animate id="step_2y" xlink:href="#object_3"
         attributeName="y" attributeType="XML"
         begin="step_1y.end" dur="5s" fill="freeze"
         additive="sum" accumulate="sum"
         from="0" to="40"/>
```

Exercise 4: Given the following program, find a minimal set of test cases which cover all paths. (Hint: draw the control-flow graph.)

```
while ((a+b+c)<100)
  { if ((a-b)<c) { b=b+1; }
    if ((b+c)==a) { c=c+2; }
    else {
      if (a==b) { a=a+1; }
    }
    a=a+1;
  }
```

Exercise 5: On the basis of the memory graph in the upper part of Fig. 4.32 compute and draw the backward memory slice starting at the object of type `PO-Box`.

For the following exercises, you have to "execute" the following program by paper and pencil for several test cases. For each program point p and each test case q, you have to count the number of times p is executed during failing runs, $f_{p,q}$, and successful runs, $s_{p,q}$. In addition, we are interested in whether a program point was executed at all during a failing run, i.e. $F_{p,q} = 1$ iff $f_{p,q} > 0$ otherwise $F_{p,q} = 0$. Analogously, for successful runs we define $S_{p,q} = 1$ iff $s_{p,q} > 0$, otherwise $S_{p,q} = 0$.

```
p |  program points                   q |  test cases
--+---------------                    --+------------
1 |  running=true;                    1 |  x=1 , y=0
2 |  while (x>0)                      2 |  x=2 , y=2
3 |     if (x%y==0)                   3 |  x=3 , y=2
4 |        x=x-1;                     4 |  x=3 , y=1
  |       else                        5 |  x=3 , y=3
5 |        y=y-2;                     6 |  x=5 , y=3
6 |  running=false;                   7 |  x=5 , y=4
```

Note that % represents the modulo operator in Java and that it throws division-by-zero exceptions.

Exercise 6: Compute the numbers $f_p = \sum f_{p,q}$, $s_p = \sum s_{p,q}$, $F_p = \sum F_{p,q}$, and $S_p = \sum S_{p,q}$. Now there are various relations for each program point that you are going to represent: f_p/s_p, $f_p/(f_p+s_p)$, F_p/S_p, and $F_p/(F_p+S_p)$. Instead of using color coding, draw for each of these relations a 3×2 matrix M such that the entry $M_{i,j}$ shows the value of the relation for the program point $(i-1)*3+j$.

Exercise 7: For each of the relations, suggest a way of using colors and possibly other means such that not only the value of each matrix entry is visualized but also its support. A simple color-coding scheme $\sigma : \mathcal{R} \rightarrow C$ would map the values to a color space, for example $C = \{\mathsf{red}, \mathsf{orange}, \mathsf{yellow}, \mathsf{green}, \mathsf{blue}\}$, and we would color the matrix entry of p with $\sigma(f_p/s_p)$. How can you, in addition to the value of each matrix element, also visualize its support or evidence; for instance, $f_p = 5$ and $s_p = 20$ has more support than $f_p = 1$ and $s_p = 4$, although $5/20$ is equal to $1/4$.

Exercise 8: For the relation $f_p/(f_p + s_p)$, draw three bar charts that differ in the way bars are colored, placed, sized, overlap, etc. Try to come up with designs that highlight program points that are very likely to cause the error.

Exercise 9: The Web Services Navigator shows the dynamic invocation tree such that the duration of each invocation is represented by the length of a vertical bar. How could one show the same information using a treemap? Draw an example. What are advantages and disadvantages of this representation? Can you think of other tree representations that would show the timing information in a suitable way?

5 Visualizing the Evolution of Software Systems

I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection.
(Charles Darwin, The Origin of Species)

Looking at the multitude of research papers that have the term "software evolution" in their titles, one could easily conclude that software evolution has just become a synonym for the software development process. A closer look reveals that many authors use the term to emphasize that a software system changes throughout its lifetime. As Frederick Brooks put it in his famous article "No silver bullet" [Bro87],

All successful software gets changed.

A software system has to be adapted to the changing needs and environment of its users. This is usually only necessary for successful software. Economically, maintaining unsuccessful software does not make sense. If successful software did not change, the software company would eventually lose customers to its competitors.

So *change* plays a major role in software evolution. In general, there are two areas within software evolution:

Design for change: Here the goal is to devise rules and architectures to facilitate the task of changing a software system over time, addressing issues such as reconfiguration, adaptation, extension, debugging, optimization, evaluation (measuring changeability), and project management.

Analysis of software histories: During the lifetime of a software system, many versions will be produced. Analyzing the source code of these versions, as well as documentation and other meta-information, can reveal regularities and anomalies in the development process of the system at hand.

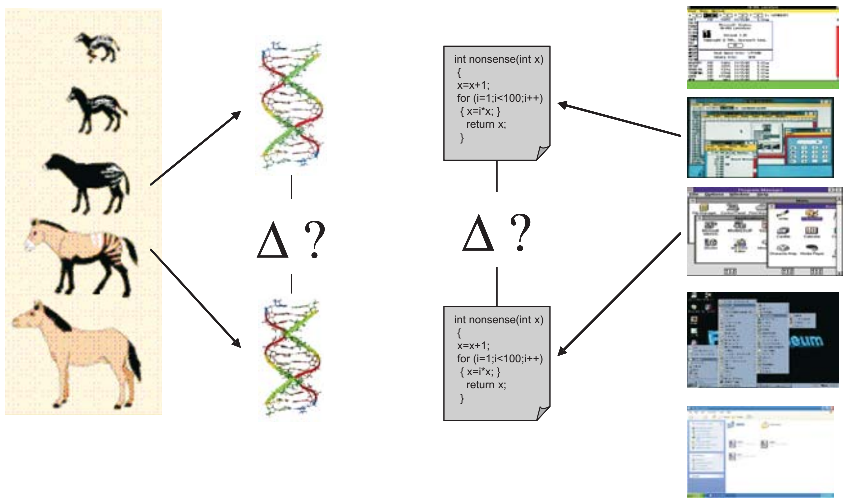
In some cases the two areas complement each other; for example, on the one hand software histories can be used to validate suggested rules or architectures, and on the other hand they can be used to discover them. Some very general rules have been suggested on the basis of some case studies by Lehman, basically stating that the size, functionality, and complexity of a software system increases over time, while its growth rate and quality decrease [BL76, Leh80]. Independently of each other Basili and Perricone, and, independently, Möller performed case studies to validate the rule that smaller changes have a higher error density than larger ones [BP84, MP95, FO00]. Intuitively, programmers tend not to spend the same amount of time on understanding the context and, in particular, the surrounding source code of smaller changes as they would do for larger changes, although in both cases basically the same level of understanding is required. For many of the rules published in the software engineering literature, the relevance with to other projects is unclear: either the rules are too general, or the results of the case studies cannot be transferred, because the constraints of the case studies are not well documented. To remedy this situation, the development of tools for validating rules on the basis of the history of ones own project or even for discovering new project-specific rules is an active area of research.

As shown in Fig. 5.1, evolutionary biologists look for differences in genes. Finding common patterns in these differences enables them to formulate rules about how a certain species evolves or even how species in general evolve. Genes are often called the programs of life. By use of this analogy, software evolution researchers can use methods similar to those of evolutionary biologists.

In this chapter we shall address mainly the visual analysis of software histories. In particular, we shall look at three kinds of visualizations. The first is those which visualize metrics on a flat representation of the software. The second is those that also show structural information. Most techniques of these two types show the changes of the metrics and/or of the structure over time. The third is techniques that extract recurring patterns from the software history using data-mining and visual data-mining techniques.

5.1 Visualizing Changes in Software Metrics

Software engineers are faced with a plethora of general requirements related to software quality. Software should be be easy to use, fast, correct, reliable, secure, extendable, and maintainable. As a consequence, software engineers have come up with many software metrics to quantify various properties of software and relate them to the above-mentioned aspects of quality. Some of the very basic measures are the size of a module, the run time of a program, the number of changes or bug fixes in a module or even a single line of code, the number of programmers that have made a change, and the depth of nested blocks. Studies that found that a high McCabe complexity (see Sect. 3.4.3 implied many bugs, have led to the adoption of the McCabe complexity as an important quality indicator in industrial software development. Recently, there have been several case studies [GKMS00] which found that



```
int nonsense(int x)
{
x=x+1;
for (i=1;i<100;i++)
 { x=i*x; }
  return x;
}
```

```
int nonsense(int x)
{
x=x+1;
for (i=1;i<100;i++)
 { x=i*x; }
  return x;
}
```

Fig. 5.1. Analogy between biological and software evolution

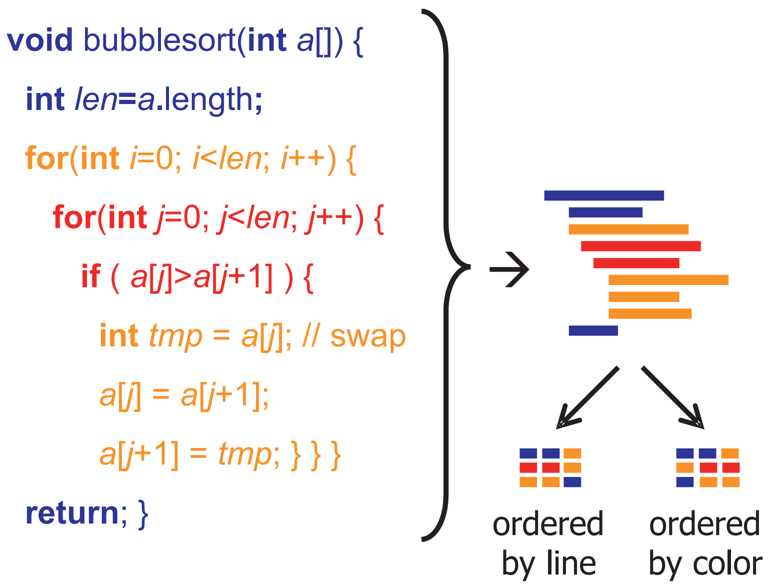
CC was strongly correlated to simpler measures such as number of functions or variables, or of lines of code. Thus, these simpler measures could be used as quality indicators, as well.

5.1.1 SeeSoft

The SeeSoft system introduced a space-filling visualization for metrics related to the individual lines of code such that modules with up to one million lines of code can be displayed and fitted on the screen [ESJ92]. The goal of *space-filling visualizations* is to convey as much information as possible with as few pixels as possible. SeeSoft provides several color-coded visualizations, as shown in Fig. 5.2:

Textual representation: Each line of text is shown as colored text. All characters in the line have the same color.
Line representation: Each line of text is represented by a colored line of pixels.
Pixel representation: Each line of text is represented by a one pixel (or a few). The pixels representing lines can be ordered according to either the order of the lines in the text or their color.
File summary representation: Every file is represented by a box. There are four different sizes (quartiles).

The metrics that have been visualized with this system are code age, number of bug fixes, depth of nested blocks, and profiling data. For example, one can use the heated-object color scale and the code-age metric. In this case red (hot) is used for lines that have been changed recently, whereas blue (cold) indicates lines that have not been changed for a long time. With this color code,



```
void bubblesort(int a[]) {
 int len=a.length;
 for(int i=0; i<len; i++) {
   for(int j=0; j<len; j++) {
     if ( a[j]>a[j+1] ) {
       int tmp = a[j]; // swap
       a[j] = a[j+1];
       a[j+1] = tmp; } } }
 return; }
```

Fig. 5.2. Representation by color-coded lines and pixels

the user can easily see what parts of a program are currently under development. Another nice feature is that common code and operating-system-specific code can be colored differently.

The SeeSys system that was developed later on [BE95] provides a hierarchical representation based on treemaps (see Sect. 2.3) of subsystems, directories, and files. At each level, the value of the metric represented is computed from the metrics of the elements of the level below, for example the number of noncommentary source lines in a directory is the sum of the numbers of the files and directories contained therein.

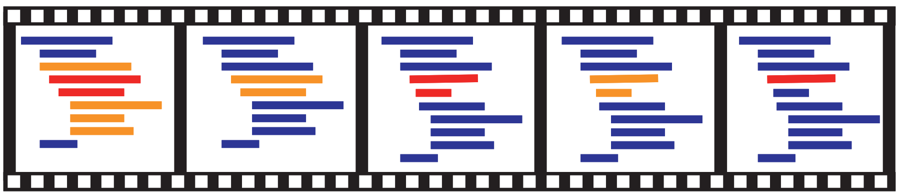
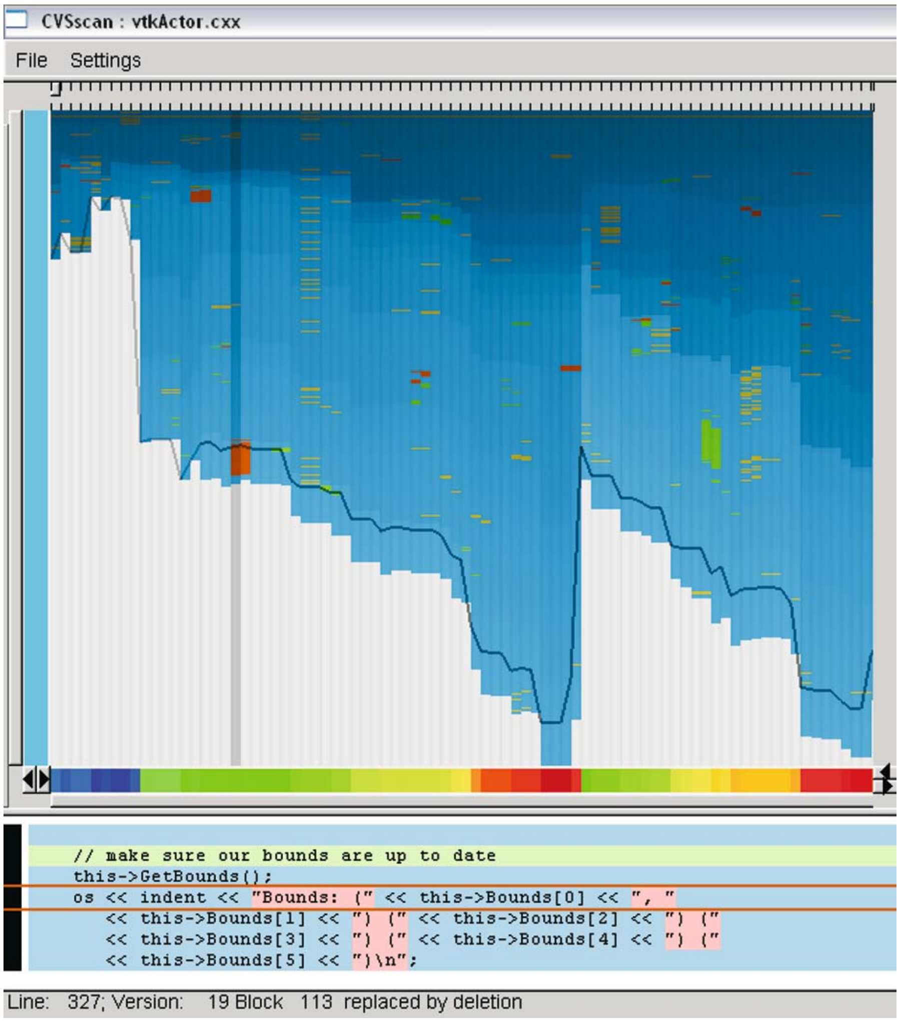


Fig. 5.3. Color-coded line representations of successive versions of a file

Both SeeSoft and SeeSys can be used to animate the evolution of a system by showing representations of the same parts of the system in a sequence of states of development. Figure 5.3 shows line representations of successive versions of a file. Assuming that the color indicates the age of the last change, i.e. red indicates that the line was changed in going from the previous to the current version, we can easily see in this example that the fourth line is a hot spot, i.e. it is often changed, while most of the other lines were changed once at the beginning of the development.



```
// make sure our bounds are up to date
this->GetBounds();
os << indent << "Bounds: (" << this->Bounds[0] << ", "
   << this->Bounds[1] << ") (" << this->Bounds[2] << ") ("
   << this->Bounds[3] << ") (" << this->Bounds[4] << ") ("
   << this->Bounds[5] << ")\n";
```

Line: 327; Version: 19 Block 113 replaced by deletion

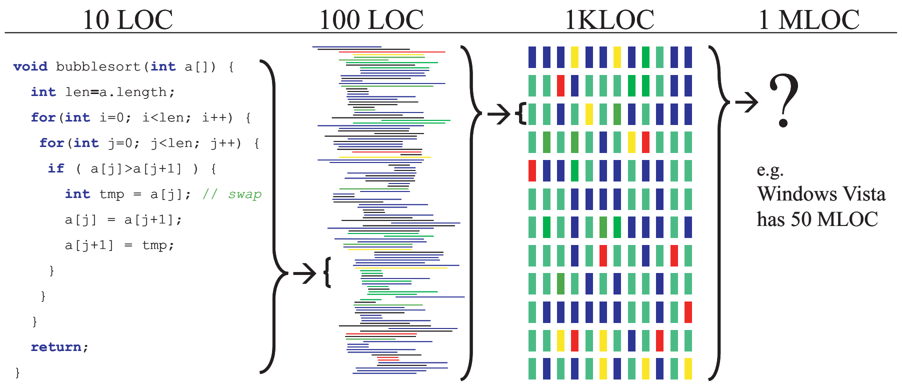
Fig. 5.4. CVSScan: evolution of a single file

As shown in Fig. 5.4, CVSScan [VTvW05] shows the evolution of a single file. Each version of the file is drawn in a column using the SeeSoft line presentation. Successive versions of the file are shown in successive columns. In Fig. 5.4, line 327 of version 19 of the file `vtkActor.cxx` has been selected and is shown in detail in the lower part of the window. The dark line in the upper part shows the trajectory of the selected line of code in the successive versions.

Aspect Browser [GYK01] applies the SeeSoft technique to the visualization of aspects. Basically, lines of code that belong to an aspect pattern described by a regular expression are drawn in a color associated with that aspect. Aspect conflicts, i.e. lines that belong to more than one aspect, are colored red.

To visualize and support collaborative, distributed software development, Augur [FD04] extends the line-oriented view of SeeSoft with author and syntactical information and combines it with other views, such as a graph view showing relations between different developers.

Recently a 3D extension of the SeeSoft approach called SV3D was developed [MFM03, Mar]. It represents each line of code by a three-dimensional box. In addition to color coding, the height of the box is used to visualize a metric, e.g. the color of a box encodes the programmer, while its height encodes the number of bug fixes.



```
void bubblesort(int a[]) {
  int len=a.length;
  for(int i=0; i<len; i++) {
   for(int j=0; j<len; j++) {
    if ( a[j]>a[j+1] ) {
      int tmp = a[j]; // swap
      a[j] = a[j+1];
      a[j+1] = tmp;
    }
   }
  }
  return;
}
```

Fig. 5.5. The limits of space-filling code visualization

On an ordinary computer screen with, for example, a resolution of 1280×1024 pixels, the SeeSoft pixel representation allows 1.3 million lines of code to be fitted on the screen. Currently, Debian GNU Linux has more than 200 million lines of code, and, allegedly Microsoft Windows Vista has 50 million lines of code. So we would need 200 computer screens for Linux and 50 computer screens for Windows to get an overview of the system. In other words, while the space-filling approach of SeeSoft works for many real software projects, it does not scale to some of today's really big software projects, as shown in Fig. 5.5.

5.1.2 Revision Towers

Revision towers are a tool for visualizing the contents of an RCS or CVS source code repository [TM02] containing C programs, i.e. files with extensions `.c` and `.h`. Each tower represents the history of a pair of files (header and implementation). In the tower, each version of a file is shown as a rectangle, which is sized and colored according to various properties, for example its width is related to the size of the version and its height to its lifetime, while the color indicates the programmer who checked it into the repository (see Fig. 5.6). Instead of showing all versions in the tower at once, one can use animation to see when new versions were checked in.

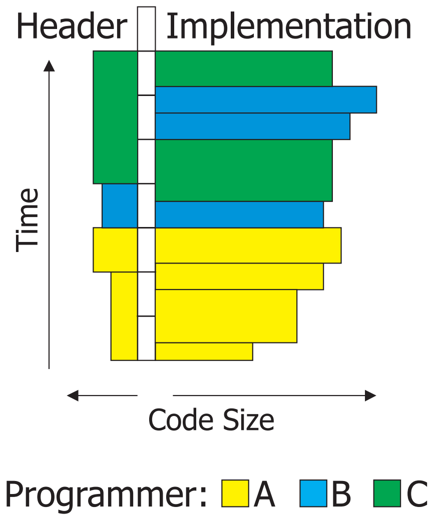


Fig. 5.6. Revision tower: evolution of a C file and its header file

5.1.3 The Evolution Matrix

Rectangles are also used by a visualization called the evolution matrix [Lan01] to show the evolution of object-oriented systems, i.e. sets of classes. Here, versions of classes are represented by rectangles. The width and height are used to indicate different metrics. For example, the width can be used to show the number of methods, while the height shows the number of member variables in the class. The evolution matrix consists of several rows. Each row contains rectangles representing different versions of the same class at different times. A column contains the versions of different classes at the same point of time.

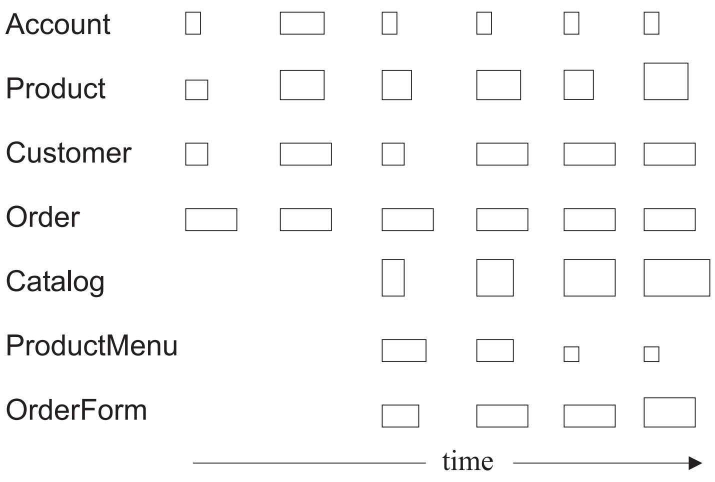


Fig. 5.7. Evolution matrix of a small system

The evolution matrix allows one to make observations about the evolution of both the whole system and a single class. On the system level, for example, one can observe the change of the system size over time caused by addition and removal of classes and identify phases of growth, stagnation, and shrinking. In the example shown in Fig. 5.7, there is a major leap from version 2 to 3.

On the class level, it allows one to categorize individual classes. With the help of this visualization, Lanza identified several categories [Lan01]. The names of these categories are the following, taken from astronomy to emphasize a property by means of analogy to an astronomical phenomenon: pulsar, supernova, white dwarf, and red giant. To illustrate these metaphorical categories, we briefly discuss two of them below:

Pulsar: A pulsar is a class which alternately increases and decreases in size (or with respect to some other metric), for example the class `Product` in the example. Extending the functionality typically increases the size of the class, whereas restructuring decreases the size. The pulsar classes are those which are at the center of the development of a system.

White dwarf: A white dwarf is a class that becomes smaller and smaller, for example the class `ProductMenu` in the example. By and by, its functionality is moved to other classes. Eventually, the class might become superfluous.

5.2 Visualizing Software Archives

Both industrial and open-source projects keep track of versions and changes using configuration management systems [CW98], for example RCS, CVS and

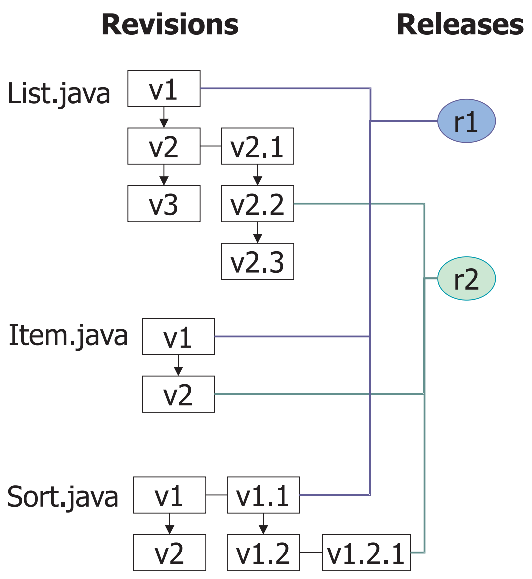


Fig. 5.8. 2D representation of three revision trees and releases

Subversion. Other tools keep track of additional information, for example bug databases. The information stored by a configuration management system and related tools is called a software archive. A software archive provides the history of a software system.

The central part of the information in a software archive is *versions* of files, including source code, documentation, and specifications (e.g. UML diagrams). Most archives do not store the complete versions of the source code files, but only initial versions and differences between subsequent versions, often called *deltas*. In addition, for each version, they store a comment or log message, a time stamp, and the name of the person (author ID) who checked the version into the configuration management system. For each version, there can be several alternative subsequent versions, called variants, resulting from parallel development. Several variants can be joined together into a single successor version. As a consequence, the versions and their successor relations form an acyclic graph, called the version graph.

A *configuration* is a set of versions. For each file, it contains at most one version. Usually, the versions in a configuration can be used to build a running implementation of the system. Often the term *release* is used as a synonym for the term configuration, but mostly it is used to refer to configurations that have been released for users.

Meta-information such as makefiles, to-do lists, bug reports, and test results is stored either in the configuration management system directly, or in a separate database.

Tools such as WinCVS [Str] and VRCE [Dur] show the version graph in a vertical tree representation, which is sometimes called an explorer view (see Fig. 5.8). In the example shown here, the versions are displayed as boxes and the releases as ellipses.

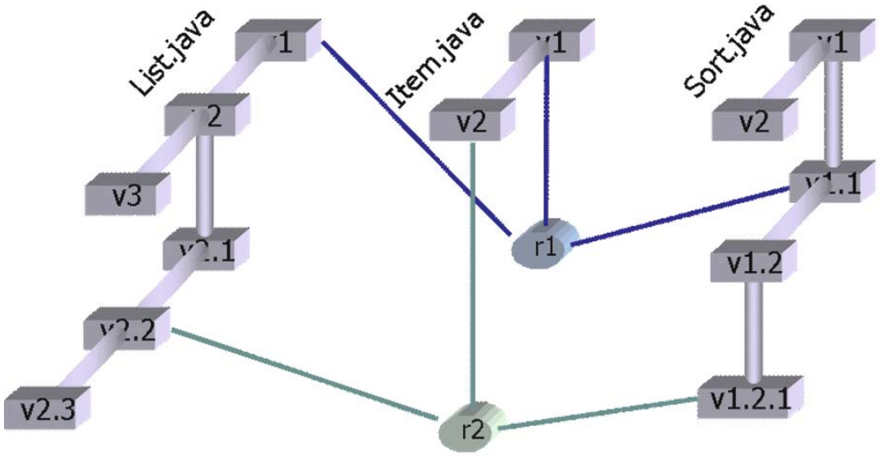


Fig. 5.9. 3D representation of revision tree and releases

In 3D visualizations such as that shown in the paper [KC97] on VRCS, the Z axis can be used as a time axis. Figure 5.9 shows a 3D representation of the same revision tree as before.

Instead of representing all versions, Gall et al. visualized only the structure of the releases [GJR99], i.e. the hierarchical decomposition of the system into subsystems, the subsystems into modules, and the modules into programs (actually versions of files). The color of each version indicates the release number of the release, which indicates when it was changed for the last time (see Fig. 5.10). The 2D visualization in the upper left of Fig. 5.10 only shows the versions; here we can easily see that there are two files that have never been changed after the first release.

5.3 Visualizing Structural Change

While there are many tools to visualize the structure of programs as graphs, tools to visualize changes of these structures over time are rare. As the algorithms for graph animations [DG02, GBPD04] become more mature, we shall hopefully see more in the future.

The GEVOL system uses a force-directed layout to draw call graphs, control-flow graphs, and inheritance graphs of Java programs [CKN+03]. In

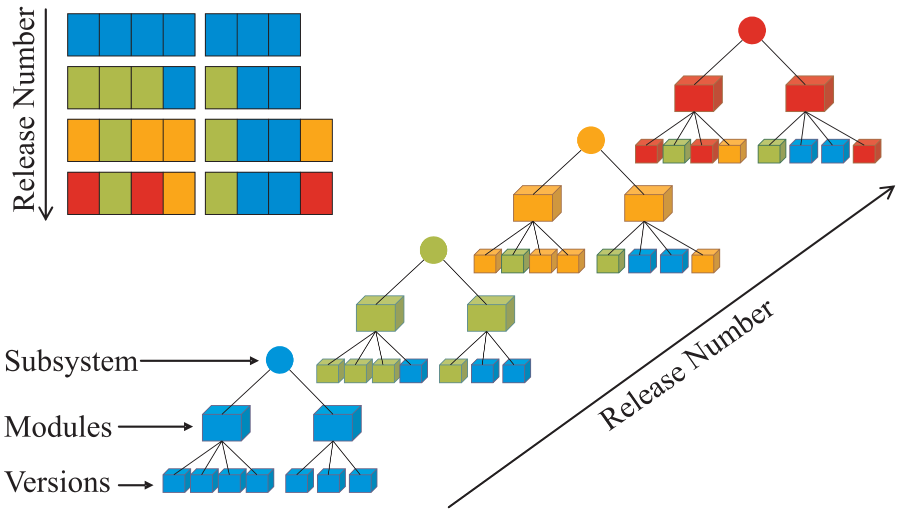


Fig. 5.10. 3D visualization of release history

these graphs the color of the edges indicates the age, i.e. when was it last changed. Instead of using one color scale, GEVOL uses two color scales for two different developers, so that one can see who made the last change. In these color scales aging is reflected by progression from the user's color, for example red or yellow, to blue. GEVOL uses real time instead of logical time and thus computes one graph per day. Animation shows successive graphs using linear interpolation for smooth transitions (see Fig. 5.11).

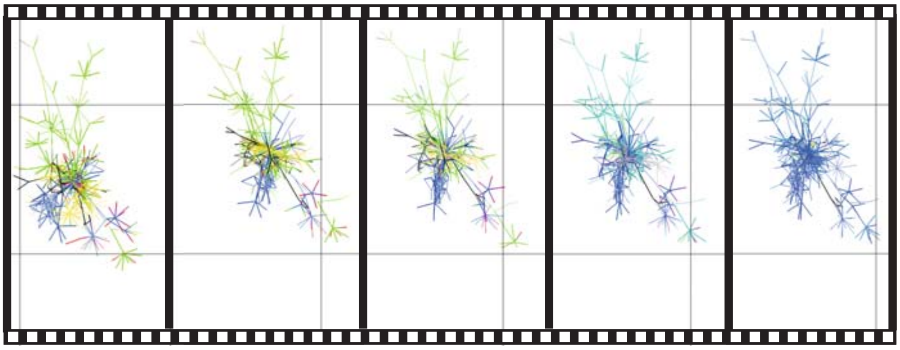


Fig. 5.11. GEVOL: inheritance graphs of successive versions of a Java program

5.4 Visualizing Evolutionary Coupling

Larger software systems consist of various libraries, modules, classes, and functions, in decreasing order of granularity. If we look, for example, at the function level we shall find that functions call other functions. If a function is called by another function, the two are coupled. So, as a simple metric for coupling, we can just define the coupling of a function as the number of functions it calls and the number of functions it is called by. Coupling measures the complexity of the dependencies and, as a consequence, the more dependencies there are the more difficult it is to understand or replace a function. We can define coupling on other levels in a similar way. As a rule of thumb, software developers should strive for low coupling, in particular at the higher levels of granularity such as classes or modules.

In this section we look at a different form of coupling, which we call evolutionary coupling[1] to distinguish it from the classical kind of coupling described above usually used in software engineering. We say that two software artifacts such as functions, classes, or files, but also images and documentation, are evolutionarily coupled if they are changed at the same time [ZDZ03]. Coupling information can even be used to implement recommender tools that suggest source code changes to the programmer [ZWDZ05].

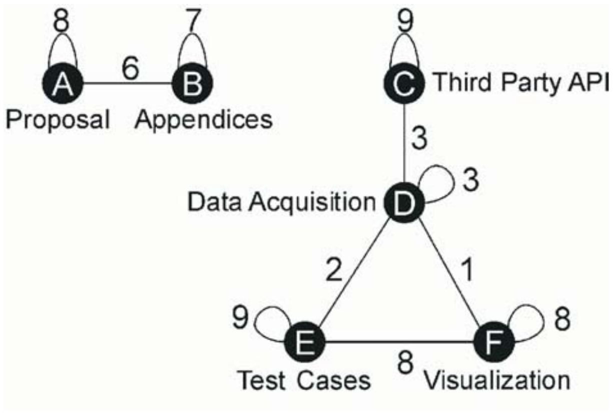


Fig. 5.12. Support graph of a small project

Let A be a set of software artifacts and S_{a_1,a_2} the number of times the two software artifacts a_1 and a_2 have been changed together. We call S_{a_1,a_2} the support count of the evolutionary coupling of a_1 and a_2. On the basis of this information we can easily compute a support graph

$$G = \{\mathcal{A}, \{(a_1, a_2) \mid a_1, a_2 \in \mathcal{A} \text{ and } S_{a_1,a_2} > 0\}\}.$$

[1] Other authors also use the terms logical coupling, change coupling, and co-change relation for the same concept [GHJ98, GJK03a, BN05].

Figure 5.12 depicts the support graph of a small project. There are two important aspects of the graph:

Existence of edges: Entities that are related are connected by edges, and sets of entities with many interrelations form clusters. In the example graph we see, for example, that D, E, and F are related to each other.

Absence of edges: The absence of edges indicates that entities are not related. In the example graph we can immediately see that there are two unconnected subgraphs.

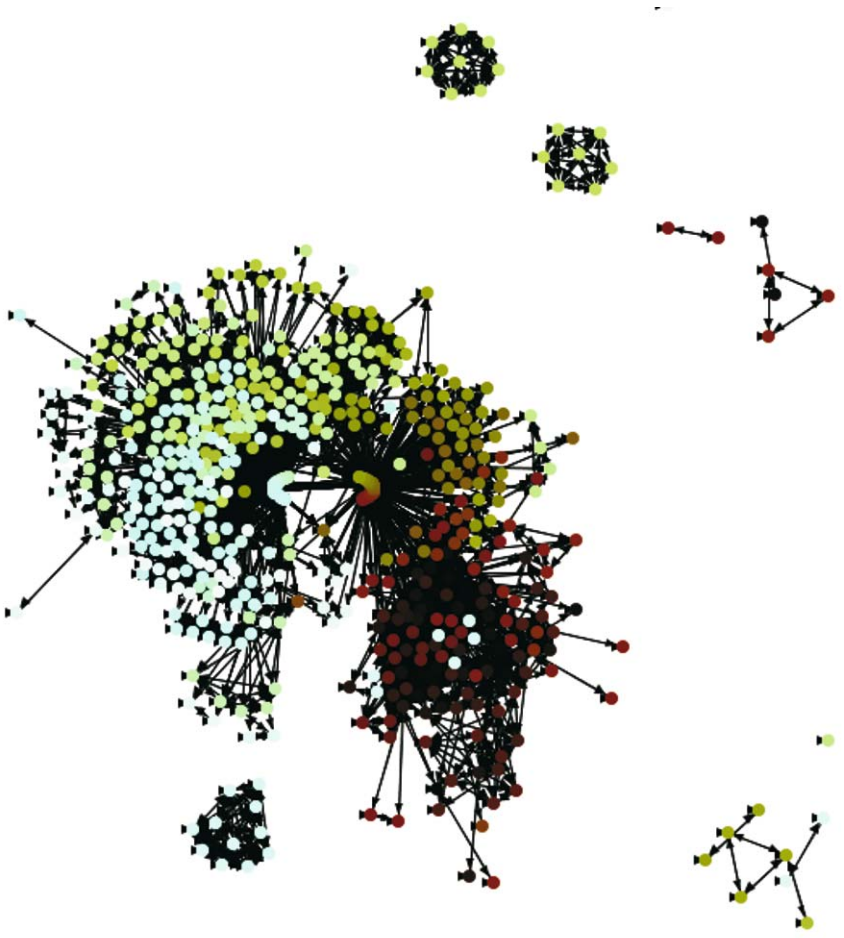


Fig. 5.13. Support graph of Mozilla Firefox

Figure 5.13 shows the support graph of the Mozilla Firefox browser project. Instead of showing the support values as numbers, we have used a force-directed graph layout with the support values as edge weights. As a result, strongly coupled items are drawn next to each other. Here the software artifacts are files, and we have used color coding to indicate whether two artifacts are close in the directory hierarchy.

In this example, there is a large cluster in the middle of the graph. The red part of this cluster corresponds to the `/base` subdirectory. In this red part, there are also a few light blue nodes. These outliers are

```
components/prefwindow/locale/pref-advanced.dtd
components/prefwindow/locale/pref-appearance.dtd
components/prefwindow/locale/pref-applications-edit.dtd
components/prefwindow/locale/pref-applications.dtd
```

A closer look reveals that only the file `pref-advanced.dtd` is related to files in the `/base` directory, whereas the remaining three DTDs are related only to the first one.

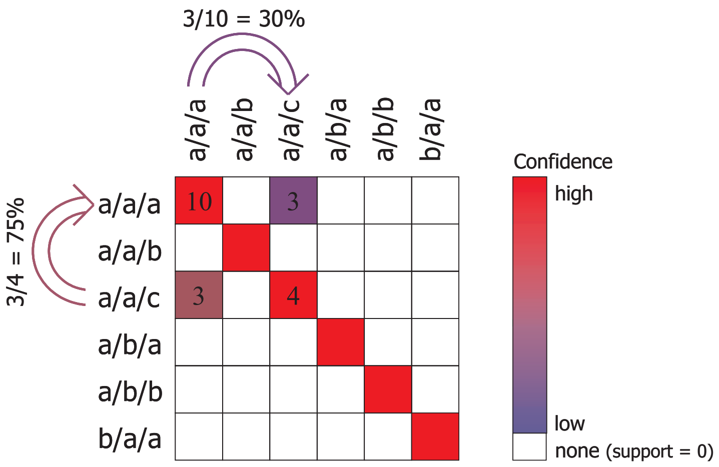


Fig. 5.14. Sorted axes and color-coded confidence (the numbers indicate the support count)

Figure 5.15 shows how the coupling information can be visualized using a matrix visualization that we call a pixelmap in what follows. In this example, the color of the pixel at position (x, y) represents the the number of times that two files f_x and f_y have been changed together relative to the total number of times the file f_x has been changed. We call this number the confidence of the coupling. More precisely, the confidence that a_2 has been changed whenever a_1 has been changed is defined as

$$C_{a_1,a_2} = \frac{S_{a_1,a_2}}{S_{a_1,a_1}}.$$

Note that whereas the support count is symmetric, the confidence is not, i.e., in general, $C_{a_1,a_2} \neq C_{a_2,a_1}$.

From the pixelmap, the developer can see how strongly different files are coupled. High confidence values are shown as red pixels, and low values as blue pixels. White indicates that there is no coupling, i.e. the support count is zero.

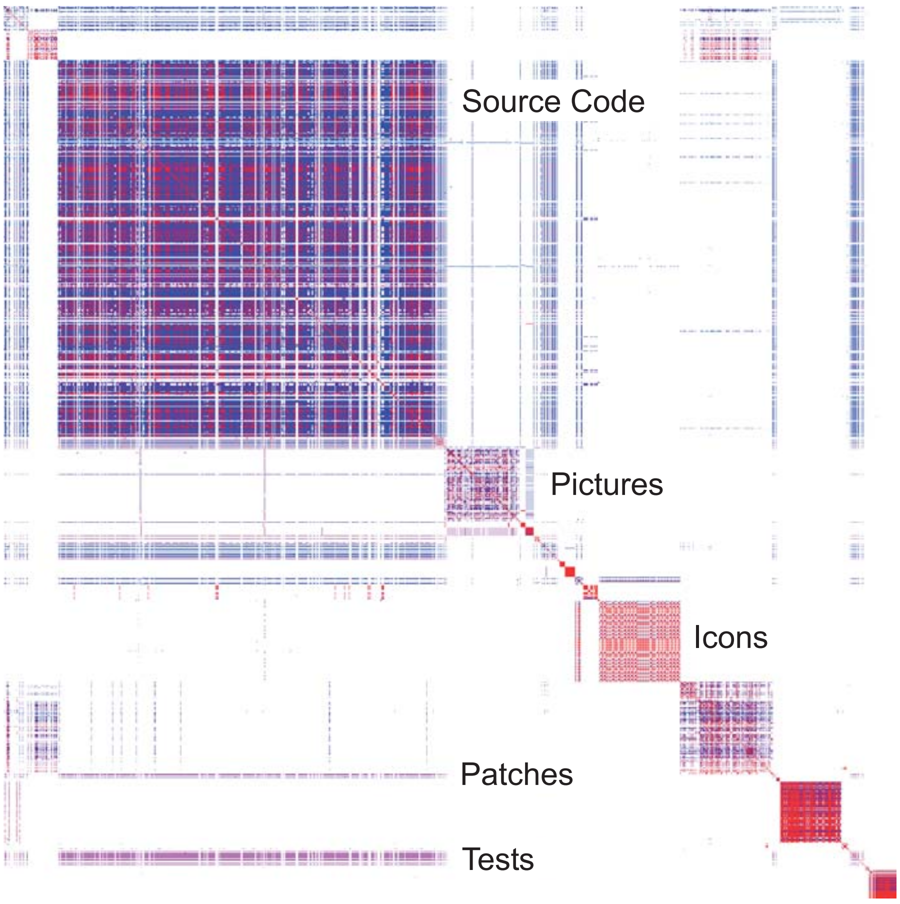
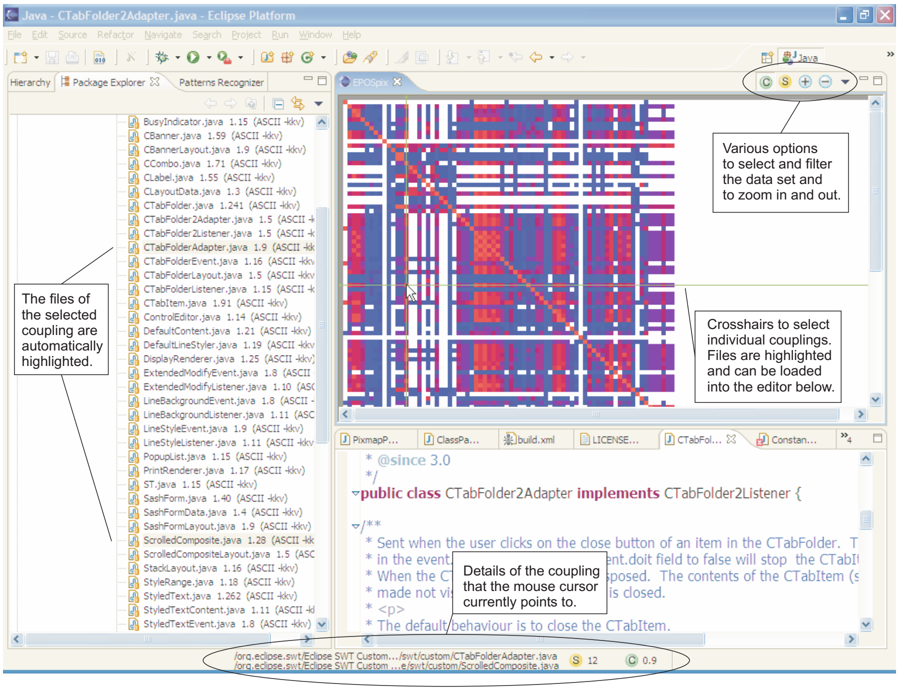


Fig. 5.15. Number of simultaneously changed files in DDD

As the package hierarchy typically reflects the architecture of the system, we expect that related files will be in the same level of the hierarchy. Very closely related files will even be in the same folder. Furthermore, we expect that files which are related will also be changed together more often than those that are unrelated. To add this hierarchical information to the visualization, we lexically sort the files along the axes, including their full path name. As


Java - CTabFolder2Adapter.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Hierarchy
Package Explorer
Patterns Recognizer
EPOSpix
PixmapP...
ClassPa...
build.xml
LICENSE...
CTabFol...
Constan...
Various options to select and filter the data set and to zoom in and out.
The files of the selected coupling are automatically highlighted.
Crosshairs to select individual couplings. Files are highlighted and can be loaded into the editor below.
Details of the coupling that the mouse cursor currently points to.

```
BusyIndicator.java 1.15 (ASCII -kkv)
CBanner.java 1.59 (ASCII -kkv)
CBannerLayout.java 1.9 (ASCII -kkv)
CCombo.java 1.71 (ASCII -kkv)
CLabel.java 1.55 (ASCII -kkv)
CLayoutData.java 1.3 (ASCII -kkv)
CTabFolder.java 1.241 (ASCII -kkv)
CTabFolder2Adapter.java 1.5 (ASCII -k
CTabFolder2Listener.java 1.5 (ASCII -k
CTabFolderAdapter.java 1.9 (ASCII -kk
CTabFolderEvent.java 1.16 (ASCII -kkv
CTabFolderLayout.java 1.5 (ASCII -kkv
CTabFolderListener.java 1.15 (ASCII -k
CTabItem.java 1.91 (ASCII -kkv)
ControlEditor.java 1.14 (ASCII -kkv)
DefaultContent.java 1.21 (ASCII -kkv)
DefaultLineStyler.java 1.19 (ASCII -kkv
DisplayRenderer.java 1.25 (ASCII -kkv)
ExtendedModifyEvent.java 1.8 (ASCII
ExtendedModifyListener.java 1.10 (ASC
LineBackgroundEvent.java 1.8 (ASCII -
LineBackgroundListener.java 1.11 (ASC
LineStyleEvent.java 1.9 (ASCII -kkv)
LineStyleListener.java 1.11 (ASCII -kkv
PopupList.java 1.15 (ASCII -kkv)
PrintRenderer.java 1.17 (ASCII -kkv)
ST.java 1.15 (ASCII -kkv)
SashForm.java 1.40 (ASCII -kkv)
SashFormData.java 1.4 (ASCII -kkv)
SashFormLayout.java 1.9 (ASCII -kkv)
ScrolledComposite.java 1.28 (ASCII -kk
ScrolledCompositeLayout.java 1.5 (ASC
StackLayout.java 1.16 (ASCII -kkv)
StyleRange.java 1.18 (ASCII -kkv)
StyledText.java 1.262 (ASCII -kkv)
StyledTextContent.java 1.11 (ASCII -kl
StyledTextEvent.java 1.8 (ASCII -kkv)
```

```java
 * @since 3.0
 */
public class CTabFolder2Adapter implements CTabFolder2Listener {

/**
 * Sent when the user clicks on the close button of an item in the CTabFolder.  T
 * in the event.                   ent.doit field to false will stop  the CTabIt
 * When the CT                     sposed.  The contents of the CTabItem (s
 * made not vi                     is closed.
 * <p>
 * The default behaviour is to close the CTabItem.
```

```
/org.eclipse.swt/Eclipse SWT Custom.../swt/custom/CTabFolderAdapter.java     S 12     C 0.9
/org.eclipse.swt/Eclipse SWT Custom ...e/swt/custom/ScrolledComposite.java
```

Fig. 5.16. EposPix: integration of the pixelmap visualization into the Eclipse IDE

a result, the pixels form blocks, as can be seen in the pixelmap in Fig. 5.15, which depicts the evolutionary coupling of the DDD. These blocks indicate that files within a directory are coupled, i.e. often changed together. Software developers are interested mainly in the outliers. These are the pixels representing couplings between files in different directories, such as those labeled "Patches" in Fig. 5.15. Outliers can be a sign of a bad system architecture. In other words, if we do not find rectangular areas nicely aligned along the diagonal in the pixelmap, then the system should be restructured.

The pixelmap visualization has been integrated as a plug-in called EposPix into the Eclipse IDE [Ecl] and thus allows one to interactively explore the evolutionary coupling between different files of a project (see Fig. 5.16).

5.5 Visual Data Mining

Using data-mining techniques [HK01, AS95, AS94], various kinds of rules can be extracted from software archives. Visual data-mining techniques [Kei02] can be used to interactively explore these rules. As an example of how data-mining techniques work, we shall take a closer look at association rule mining.

Association rule mining is a technique to find correlations between several items, i.e. rules that state that, for example, whenever a set of items occurs in a transaction, another set of items will occur in the same transaction with a certain probability. What a transaction is depends on the application. For example, a transaction can be a set of goods bought by an individual person at the same time. For software evolution, a transaction can be a set of files changed at the same time.

Association rules are of the form $A \Rightarrow B$, where A and B are disjoint sets. We define the support and the confidence with respect to a set of transactions T as follows:

Support: $\text{supp}(A \Rightarrow B) = \text{freq}(A \cup B)$.
Confidence: $\text{conf}(A \Rightarrow B) = \text{freq}(A \cup B)/\text{freq}(A)$.

In these formulas, the frequency of a set M, $\text{freq}(M)$, is defined as $|\{t \in T : M \subseteq t\}|$.

The Apriori algorithm [AS94] can be used to compute such association rules. This algorithm takes the set of transactions and a constant, minimal support threshold *minsupp* as input and yields all association rules with a support greater than or equal to *minsupp*.

The algorithm works in two phases. In the first phase, it computes iteratively all sets of items which have a frequency greater than or equal to *minsupp*. Here, "iteratively" means that first all sets with one element are computed that satisfy the minimal-support condition. Then all sets with two elements are computed by adding an element to the sets of the previous iteration and checking whether this new set still satisfies the minimal-support condition, and so on. The iterative construction works because the frequency is a monotonic function: $A \subseteq B \Rightarrow \text{freq}(A) \leq \text{freq}(B)$.

Let L be the set of all sets computed in the first phase that satisfy the minimal-support condition. In the second phase, the rules are built as follows: for each set $M \in L$ all rules $A \Rightarrow B$ with $A, B \subseteq M$, $|A| > 0$, $|B| > 0$, and $A \cap B = \emptyset$ are computed, as well as the confidence for each of these rules. For a single set with n elements, we already obtain $n * (n - 1)$ rules.

EPOSee [BDW05] is a tool that extends the standard visualization techniques for association rules and sequence rules to show also the hierarchical order of items. Clusters and outliers in the resulting visualizations provide interesting insights into the relation between the temporal development of a system and its structure.

Similar to association rules, we characterize the evidence and strength of a sequence rule using the two measures support and confidence. A sequence p is a subsequence of another sequence q, if one can derive p from q by deleting elements from q. Several subsequent transactions into the software archive can be combined into transaction sequences. The support $\text{supp}(s)$ of a sequence s is the number of transaction sequences it is a subsequence of. The confidence of a sequence rule $s_1 \Rightarrow s_2$ is then defined as

$$\text{conf}(s_1 \Rightarrow s_2) = \frac{\text{supp}(s_1 \cup s_2)}{\text{supp}(s_1)}$$

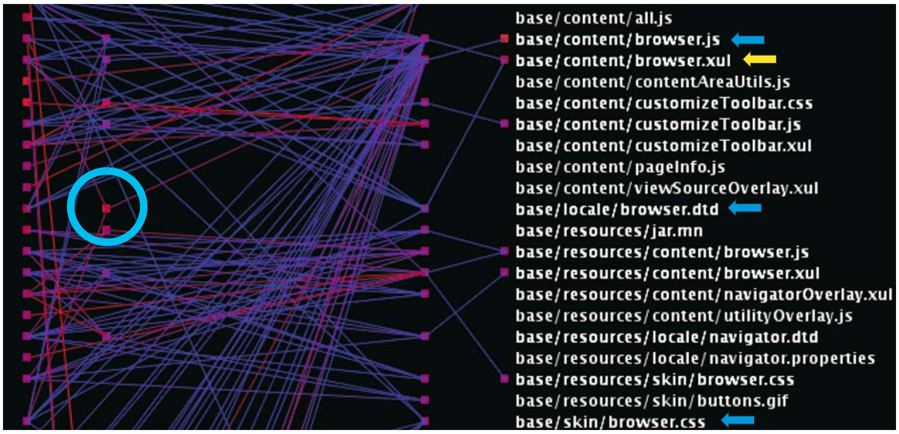


Fig. 5.17. EpoSee: parallel-coordinates visualization of the Mozilla Firefox project

Figure 5.17 shows a parallel-coordinates view [ID90] of the `/browser` directory of the Mozilla Firefox project. In this view every sequence rule is displayed by connecting the node in the n-th column representing the n-th item in the sequence with the node in the $n+1$-th column representing the $n+1$-th item in the sequence. The items are sorted hierarchically and we can see clusters, i.e. all rules only contain items of the same directory. The color of a node indicates the weighted sum of the support values of the prefixes of all rules which share this node, while the color of an edge indicates the weighted sum of the confidences. As the nodes are ordered with respect to the hierarchical order of the items, we see multiple clusters consisting of many edges which relate only items in the same subdirectory. We also see that the files `base/content/browser.js` and `base/content/browser.xul` are related in a very interesting way to almost all Javascript and XUL files, respectively: they are typically changed after one of those files has been changed.

5.6 Summary

In this chapter we have looked at approaches to visualizing the evolution of software systems. First, one can use color coding to show, for an entity or a set of entities, how the values of metrics associated with entities change over time. Instead of using animation, one can also use the third dimension as a time axis, as seen in the 3D version graph in Fig. 5.9. Second, one can show how the structural relations between entities change over time by means of animated graphs. Finally, on the basis of the assumption that two entities are related if they are changed at the same time, one can show this relation either using color-coding in a pixelmap or by drawing a graph.

There is much more information in software archives than current tools exploit, because they mostly leave the exploration and analysis to the user.

Exercises

Exercise 1: Read the Wikipedia article on evolution at `http://wikipedia.org/wiki/Evolution`. Can you identify additional analogies between biological evolution and software evolution?

Exercise 2: On the basis of the editor `TextComponentDemo.java` from the Java Swing tutorial implement a SeeSoft-like line representation simply by allowing the user to interactively change (e.g. by means of a slider) the font size of styled text from 1 to 32. Your program should read in two files: one is a source code file, and the second file contains integers (0–255, one integer per line) that represent some metrics for the source code. Color-code the lines using the class `LOCS.java` from Project 1 and/or define a blue-to-red scale on your own.

You can extend your program to animate program evolution by having it read in more than two files (and, for each, file a metrics file). Then, show the sequence of color-coded visualizations in a thread or provide a slider for the user to interactively select the point in time to be shown; redraw the visualization while the slider is being dragged.

Exercise 3: Evolutionary-coupling data forms a graph that can be represented using both a node link diagram as in Fig. 5.13 or a pixelmap representation as in Fig. 5.15. Can you think of advantages and disadvantages of each of the two representations? Which one would you prefer, for example, for dense, sparse, small, or large graphs? Which one is better suited to finding regularities or outliers?

6 Evaluation

In this chapter we look at evaluation methods that have been used to assess various properties of software visualizations. We discuss both quantitative and qualitative methods, with a focus on the latter. We then take a closer look at the question of how one can evaluate the learning effectiveness of an algorithm animation and, finally, discuss some evaluation results that have been published on the learning effectiveness of algorithm visualizations.

6.1 Claims About Visualization Techniques

Section 2.1 presented several general reasons for using visualization techniques from a cognitive-psychological point of view. But this does not imply that every software visualization is per se a helpful tool. For each particular visualization technique, and indeed for each use of it for a particular task, we have to assess its usefulness. To this end, one has to make explicit what "useful" means. The primary goal of visualization is to convey information. It should convey this information in an understandable, effective, easy-to-remember way. These properties of the visualization are often compared to text-based representations of the same information.

Typical problems with evaluations of visualization techniques are the use of toy data sets, and the generation of visual artifacts that suggest nonexistent relations. But, above all, many evaluations are biased because they have been done by the developers of the visualization.

6.2 Quantitative Evaluation

Quantitative methods measure properties of the visualization or of the algorithm, or properties of the human observer interacting with the visualization. Typical questions about a visualization or algorithm are "How much information fits on the screen?" and "How fast does the algorithm compute the visualization?" Typical questions about the human observer include "How fast does the observer react?", "How much of the information does the observer remember?", and, in particular, in educational settings', "How many tasks were performed correctly?"

Quantitative evaluation requires a statistical analysis of the results of a controlled experiment. An experiment can be described by a set of factors or variables. Factors that are controlled by the person who designs the experiment are called independent, whereas recorded values, such as answers to questions, are called dependent factors. Finally, there are what are called covariate factors, such as age or previous knowledge, which can have some influence on the outcome of the experiment and should be recorded as well.

After the experiment, the resulting data set can be used for descriptive and inferential statistical analyses. Typical statistical measures to describe the central tendency of the data set are its median and mean, and typical measures to describe the dispersion of the data set are its standard deviation and variance.

Inferential statistics tries to assess to what extent observed properties of the data set might have happened by chance. One of the most common methods of inferential statistics is to have two test groups and to compare their performance using a T-test.[1] A T-test determines whether the difference between the means of the two groups is statistically significant. Intuitively, a result is statistically significant if the probability that it is wrong is below a given value, sometimes called the α level. An α level of 5% has become widely accepted.

Some authors argue that in software engineering research it would be better to simply provide the error probability and leave the decision of whether it is "significant" to the reader [PPV00]. On the one hand, for safety-critical applications, an extremely low α level would be important. On the other hand, in situations where no better information is available, even an α level of 40% may still help to make a decision.

6.3 Qualitative Evaluation

Qualitative methods gather data about the individual experience of human observers with the visualization. This experience is verbalized in the form of introspective reports. Ericsson and Simon found that *think-aloud* reports during the task are more reliable than recall protocols [ES80].

When it comes to the human perception of and interaction with a visualization, qualitative evaluation methods are of the outmost importance for various reasons [McC93], including the fact that they require fewer test persons and cover more aspects of the visualization. In contrast, to achieve significant results, quantitative evaluations require that a sufficient number of participants perform a controlled experiment

[1] For more than two groups, a common method is ANOVA (analysis of variance).

Qualitative methods are in particular useful for formative evaluation. Here the goal is to improve a program under development.

6.3.1 Evaluation Based on Gestalt Theory

Sun and Wong have developed a set of 14 criteria for the evaluation of UML layouts [SW05]. On the basis of these criteria, they evaluated two commercial UML tools (Borland Together and Rational Rose). Their criteria were based mostly on gestalt-theoretical principles (see Sect. 2.1.4). These principles can be broadly characterize as guiding either the perceptual organization or the perceptual segregation of objects. The former is related to the question of which objects belong together, whereas the latter is related to the question of how a figure separated from the background.

The gestalt-theoretical principles that are important for perceptual organization are simplicity (the use of a suggestive figure), similarity, proximity, familiarity (meaningfulness), and connectedness. For perceptual segregation, symmetry, orientation, and contour play a crucial role.

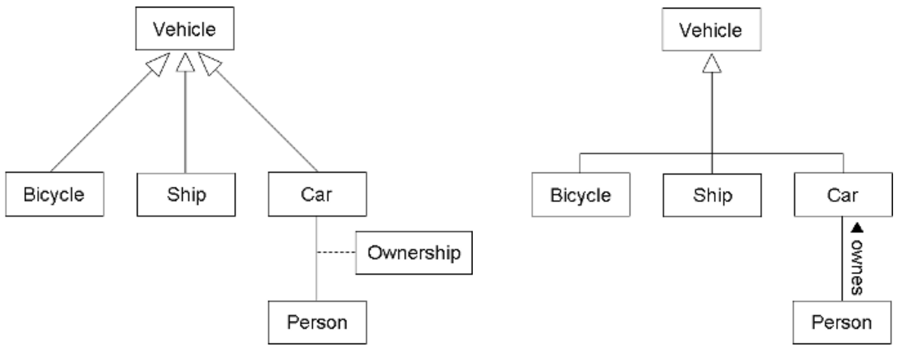


Fig. 6.1. UML diagrams: separate vs. joined inheritance arcs and association classes vs. labeled association links

For the perceptual organization of UML layouts, Sun and Wong suggested, for example, the following three criteria (see Figure 6.1):

- Inheritance arcs should be joined rather than separated (simplicity).
- An association name should be set beside the line, rather than in another association class linked to the line.
- The layout should only show selective information, i.e. information that suits the purpose.

For the perceptual segregation of UML layouts, they suggested, for example, the criterion that overlapping should be avoided, because overlapping will destroy the contours of objects and they will become difficult to recognize (contour and connectedness).

6.3.2 Task-Oriented Evaluation

We all know about task-oriented analysis from computer magazines: we all know task-oriented analysis: software products from different vendors are compared in large tables by listing features and other properties such as prices and licenses, but, more importantly, tasks which are typical of the application. For each product, the table indicates what tasks it supports and possibly even how well it supports each task.

Task-oriented analysis is very popular, because it does not require too much effort, while it actually provides an answer to a key question from a user's point of view: Does the system solve the user's problem?

Pacione et al. performed a task-oriented evaluation of five dynamic visualization tools [PRW03], ranging from visual debugging to UML tools that generate sequence diagrams. For their analysis, they came up with nine large-scale and six small-scale tasks which are typical of software comprehension. The former set of tasks included, for example, finding design patterns that have been applied in the subject system, and or reconstructing the high-level structure of the subject system. The small-scale tasks included finding the collaborations of a small set of objects, and how the state of a specific object changes. Pacione et al. concluded that none of the tools in their study supported all tasks at all levels and that combining static and dynamic information might lead to better results (see also Sect. 4.3).

The cognitive walkthrough is a task-oriented technique for evaluating the usability of a system. The technique is named in analogy to code walkthrough in software engineering. When performing a cognitive walkthrough one keeps track of the user's thought processes, decision making, and memory load at each step when the task is performed [WRLP94, PLRW92].

6.3.3 The Cognitive-Dimensions Framework

The cognitive-dimensions framework [GP96, BG03] is a qualitative and often formative evaluation method to assess (interactive) tools for storing, manipulating, and displaying information, or, as the authors of the framework put it, they are "a set of discussion tools for use by designers and people evaluating designs". Six types of generic tasks are distinguished:

Incrementation is the task of adding information to a system or document.
Transcription is the task of transforming information from one representation to another.
Modification refers to the changing of information or its representation.
Exploratory design is the task of sketching new representations without knowing the solution in advance.
Searching is the task of looking for a known piece of information.
Exploratory understanding is the task of discovering the overall structure of the information.

Table 6.1. Cognitive-dimensions profiles of four generic tasks

	Viscosity	Hidden dependencies	Premature commitment	Visibility
Exploratory design	Harmful	Acceptable	Harmful	Important
Modification	Harmful	Harmful	Harmful	Important
Incrementation	Acceptable	Acceptable	Harmful	Useful, nonvital
Transcription	Acceptable	Acceptable	Harmful	Useful, nonvital

Evaluation using the framework works as follows. First, one has to identify what generic tasks the user will perform with the system. Then, for each task, one has to assess how it fits each cognitive dimension. This leads to an observed profile for each task, which can be compared with an ideal profile listed in the cognitive dimensions tutorial [BG]. Table 6.1 shows the profiles of four generic tasks based on four of the 14 cognitive dimensions listed in Table 6.2.

Table 6.2. List of cognitive dimensions

Cognitive dimension	Description
Viscosity	Resistance to change
Visibility	Ability to view components easily
Premature commitment	Constraints on the order of doing things
Hidden dependencies	Important links between entities are not visible
Role expressiveness	The purpose of the entity is readily inferred
Error-proneness	The notation invites mistakes and the system gives little protection
Abstraction	Types and availability of abstraction mechanisms
Secondary notation	Extra information in means of other than formal syntax
Closeness of mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Hard mental operations	High demand on cognitive resources
Provisionality	Degree of commitment to actions or marks
Progressive evaluation	Work-to-date can be checked at any time

As an example, consider the Creole plug-in that analyzes a Java program and produces various kinds of diagrams such as those shown in Sect. 3.4.4. With respect to most of the cognitive dimensions, Creole is well designed. Considering visibility, for example, boxes can be expanded by simply clicking on a box to show its components. On the other hand, the expressiveness of the graphical representation is very restricted: classes, packages, methods, and attributes are all represented by boxes and the semantics of a box can be inferred from the context. Furthermore, the viscosity of Creole is high.

When the source code changes, for example when a class is renamed, the class hierarchy has to be recomputed, and a new layout of the diagram is computed that can be quite different from the previous one.

For each of the dimensions, the cognitive dimensions tutorial [BG] provides a definition; one or more thumbnail examples; a more detailed explanation; the cognitive relevance of the dimension, in particular with respect to the generic tasks; cost implications; more detailed examples; workarounds; remedies; and trade-offs.

6.4 Educational Evaluation

Learning scenarios such as those described in Sect. 4.4.8 have been developed with the goal of facilitating understanding and learning of an algorithm. But the question arises of whether they really achieve this goal. In other words, the learning effectiveness has to be evaluated. To this end, user studies have to be performed.

We shall illustrate the various aspects of the design of a study by examples taken from an evaluation that we did several years ago for an interactive animation of a generation algorithm for finite automata which we briefly called GANIFA [GAN].

The screen dump in Fig. 6.2 shows how the generation process of a lexical analyzer is visualized by GANIFA. This example shows the conversion of a regular expression $(a|b)^*$ into an appropriate nondeterministic finite-state automaton ($RE \rightarrow NFA$). The generator has been integrated into an applet for visualizing the generation and computation of finite automata, which has been used in an electronic textbook on the theory of finite automata [Gan00].

Design of a Study

For a study of learning effectiveness, two key decisions have to be made: what kind of experiment will be performed, and who will be the test persons?

Test Persons Obviously, the test persons should belong to the target group of the learning scenario, for example, one can hardly evaluate a textbook for children with adult test persons. On the other hand, a sufficient and, in particular, representative number of members of the target group are often not available as test persons. Therefore many studies on algorithm animation have been done with psychology or computer science students and as a result are biased to some extent. After the test persons have been chosen, one has to split them into two ore more groups. These groups then use alternative means to learn about an algorithm. The simplest approach to forming these groups is to assign test persons randomly to groups. In the evaluation of GANIFA we had about 118 first-year computer science students, split randomly into four groups. Later it turned out that in some groups there were considerably more students with very good high-school grades in mathematics.

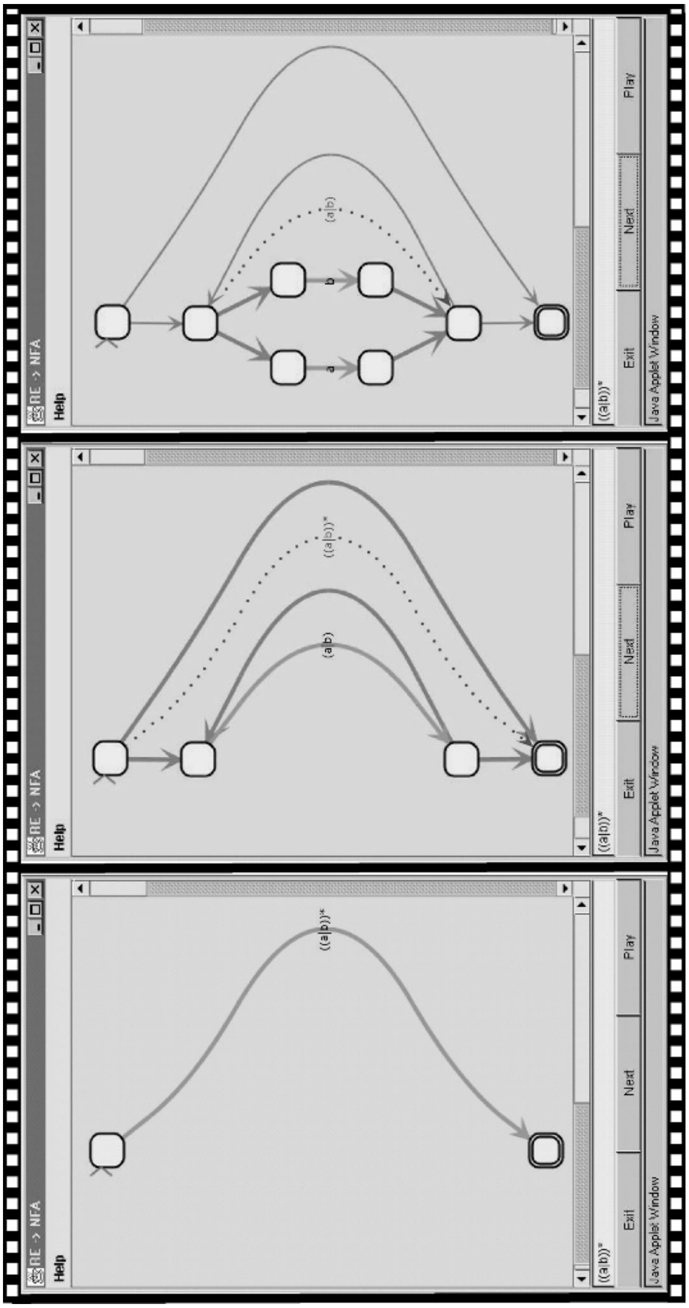


Fig. 6.2. GANIFA: animation of the generation of a finite automaton

If one has only a small number of test persons, one can try to prevent such unexpected effects by performing a pre-test first and taking the results of the test into account when forming the groups: for example, one may use the same numbers of people with good and bad pre-test results in each group.

Factors Now one can perform the actual experiment. In general terms, the goal of such an experiment is to find a relation between independent variables, i.e. the controlled factors, and dependent variables, i.e. the measured factors. To this end, the experiment is performed with several different values of the independent variables for each test group, and the changes in the dependent variables are recorded. Assuming n independent variables with m different values for each of these variables, one would need $n * m$ test groups with a sufficient number of persons per group to allow statistically significant results. So, as a matter of fact, most studies have concentrated on one or two independent variables. For example, learning efficiency is typically investigated by comparing different learning scenarios, i.e. the learning scenario is the independent variable.

In the GANIFA evaluation [Ker04, DK01], the independent variable investigated was the learning medium. We chose four media: a learning-software package with fixed animations, a learning-software package with generated animations depending on the students' inputs, a textbook, and a conventional lecture.

Measurement As we are interested in learning efficiency, we need appropriate measures. One approach is to measure performance during the experiment, i.e. to capture the students' behavior with traces or video recording. Instead or in addition, one can do post-tests, i.e. after the experiments the students have to answer a set of questions. With these questions, one tries to measure the students' conceptual or declarative knowledge, for example their understanding of the abstract properties of an algorithm, and their procedural knowledge or skills, for example their understanding of the procedural, step-by-step behavior of an algorithm. With a pre-test, one can see what the students knew beforehand, and in combination with the post-test results, one can measure the improvement or increase of knowledge.

For the GANIFA evaluation, we performed both pre- and post-tests to see whether the students' knowledge and skills had improved. In these tests, we had knowledge questions about conceptual and procedural understanding, transfer questions asking the students to transfer/apply knowledge in a different context, and open questions, where students could make comments and suggestions. Open questions are very important for identifying factors and problems that one was not aware of when designing the study. Below are some example questions from the GANIFA evaluation:

Pre-test

Do you know what finite automata are?
Which words belong to the language defined by the regular expression $(\texttt{ab})^*$?

Post-test

Knowledge question
Which words belong to the language defined by the regular expression $\texttt{ab}^*\texttt{a}$?

Transfer question
We add the notation r^+ to our regular-expression language, where r is a regular expression, such that $L(r^+) = \{w^n | w \in L(r), n \geq 1\}$. For example, $L(\texttt{a}^+) = \{\texttt{a}, \texttt{aa}, \texttt{aaa}, \dots\}$. Give a construction rule for a transition diagram of an NFA.

Open question:
What properties of the animation helped you to better understand the generation algorithm?

Learning Theories While it might be sufficient for a designer of an algorithm animation to see that it helps learning, cognitive scientists have the more ambitious goal of trying to explain why. To this end, they try to apply existing learning theories or create new ones which address issues such as the correspondence of the graphical metaphors and the mental model, the use of both hemispheres of the brain, and the active construction of subjective knowledge by the test person.

6.5 Some Interesting Empirical Results

In this section, we discuss several empirical studies related to various fields of software visualization.

Algorithm animation In a metastudy performed by Hundhausen et al., more than 40% of the 24 studies considered did not find significant results [HDS02]. Several studies found that electronic learning material (multimedia or hypermedia) with algorithm animations outperformed lectures. The comparisons with textbooks are less clear. But the most important result of the metastudy was that the form of the learning exercise is actually more important than the quality of the visualizations used:

> Thus, according to our analysis, how students use AV [algorithm visualization] technology, rather than what students see, appears to have the greatest impact on educational effectiveness.

In particular, algorithm visualizations were efficient in scenarios were students had to actively solve prediction and programming exercises.

Static Program Visualization and Visual Programming There have also been many empirical studies about the use of diagrams for programming. For LabView, a widely used visual programming language, Green and Petre [GP92] found, in a quantitative study, that in most cases the diagrams were more difficult to understand than a textual representation, for both occasional and experienced users. Similar experimental results had been already reported about flowcharts in 1977 [SMMH77], but about ten years later, in a replicated experiment [Sca89] using time as an additional dependent variable, it turned out that flowcharts outperformed source code for algorithm comprehension.

Visual debugging Jones et al. evaluated the effectiveness of the Tarantula tool [JHS02, Tar] not with test persons, but by applying the system to different versions of the same program with different faults and assessing how many of the faulty and nonfaulty statements were colored reddish. In their experiments, they applied 1000 test cases to 60 different versions of a program with 9500 lines of code. Of these, 20 versions contained a single (known) fault, 10 versions two faults, 10 versions three faults, 10 versions four faults, and 10 version five faults. They found that fewer than 20% of the nonfaulty statements were colored in the reddest 20% of colors used. While the majority of faulty statements were colored in a reddish color, they also found that with an increasing number of faults, the number of reddish-colored faulty lines decreased.

Class Diagrams Irani and Ware have performed several experiments to compare UML diagrams and geon diagrams (see Sect. 3.4.2 for more details on geon diagrams). In one experiment [IW00], they first showed a UML or geon diagram for 15 seconds to the test persons, and then they presented them with diagrams of substructures and asked whether these occurred in the original diagram. All diagrams were shown on the computer screen and the test persons had just to press the “Y” or “N” key. In this experiment the average identification time was 4.3 seconds for geon diagrams, but 7.1 seconds for UML diagrams. But, even more importantly, the error rate was only 13% for geon diagrams, compared with 26% for UML diagrams. Thus the identification of substructures was much faster and more accurate with geon diagrams.

In another experiment [IW04], the test persons were given a problem description and four UML or geon diagrams. The task was to decide which one of the diagrams was created for that problem. In this experiment the average error rate was 15% for geon diagrams and 36% for UML diagrams.

Graph Drawing For more than a decade, Helen Purchase has performed empirical studies related to the aesthetics of graph layouts. Whereas in earlier work she found that reducing the number of edge crossings was the most important aesthetic consideration [PCJ96, Pur97], in recent work continuity also turned out to be an important factor [WPCM02]. Here, continuity means the sum of the angular deviations of the incoming and outgoing edges for each node on a path. If the edges on a path form a straight line, the path has a continuity of 0 degrees. In Fig. 6.3 the path from node A to C has a continuity of 0 degrees, whereas the path from A to E has a continuity of 180 degrees.

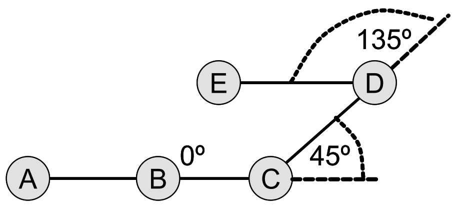


Fig. 6.3. Angular deviations along a path

In one of her experiments, test persons were asked to identify the shortest path between two highlighted nodes. For each graph shown, the values of several variables were recorded: the response time *rt*, the shortest path length *spl* in terms of the number of edges, the continuity *con* of the shortest path in degrees, the number of edges that cross the shortest path *cr*, and the number of branches *br* from the intermediate nodes on the shortest path. Some other properties, such as the total number of edge crossings, that turned out to be not significant using an α level of 5% were also recorded. By stepwise multiple-regression analysis the following relation was obtained between these variables with a regression correlation coefficient $R^2 = 0.784$:

$$rt = 0.414\, spl + 0.406\, con + 0.317\, cr + 0.172\, br \qquad (6.1)$$

This indicates that the response time increases with the length of the shortest path, but also with its continuity and the number of crossings on this path.

6.6 Summary

There have been quite a lot of studies related to algorithm visualization systems in education. This is unfortunately not the case for other areas of software visualization. The lack of empirical studies is a shortcoming no only of software visualization research, but also of software engineering and computer science in general [Tic98]. In a survey of 400 research articles [TLPH95], Tichy et al. found that more than half of the software engineering papers were not experimentally validated.

As quantitative evaluations that involve human test persons are very time-consuming, at least qualitative evaluations should be performed during the design of visualization tools or posthoc.

Only a few software visualization systems exist that support the development of large software systems with the large, distributed teams of programmers that are common in the software industry today. The evaluation of industrial software visualization is full of open questions for empirical studies. To what extent has software visualization been applied effectively in industry? Does it keep its promises of increased productivity and decreased development and maintenance costs?

Exercises

Exercise 1: Design a controlled experiment to evaluate whether a tool such as those discussed in Sect. 3.4.4 can help a user to understand the source code of a given software system better than by reading only the textual representation of the source code itself. What are the independent, dependent, and covariate variables in your experiment?

Exercise 2: In Exercise 2 of Chap. 4 you had to implement a visualization of an algorithm for the scheduling of independent tasks. Now imagine that you had to evaluate the learning effectiveness of your visualization. To this end, you would have to design a pre- and a post-test. Suggest at least a pre-test question, a knowledge question, a transfer question, and an open question for such a questionnaire.

Exercise 3: Use the cognitive-dimensions framework to evaluate a software visualization system of your choice, preferably one that you have access to. For example, you could evaluate a tool that analyzes a Java program and produces a UML class diagram such as those discussed in Sect. 3.4.4. Do all dimensions apply to the system that you have chosen?

7 Conclusions

7.1 The Visualization Pipeline – Revisited

In the first chapter of this book we introduced the visualization pipeline, emphasizing that software visualization tools typically have to address all three phases: data acquisition, analysis, and visualization. We now give a quick summary of the various techniques and data sources used in the various fields of software visualization covered in this book:

Static program visualization: The starting point for static program visualization is typically the source or machine code of a program. For some visualizations, such as pretty printing or Jackson diagrams, a syntactical analysis is sufficient. Further analyses include control-flow and data-flow analysis. Annotated graph diagrams are the prevalent representation here. For example, to support optimization of real-time applications, one can highlight critical paths in such graphs. Instead of nodes and edges, Nassi–Shneiderman diagrams use containment and neighborhood, i.e. nested and attached boxes.

Dynamic program visualization: Instrumentation of the source, intermediate, or binary code is the main data acquisition method. For algorithm animation, identifying interesting events is a widely used manual instrumentation technique. The actual data is gathered at run time, which includes direct access to program memory. Filtering, for example of events, can be used to reduce the amount of available runtime data. Visualization of the dynamic information is based either on accumulation, spatial projection, or (smooth) animation using primitive graphical objects such as boxes, lines, and circles, as well as 3D computer graphics.

Visual debugging: Visual debugging is in fact a special case of dynamic program visualization. In addition to the source or machine code, input data (test cases) and direct access to program memory (registers, stacks, and heaps) are important. The analysis is either left to the user, for example by setting break points or interactively unfolding data structures, or automated, for example extraction of memory graphs or reference patterns. Dicing, i.e. the computation of the difference between two slices (here memory slices or execution slices), is another important analysis technique. Graph representations are used to visualize the program state graph representations are used, whereas the results of test cases are shown as color-coded program text.

Software architecture: For the visualization of software architectures, the documentation is used in addition to the source code. Static and dynamic program analyses are used to compute software metrics. Aggregation is an important technique for reducing the size of the resulting, typically graph-based visualizations. Software metrics are indicated by color coding in these graphs. Recently, several 3D visualizations based on real-world metaphors have been developed.

Software evolution In addition to the static and dynamic information used in the above techniques, information about the development process is extracted from software repositories, mailing lists, and bug databases. Some of the classical program analyses can be extended to work on several versions of a program; this is called multiversion program analysis. Statistical analyses and, in particular, (visual) data-mining techniques have been applied to deal with the enormous amount of data. The temporal aspect of evolution is often visualized only by accumlation, for example in the pixelmap visualization, or by a time axis in two- and three-dimensional diagrams. In some cases animated color-coded program text or graphs have been used.

Whereas the original research on software visualization tried to produce visualizations of either the source code or the data structures at run time, later work included results of program analyses, and more recent work has integrated other sources of information such as software repositories and bug databases. As different as the information sources are the analysis methods, ranging from syntactical analysis through static and dynamic program analysis to data mining.

From the early days of software visualization, graphs have played a major role – even in text-based displays. With the availability of better displays, color coding, animation, and 3D representations have been investigated, indicating that research has often been more driven by technology than actually trying to address the user's needs.

While there exists a lot of work on evaluating algorithm animations that even allows one to perform metastudies, other areas of software visualization research lack such evaluations. In Chap. 6 we discussed several qualitative approaches that may be used to assess the quality of software visualization tools with moderate effort.

7.2 Further Reading and Resources

There exist many more approaches and tools than we have covered in this book. Table 7.1 lists some more software visualization tools. The proceedings of the ACM Symposia on Software Visualization (SOFTVIS), the IEEE Workshops on Visualizing Software for Understanding and Analysis (VISSOFT), and the IEEE Symposia on Information Visualization, as well as several anthologies [SDBP98, Die02b, Zha03], are valuable resources for finding more information on software visualization research.

Table 7.1: Some software visualization tools

Tool	Availability	URL
	Static program visualization	
AiCall	Commercial	`http://www.aisee.com/aicall/`
CodeSurfer	Commercial	`http://www.grammatech.com/products/codesurfer/index.html`
jGrasp	Research	`http://www.eng.auburn.edu/grasp/`
OptimalAdvisor	Commercial	`http://javacentral.compuware.com/products/optimaladvisor/`
Portable Bookshelf	Research	`http://swag.uwaterloo.ca/pbs/`
Source-Navigator	Open source	`http://sourcenav.sourceforge.net/`
VISTA	Research	`http://www.cigitallabs.com/research/demos/vista/`
Visualize it!	Commercial	`http://www.powersoftware.com/vz/`
	Algorithm animation	
DsCats	Research	`http://www.cs.arizona.edu/dscats/dscats.html`
Jeliot	Research	`http://cs.joensuu.fi/jeliot/`
LEONARDO	Research	`http://www.dis.uniroma1.it/~demetres/Leonardo/`
MatrixPro	Research	`http://www.cs.hut.fi/Research/MatrixPro/`
POLKA	Research	`http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html`
SAMBA	Research	`http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html`
	Visual debugging	
DDD	Open source	`http://www.gnu.org/software/ddd/`
IDA	Commercial	`http://www.datarescue.com/idabase/`
Jinsight	Research	`http://www.alphaworks.ibm.com/tech/jinsight`
Logiscope	Commercial	`http://www.telelogic.com/products/tau/logiscope/`

Continued on next page

Table 7.1 – continued from previous page

Tool	Availability	URL
TuningFork	Research	`http://www.alphaworks.ibm.com/tech/tuningfork`
TimeMachine	Commercial	`http://www.ghs.com/products/timemachine.html`
Tarantula	Research	`http://www.cc.gatech.edu/aristotle/Tools/tarantula/`
Web Services Navigator	Research	`http://www.alphaworks.ibm.com/tech/wsnavigator`
xSlice	Commercial	`http://www.cleanscape.net/products/testwise/tools_xslice.html`
	Software architecture	
ArgoUml	Open source	`http://www.argouml.org/`
BlueJ	Research	`http://www.bluej.org/`
ESS-Model	Open source	`http://essmodel.sourceforge.net/`
Fujaba Tool Suite	Research	`http://www.uni-paderborn.de/cs/fujaba/`
GOOSE	Research	`http://esche.fzi.de/PROSTextern/software/goose/`
GoVisual	Commercial	`http://www.oreas.com/`
Imagix	Commercial	`http://www.imagix.com/`
JarInspector	Research	`http://www-pr.informatik.uni-tuebingen.de/c/forschung/uml/jarinspector.xml`
RIGI	Research	`http://www.rigi.csc.uvic.ca/`
SHriMP	Research	`http://www.thechiselgroup.org/shrimp`
Sotograph	Commercial	`http://www.software-tomography.com/html/sotograph.htm`
SugiBib	Research	`http://www.sugibib.de/`
Surveyor	Commercial	`http://www.lexientcorp.com/`
VizzAnalyzer	Research	`http://www.arisa.se/index_tools.html`
yDoc	Commercial	`http://www.yworks.de/en/products_ydoc.htm`
	Software evolution	
CodeCrawler	Research	`http://www.iam.unibe.ch/~scg/Research/CodeCrawler/`
EpoSee	Research	`http://www.eposoft.org/eposee`
CCVisu	Research	`http://www.cs.sfu.ca/~dbeyer/CCVisu/`
Visual Code Navigator	Research	`http://www.win.tue.nl/~lvoinea/VCN.html`
VRCE	Commercial	`http://www.aicas.com/vrce_en.html`

Continued on next page

Table 7.1 – continued from previous page

Tool	Availability	URL
	Graph drawing	
AGD	Research	`http://www.ads.tuwien.ac.at/AGD/`
aiSee	Commercial	`http://www.aisee.com/`
DGD	Research	`http://www.st.uni-trier.de/GD`
GraphViz	Open source	`http://www.graphviz.org/`
yFiles	Commercial	`http://www.yworks.de/`

7.3 The Future of Software Visualization

In 2002, I wrote a chapter introduction titled "Future Perspective" for an anthology on software visualization [Die02b]. There, I identified the following four challenges to increase the impact of software visualization research in the future:

> *Breaking New Ground* [...] We expect that exploring all aspects at all layers of software will lead to synergies and thus will ultimately stir the traditional areas of software visualization as well.
>
> *Integration* Software visualization will be doomed to remain an academic endeavor if we do not succeed in integrating it into working environments and thus into the work flow of programmers, designers and project managers. [...]
>
> *Theory* [...] Based on such empiric data, cognitive and pedagogical theories can be formulated and validated. Ultimately, these should guide the design and use of future software visualization systems.
>
> *Forum* [...] The software visualization community needs a forum in which to share best practices and to promote the state of the art.

Since 2002, some of these challenges have been met, or at least partially met. For example, the ACM Symposium on Software Visualization (SOFTVIS) has been established as such a forum, and many software visualization tools have been integrated as plug-ins into the Eclipse development platform [Ecl]. Other challenges remain, and new ones have arisen. Here are some challenges for the coming years:

Visual standards In scientific visualization such as medical imaging, there is a physical analog. In software visualization we lack this physical analog and have to resort to metaphors. While it makes sense for researchers to explore the whole design space, in order to achieve wide adoption of software visualization in practice it is important that there are established visual standards, such that the user's effort in learning how to interpret them pays off.

Automatic focusing: Information visualization researchers have come up with many kinds of focus + context visualizations (see Sect. 2.3), and some of these techniques have already been adapted for software visualization. While these techniques provide a means to show some parts of the information in more detail, there are few techniques (see Sect. 4.6.1) on how to automatically decide what these parts are.

Structural change: The structure of both data structures and program code is mostly visualized as graphs. While data structures change during program execution, program code changes during the evolution of the software system. For small graphs, these changes can be shown using graph animation. For larger graphs, these animations are confusing. Either one has to develop automatic focusing techniques to reduce the size of the graphs, or, alternatively, find static visualizations of changes.

Continuous navigation: Early on, the importance of smooth animations for showing state transitions in algorithm animation has been realized [Sta90b]. But smooth animations are not only important for dynamic program visualization but also for static program visualization. For example, in reverse engineering tools, it could improve continuous navigation from lower levels of abstraction to higher levels and vice versa.

Algorithm explanation: How can we visually execute real programs with abstract data (see Sect. 4.4.7)? For visual debugging, it might be useful to execute a program with real data to a breakpoint or specified line, and to continue by stepping through the program using abstract data.

Real-time systems: Currently, few dynamic visualizations of real-time systems go beyond capturing real-time events and showing time-series [BCF$^+$06, Gre]. For example, one could augment static program visualizations with both static and dynamic analysis results for comparing theoretical worst-case execution times, paths, and memory usage with those estimated on the basis of several test runs.

3D visualization In modern computer games, 3D graphics and narrative elements help the user to find her way through the virtual worlds. Recently, for every major operating system, 3D desktops have been released, or at least some animated 3D effects have been added to the old 2D desktop. Today's software visualization tools do not even exploit the graphics power of an average PC or laptop. 3D visualization may fulfill its promises (see Sects. 3.4.5 and 4.4.5, for example), once we turn to post-WIMP[1] interfaces which will certainly be three-dimensional and involve other interaction devices like gaze and motion tracking.

End-user visualization: Today, software visualization tools are used by software developers. Some kinds of visualizations may be helpful for end-users who want to get some information about an application that they are using. For example, they may want to know what components are required for this application, and what are licences of these components, how much

[1] WIMP stands for windows, icons, menus and pointing device.

of the functionality the user is actually using, and what resources such as network or external storage does each of the components access.

Trusting software This is an instance of the previous challenge. Is it possible to analyze and visualize the flow of information in a distributed application such that the user trusts the application not to leak confidential information. For example, before using a Web service (see Sect. 4.6.3), the user might want to make sure that her credit card information is only passed through trusted intermediate Web services.

References

[AH98] Keith Andrews and Helmut Heidegger. Information slices: Visualising and exploring large hierarchies using cascading, semi-circular discs (late breaking hot topic paper). In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'98)*, pages 9–12, Research Triangle Park, NC, 1998.

[Ang] AbsInt Angewandte Informatik GmbH. Call graph visualization and stack usage analysis. `http://www.aisee.com/aicall/`.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20th Very Large Data Bases Conference (VLDB)*, pages 487–499, San Francisco, CA, 1994. Morgan Kaufmann.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee S. P. Chen, editors, *Eleventh International Conference on Data Engineering, Taipei, Taiwan*, pages 3–14, Washington, DC, 1995. IEEE Computer Society Press.

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information pyramids: A new approach to visualising large hierarchies. In *Proceedings of IEEE Visualization97*, 1997.

[Bae73] Ronald Baecker. Towards Animating Computer Programs: A First Progress Report. In *Proceedings of the Third NRC Man-Computer Communications Conference*, 1973.

[Bae81] Ronald Baecker. Sorting Out Sorting. 30 minute color film (developed with assistance of Dave Sherman, distributed by Morgan Kaufmann, University of Toronto), 1981.

[BCF$^+$06] David F. Bacon, Perry Cheng, Daniel Frampton, David Grove, Matthias Hauswirth, and V. T. Rajan. Demonstration: On-line visualization and analysis of real-time systems with TuningFork. In Alan Mycroft and Andreas Zeller, editors, *Proceedings of 15th International Conference on Compiler Construction (CC 2006), Vienna, Austria, March 30–31*, volume 3923 of *Lecture Notes in Computer Science*, pages 96–100. Springer, 2006.

[BD05] Michael Balzer and Oliver Deussen. Voronoi treemaps. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2005),* *23–25 October 2005, Minneapolis, MN*, Washington, DC, 2005. IEEE Computer Society Press.

[BDW05] Michael Burch, Stephan Diehl, and Peter Weißgerber. Visual data mining in software archives. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS05), St. Louis, MO*, New York, NY, May 2005. ACM Press.

[BE95] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(2):119–133, 1995.

[BE96] Thomas Ball and Stephen G. Eick. Software Visualization in the Large. *IEEE Computer*, 29(4):33–43, 1996.

[BG] Alan F. Blackwell and Thomas R. G. Green. Cognitive dimensions resources (including tutorial). `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/`.

[BG03] Alan F. Blackwell and Thomas R. G. Green. Notational systems – the cognitive dimensions of notations framework. In J. M. Carroll, editor, *HCI Models, Theories and Frameworks: Toward a multidisciplinary science*, pages 103–134. Morgan Kaufmann, San Francisco, 2003.

[BHK$^+$02] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: an abstraction for multimedia streaming. *Multimedia Systems*, 8(5):406–419, 2002.

[BHvW00] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. Squarified treemaps. In *Proceedings of Joint IEEE TCVG Symposium on Visualization*, pages 33–42, Vienna, 2000. IEEE Computer Society Press.

[BK01] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC2001)*, pages 7–17, Toronto, Canada, 2001.

[BL76] Laszlo A. Belady and Meir M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BM89] Ronald M. Baecker and Aaron Marcus. *Human factors and typography for more readable programs.* ACM Press, New York, NY, USA, 1989.

[BM98] Ronald M. Baecker and Aaron Marcus. Printing and publishing C programs. In John Stasko et al., editor, *Software Visualization – Programming as a Multimedia Experience*, pages 45–61. MIT Press, Cambridge, MA, 1998.

[BN93] Marc H. Brown and Marc Najork. Algorithm animation using 3D interactive graphics. In *Proceedings of ACM Symposium on User Interface Software and Technology, Atlanta*, pages 93–100, New York, NY, 1993. ACM Press.

[BN96] Marc Brown and Marc Najork. Collaborative active textbooks: A Web-based algorithm animation system for an electronic classroom. In *Proceedings of the 1996 IEEE International Symposium on Visual Languages, Boulder, CO*, Washington, DC, 1996. IEEE Computer Society Press.

[BN05] Dirk Beyer and Andreas Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of 13th International Workshop on Program Comprehension (IWPC 2005), St. Louis, MO*, pages 259–268, Washington, DC, 2005. IEEE Computer Society.

[BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems

(VisSym 2004, Konstanz, Germany). In *Proceedings of Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization (2004)*, pages 261–266, 2004.

[BP84] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.

[BR96] Marc Brown and Roope Raisamo. JCAT: Collaborative active textbooks using Java. In *Proceedings of CompuGraphics'96*, Paris, France, 1996.

[Bra01] J. Branke. Dynamic graph drawing. In *Drawing Graphs [KW01]*. Springer Verlag, 2001.

[Bri02] Robert Bringhurst. *The Elements of Typographic Style*. Hartley & Marks, Vancouver, 2002.

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[Bro88] Marc Brown. Exploring algorithms with Balsa-II. *Computer*, 21(5):14–36, 1988.

[BS84] Marc Brown and Robert Sedgewick. A system for algorithm animation. In *Proceedings of ACM SIGGRAPH'84, Minneapolis, MN*, pages 177–186, New York, NY, 1984. ACM Press.

[BW00] Beatrix Braune and Reinhard Wilhelm. Focussing in algorithm explanation. *Transactions on Visualization and Computer Graphics*, 6(1):1–7, 2000.

[CB97] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison Wesley Longman, 1997.

[Che04] Chaomei Chen. *Information Visualization — Beyond the Horizon*. (2nd edition), Springer Verlag, Berlin, Heidelberg, New York, 2004.

[Chia] Chisel Group. Creole Homepage. `http://www.thechiselgroup.org/creole`.

[Chib] Chisel Group. SHriMP Homepage. `http://www.thechiselgroup.org/shrimp`.

[Chi00] Ed H. Chi. A taxonomy of visualization techniques using the data state reference model. In *Proceedings of the Symposium on Information Visualization (InfoVis'00), Salt Lake City, UT*, Washington, DC, 2000. IEEE Press.

[CHM98] James H. Cross, T. Dean Hendrix, and Saeed Maghsoodloo. The control structure diagram: An overview and initial evaluation. *Empirical Software Engineering*, 3:131–158, 1998.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–18, 1990.

[CKN$^+$03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), San Diego, CA, June 11–13*, New York, NY, 2003. ACM Press.

[CM01] Andy Cockburn and Bruce McKenzie. 3D or not 3D? evaluating the effect of the third dimension in a document management system. In *Proceedings of CHI'2001 Conference on Human Factors in Computing Systems (Seattle, Washington, March 31-April 6)*, pages 434–441, New York, 2001. ACM Press.

[CM02] Andy Cockburn and Bruce McKenzie. Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *Proceedings of CHI'2002 Conference on Human Factors in Computing Systems (Minneapolis, Minnesota, 20–25 April)*, pages 203–210, New York, 2002. ACM Press.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, San Francisco, CA, 1999.

[CR93] Gruia-Catalin Cox and Kenneth C. Roman. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.

[Cro] James H. Cross. Grasp: Graphical representations of algorithms, structures, and processes. `http://www.eng.auburn.edu/grasp`.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[dBETT98] Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1998.

[DeM82] Tom DeMarco. *Controlling Software Projects – Management, Measurement and Estimation*. Yourdon Press Computing Series, Upper Saddle River, NJ, 1982.

[DF] Camil Demetrescu and Irene Finocchi. LEONARDO Webpage. `http://www.dis.uniroma1.it/~demetres/Leonardo`.

[DFS02] Camil Demetrescu, Irene Finocchi, and John T. Stasko. Specifying algorithm visualizations: Interesting events or state mapping? In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 16–30. 2002.

[DG02] Stephan Diehl and Carsten Görg. Graphs, they are changing – dynamic graph drawing for a sequence of graphs. In *Proceedings of 10th International Symposium on Graph Drawing*, pages 23–30, Irvine, CA, 2002. Springer.

[Die02a] Stephan Diehl. Future perspectives. In *Software visualization*, pages 347–353. Springer, Berling, Heidelberg, New York, 2002.

[Die02b] Stephan Diehl, editor. *Software Visualization*, volume 2269 of *LNCS State-of-the-Art Survey*. Springer Verlag, 2002.

[DK01] Stephan Diehl and Andreas Kerren. Levels of exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, NC*, pages 60–64, New York, NY, 2001. ACM Press.

[DPS97] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of 21st International Computer Software and Applications Conference COMPSAC'97*, pages 515–521, Washington, DC, 1997. IEEE Computer Society.

[dPS99] Wim de Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *Proceedings of European Symposium on Object-Oriented Programming ECOOP'99*, pages 116–134. Springer, Berlin, Heidelberg, New York, 1999.

[Dur] DuraSoft GmbH. RCE, VRCE, BDE . `http://wwwipd.ira.uka.de/~RCE`.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[EB02] Alexander A. Evstiougov-Babaev. Call graph and control flow graph visualization for developers of embedded applications. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 337–346. 2002.

[Ecl] Eclipse Foundation. Eclipse Homepage. `http://www.eclipse.org`.

[EGK$^+$04] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

[EHK$^+$03] Cesim Erten, Philipp J. Harding, Stephen G. Kobourov, Kevin Wampler, and Gary V. Yee. GraphAEL: Graph Animations with Evolving Layouts. In *Proceedings of the 11th Symposium on Graph Drawing (GD)*. Springer Verlag, 2003.

[Eic02] Holger Eichelberger. Evaluation-report on the layout facilities of UML tools. Technical Report TR 298, Institut für Informatik, Universität Würzburg, `http://www2.informatik.uni-wuerzburg.de/staff/eichelberger/reports/evalReport2002.pdf`, 2002.

[Eic03] Holger Eichelberger. Nice class diagrams admit good design? In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), San Diego, CA, June 11–13*, pages 159–167, New York, NY, 2003. ACM Press.

[ES80] K. Anders Ericsson and Herbert A. Simon. Verbal reports as data. *Psychological Review*, 87(3):215–251, 1980.

[ESJ92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner Jr. Seesoft TM — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[Fal02] Nils Faltin. Structure and constraints in interactive exploratory algorithm learning. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 213–226. 2002.

[FD04] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the International Conference on Software Engineering ICSE'04*, pages 387–396, Washington, DC, 2004. IEEE Computer Society Press.

[FO00] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[Fou] Apache Software Foundation. Byte code engineering library. `http://jakarta.apache.org/bcel/`.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1996.

[GAN] GANIFA. Interactive online textbook on the subject of generating finite automata. `http://www.cs.uni-sb.de/~ganimal/GANIFA/`.

[Gan00] Ganimal. Project homepage. `http://www.cs.uni-sb.de/GANIMAL`, 2000.

[GARG] University of Trier Graph Animation Research Group. DGD – dynamic graph drawing. `http://www.st.uni-trier.de/DGD`.

[GBPD04] Carsten Görg, Peter Birke, Mathias Pohl, and Stephan Diehl. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proceedings of 12th International Symposium on Graph Drawing, September 29–October 2, New York City, USA*, pages 228–238, Washington, DC, 2004. IEEE Computer Society Press.

[Ger94] Nahum D. Gershon. From perception to visualization. In L. Rosenblum, R.A. Earnshaw, J. Encarnacao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann, editors, *Scientific Visualization: Advances and Challenges*. Academic Press, 1994.

[GH03] John C. Grundy and John G. Hosking. SoftArch: Tool support for integrated software architecture development. *International Journal on Software Engineering and Knowledge Engineering*, 13(2):125–151, 2003.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM 98), Los Alamitos, CA*, pages 190–198, Washington, DC, 1998. IEEE Computer Society Press.

[GJK03a] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2003), Los Alamitos, CA*, pages 13–23, Washington, DC, 2003. IEEE Computer Society Press.

[GJK$^+$03b] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A New Approach for Visualizing UML Class Diagrams. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), San Diego, CA, June 11–13*, pages 179–188, New York, NY, 2003. ACM Press.

[GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance (ICSM'99), Oxford, UK*, pages 99–108, Washington, DC, 1999. IEEE Computer Society.

[GKMS00] Todd L. Graves, Alan F. Karr, James S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[Goo] GOOSE Homepage. `http://esche.fzi.de/PROSTextern/software/goose/index.html`.

[GP92] Thomas R. G. Green and Marian Petre. When visual programs are harder to read than textual programs. In G. C. van der Veer, M. J. Tauber, S. Bagnarola, and M. Antavolits, editors, *Proceedings of 6th European Conference on Cognitive Ergonomics (ECCE-6)*, Rome, Italy, 1992.

[GP96] Thomas R. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[Gre] Green Hills Software Inc. Timemachine suite. `http://www.ghs.com/products/timemachine.html`.

[GRR99] Martin Gogolla, Oliver Radfelder, and Mark Richters. Towards three-dimensional animation of UML diagrams. In *Proceedings of UML'99: The Unified Modeling Language – Beyond the Standard, Second International Conference, Fort Collins, CO*, pages 489–502. Lecture Notes in Computer Science, 1999.

[GS93a] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, Series on Software Engineering and Knowledge Engineering, volume 2, pages 1–39. World Scientific, Singapore, 1993.

[GS93b] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument, 1947. Part II, volume I of a report prepared for the U.S. Army Ord. Dept., reprinted in [Tau65].

[GYB04] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. In *Proceedings of the Australasian Symposium on Information Visualisation*, pages 53–59, Christchurch, 2004. ACM Press, New York, NY.

[GYK01] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 21st International Conference on Software Engineering ICSE 2001*, pages 265–274, Washington, DC, 2001. IEEE Computer Society Press.

[Hai59] Lois M. Haibt. A program to draw multi-level flowcharts. In *Proceedings of the Western Joint Computer Conference*, pages 131–137, San Francisco, CA, 1959.

[HBE96] Christopher G. Healey, Kellogg S. Booth, and James T. Enns. High-speed visual estimation using preattentive processing. *ACM Transactions on Computer-Human Interaction*, 3(2):107–135, 1996.

[HDS02] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

[HK01] Jiawei Han and Micheline Kamber, editors. *Data Mining – Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 2001.

[Hop74] F. Robert A. Hopgood. Computer animation used as a tool in teaching computer science. In *Proceedings of the IFIP Congress*, pages 889–892, Stockholm, Sweden, 1974.

[IBMa] IBM Coorporation, alphaWorks Technology. Jikes Bytecode Toolkit. `http://www.alphaworks.ibm.com/tech/jikesbt`.

[IBMb] IBM Coorporation, alphaWorks Technology. JINSIGHT. `http://www.alphaworks.ibm.com/tech/jinsight`.

[IBMc] IBM Coorporation, alphaWorks Technology. Web Services Navigator. `http://www.alphaworks.ibm.com/tech/wsnavigator`.

[ID90] Alfred Inselberg and Bernhard Dimsdale. Parallel coordinates: A tool for visualizing multi-dimensional geometry. In *Proceedings of Visualization '90, San Francisco, CA*, pages 361–378, Washington, DC, 1990. IEEE Computer Society Press.

[IEE98] IEEE. *IEEE Standard for a Software Quality Metrics Methodology.* IEEE Standard 1061. IEEE Computer Society press, Washington, DC, 1998.

[IEE00] IEEE. *IEEE recommended practice for architecture description.* IEEE Standard 1471. IEEE Computer Society Press, Washington, DC, 2000.

[IW00] Pourang P. Irani and Colin Ware. Diagrams based on theories of structural perception. In *Proceedings of Working Conference on Advanced Visual Interfaces AVI, Palermo, Italy*, pages 61–67, New York, NY, 2000. ACM Press.

[IW04] Pourang Irani and Colin Ware. The effect of a perceptual syntax on the learnability of novel concepts. In *Proceedings of Information Visualisation, Eighth International Conference on (IV'04)*, pages 308–314, Washington, DC, 2004. IEEE Computer Society.

[IWT01] Pourang P. Irani, Colin Ware, and Maureen Tingley. Using perceptual syntax to enhance semantic content in diagrams. *IEEE Computer Graphics & Applications*, 21(5):76–84, 2001.

[Jac75] Michael Jackson. *Principles of Program Design.* Academic Press, 1975.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 22rd International Conference on Software Engineering ICSE 2002, Orlando, FL*, pages 467–477, New York, NY, 2002. ACM Press.

[Joh] John T. Stasko. SAMBA Web page. `http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html`.

[JOH04] James A. Jones, Alessandro Orso, and Mary Jean Harrold. Gammatella: Visualizing program-execution data for deployed software. *Information Visualization*, 3(3):173–188, 2004.

[JPH$^+$99] Christopher Johnson, Steven G. Parker, Charles Hansen, Gordon L. Kindlmann, and Yarden Livnat. Interactive simulation and visualization. *Computer*, 32(12):59–65, 1999.

[Jr.96] Frederick P. Brooks Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of IEEE Visualization Conference*, pages 284–291, San Diego, CA, 1991.

[JSW05] Dierk Johannes, Raimund Seidel, and Reinhard Wilhelm. Algorithm animation using shape analysis: Visualising abstract executions. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS '05), St. Louis, MI*, pages 17–26, New York, NY, 2005. ACM Press.

[KA98] Jeffrey L. Korn and Andrew W. Appel. Traversal-based visualization of data structures. *In IEEE Information Visualization '98*, 1998. Also available at `http://www.cs.princeton.edu/~jlk/viz`.

[KB04] Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *Proceedings of 10th International Software Metrics Symposium METRICS 2004*, 2004.

[KC97] Hideki Koike and Hui-Chu Chu. VRCS: Integrating version control and module management using interactive three-dimensional graphics. In *Proceedings of IEEE Symposium on Visual Languages VL'97, Capri, Italy*, pages 168–173, Washington, DC, 1997. IEEE Computer Society Press.

[KDvV02] Henk Koning, Claire Dormann, and Hans van Vliet. Practical guidelines for readability of IT-architecture diagrams. In *Proceedings of the 20th Annual International Conference on Documentation (SIGDOC 2002)*, pages 90–99, New York, NY, October 2002. ACM Press.

[Kei02] Daniel A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):100–107, 2002.

[Ker04] Andreas Kerren. Learning by generation in computer science education. *Journal of Computer Science and Technology (JCS&T)*, 4(2):84–90, 2004.

[KM99] Claire Knight and Malcolm Munro. Comprehension within virtual environment visualisations. In *Proceedings of Seventh International Workshop on Program Comprehension (IWPC'99)*, pages 4–11, Pittsburgh, PA, 1999.

[Kno66] Kenneth Knowlton. Bell telephone laboratories low-level linked list language. 16-minute black and white film, 1966.

[Knu63] Donald E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6(9):555–563, 1963.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Knu92] Donald E. Knuth. *Literate Programming.* Center of the Study of Language and Information – Lecture Notes, No. 27. CSLI Publications, Stanford, CA, 1992.

[Kos02] Rainer Koschke. Software Visualization for Reverse Engineering. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 138–150. 2002.

[KS02] Andreas Kerren and John T. Stasko. Algorithm Animation. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 1–15. 2002.

[KST98] Kai Koskimies, Tarja Systä, and Jyrki Tuomi. Automated support for modeling OO software. *IEEE Software*, 15(1):87–94, 1998.

[KST02] Ari Korhonen, Erkki Sutinen, and Jorma Tarhio. Understanding algorithms by means of visualized path testing. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 256–268. 2002.

[KvdWW01] Ernst Kleiberg, Huub van de Wetering, and Jarke J. Van Wijk. Botanical visualization of huge hierarchies. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, pages 87–94, Washington, DC, 2001. IEEE Computer Society.

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs – Methods and Models.* Springer, Berlin, Heidelberg, New York, 2001.

[Lab] Cigital Labs. VISTA tool. `http://www.cigitallabs.com/research/demos/vista/`.

[Lan] Michele Lanza. CodeCrawler Homepage. `http://www.iam.unibe.ch/~scg/Research/CodeCrawler/index.html`.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of International Workshop on Principles of Software Evolution IWPSE 2001*, pages 37–42, 2001.

[LD01] Michele Lanza and Stephane Ducasse. A categorization of classes based on the visualization of their internal structure: The class blueprint. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'2001*, pages 300–311, New York, NY, 2001. ACM Press.

[Leh80] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE (Special Issue on Software Engineering)*, 68(9):1060–1076, 1980.

[Lem92] Karen A. Lemone. *Design of compilers: techniques of programming language translation.* CRC Press, Boca Raton, FL, 1992.

[LH92] Haim Levkowitz and Gabor T. Herman. Color scales for image data. *IEEE Computer Graphics and Applications.*, 12(1):72–80, 1992.

[LJ80] George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, Chicago, 1980.

[LS87] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth 10,000 words. *Cognitive Science*, 11(1):65–99, 1987.

[Mad94] Kim Halskov Madsen. A guide to metaphorical design. *Communications of the ACM*, 37(12):57–62, 1994.

[Mar] Adrian Marcus. SV3D Web page. `http://www.sv3d.org`.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[McC93] Deidre A. McConathy. Evaluation methods in visualization: Combating the emperor's new clothes phenomenon. *SIGBIO Newsletter*, 13(1):2–8, 1993.

[Meh02] Katharina Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 163–175. 2002.

[MELS95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[MFM03] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2003), San Diego, CA, June 11–13*, pages 27–36, New York, NY, 2003. ACM Press.

[MHvS05] Paul McIntosh, Margaret Hamilton, and Ron G. van Schyndel. X3D-UML: enabling advanced UML visualisation through X3D. In *Proceedings of the Tenth International Conference on 3D Web Technology (Web3D 2005), Bangor, UK*, pages 135–142, New York, NY, 2005. ACM Press.

[MLMD01] Jonathan I. Maletic, Jason Leigh, Andrian Marcus, and Greg Dunlap. Visualizing object-oriented software in virtual reality. In *Proceedings of Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada*, pages 49–54, Washington, DC, 2001. IEEE Computer Society Pres.

[Mos01] Yiannis N. Moschovakis. What is an algorithm? In Bjorn Engquist and Wilfried Schmid, editors, *Mathematics Unlimited – 2001 and Beyond*, pages 919–936. Springer, Berlin, Heidelberg, New York, 2001.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[MP95] Karl-Heinrich Moeller and Daniel J. Paulish. An empirical investigation of software fault distribution. In Norman Fenton, Robin Whitty, and Yoshinori Lizuka, editors, *Software Quality Assurance and Measurement: Worldwide Perspective*, pages 242–253. International Thomson Computer Press, London, 1995.

[MRC91] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: detail and context smoothly integrated. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems CHI'91*, pages 173–176, New York, NY, USA, 1991. ACM Press.

[MvWvL99] Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, 1999.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* Wiley, Chichester, 1979.

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

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, Berlin, Heidelberg, New York, 1999.

[NRA$^+$03] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, 2003.

[NS73] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8):12–26, August 1973.

[Opp80] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages*, 2(4):465–483, 1980.

[Pai90] Allan Paivio. *Mental Representations: A Dual Coding Approach.* Oxford University Press, New York, 1990.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PBG03] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3D metaphor for software production visualization. In *Proceedings of Seventh International Conference on Information Visualization (IV'03)*, pages 314–319, 2003.

[PBS93] Blaine A. Price, Ronald Baecker, and Ian Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In Franz J. Brandenburg, editor, *Graph*

Drawing (Proceedings of GD '95), volume 1027 of *Lecture Notes Computer Science*, pages 435–446, Berling, Heidelberg, New York, 1996. Springer.

[PJM$^+$02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of Java programs. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 151–162. 2002.

[PKM06] Wim De Pauw, Sophia Krasikov, and John F. Morar. Execution patterns for visualizing web services. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis 2006), Brighton, UK*, 2006.

[PLL04] Thomas Panas, Jonas Lundberg, and Welf Löwe. Reuse in reverse engineering. In *Proceedings of the 12th International Workshop On Program Comprehension, Bari, Italy*, pages 52–61, Washington, DC, 2004. IEEE Computer Society Press.

[PLRW92] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: A method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741–773, 1992.

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

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.

[PPV00] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering ICSE'00, Future of SE Track, Limerick, Ireland*, pages 345–355, New York, NY, 2000. ACM Press.

[PR94] Stephen Palmer and Irvin Rock. Rethinking perceptual organization: The role of uniform connectedness. *Psychonomic Bulletin and Review*, 1(1):29–55, 1994.

[Pro02] Christian Probst. *A Demand-Driven Solver for Constraint-Based Control Flow Analysis.* PhD thesis, University of Saarland, Saarbrücken, Germany, 2002.

[PRW03] Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualization tools. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 1095–1350, Washington, DC, 2003. IEEE Computer Society Press.

[Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, Berlin, Heidelberg, New York, 1997. Springer.

[RC94] Ramana Rao and Stuart K. Card. The table lens: Merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems CHI'94, Boston, MA*, pages 318–322, New York, NY, 1994. ACM Press.

[RG00] Oliver Radfelder and Martin Gogolla. On better understanding UML diagrams through three-dimensional visualization and animation. In *Proceedings of Advanced Visual Interfaces (AVI'2000), Palermo, Italy*, pages 292–295, New York, NY, 2000. ACM Press.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of ACM Computer-Human Interaction '91 Conference on Human Factors in Computing Systems*, pages 189–194, New York, 1991. ACM Press.

[ROC97] Ronald A. Rensink, J. Kevin O'Regan, and James J. Clark. To see or not to see: The need for attention to perceive changes in the senses. *Psychological Science*, 8(5):368–373, 1997.

[San96] Georg Sander. *Visualization Techniques for Compiler Construction.* Dissertation (in german), University of Saarland, Saarbrücken, Germany, 1996.

[Sca89] David A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(5):28–36, 1989.

[Sco58] Art E. Scott. Automatic preparation of flow chart listings. Journal of the ACM. *International Business Machines Corporation*, 5(1):57–66, January 1958.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization – Programming as a Multimedia Experience.* MIT Press, 1998.

[She67] Roger N. Shepard. Recognition memory for words, sentences, and pictures. *Journal of Verbal Learning and Behavior*, 6:156–163, 1967.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of 1996 IEEE Conference on Visual Languages, Boulder, CO*, pages 336–343, Washington, DC, 1996. IEEE Computer Society Press.

[SK93] John Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[SKM01] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba – an environment for reverse engineering Java software systems. *Software – Practice and Experience*, 31(4):371–394, 2001.

[SMC74] Wayne Stevens, Glenford Myers, and Larry Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SMMH77] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20(6):373–381, 1977.

[Sof] Software-Tomography GmbH. Sotograph homepage. `http://www.software-tomography.com/html/sotograph.htm`.

[Sol] Solomon Duskis et. al. JSAMBA Web page. `http://www.cc.gatech.edu/gvu/softviz/algoanim/jsamba`.

[Spe68] Roger W. Sperry. Hemisphere disconnection and unity in conscious awareness. *American Psychologist*, 23(10):723–733, 1968.

[Spe01] Robert Spence. *Information Visualization.* Pearson Education, Harlow, 2001.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, St. Petersburg Beach, FL*, pages 1–50, New York, NY, 1996. ACM Press.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape-analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Antonio, TX*, pages 105–118, New York, NY, 1999. ACM Press.

[Sta] John Stasko. Polka. `http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html`.

[Sta73] Lionel Standing. Learning 10,000 pictures. *Quartertly Journal of Experimental Psychology*, 25(2):207–222, 1973.

[Sta90a] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.

[Sta90b] John T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[Sta97] John Stasko. Using student-built algorithm animations as learning aids. In *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997. Extended version including SAMBA documentation available as Technical Report GIT-GVU-96-19 from Georgia Institute of Technology at `ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/96-19.ps.Z`.

[Str] Strata Inc. WinCVS homepage. `http://www.cvsgui.org`.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[Sun90] Vaidy Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, 1990.

[SW01] Ben Shneiderman and Martin Wattenberg. Ordered treemap layouts. In *Proceedings of IEEE Symposium on Information Visualization INFOVIS'01*, pages 73–78, Washington, DC, 2001. IEEE Computer Society Press.

[SW05] Dabo Sun and Kenny Wong. On evaluating the layout of UML class diagrams for program comprehension. In *Proceedings of the 13th International Workshop on Program Comprehension IWPC 2005*, pages 317–326, Washington, DC, 2005. IEEE Computer Society Press.

[SZ00] John Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the Symposium on Information Visualization (InfoVis'00), Salt Lake City, UT*, pages 57–65, Washington, DC, 2000. IEEE Computer Society Press.

[Tar] Tarantula. `http://www.cc.gatech.edu/aristotle/Tools/tarantula`.

[Tau65] Abraham H. Taub, editor. *John von Neumann: Collected Works*, volume V: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon, New York, NY, 1965.

[Tho81] Carsten Thomassen. Kuratowski's theorem. *Journal of Graph Theory*, 5:225–241, 1981.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, May 1998.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, January 1995.

[TM02] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the Workshop on Visualizing Software for Understanding and Analysis VISSSOFT 2002, Paris, France, June*, pages 43–50, Washington, DC, 2002. IEEE Computer Society Press.

[TS96] Scott R. Tilley and Dennis B. Smith. Coming attractions in program understanding. In Alan W. Brown, editor, *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. Wiley, New York, NY, 1996.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Viz] Vizz3D. `http://vizz3d.sourceforge.net/`.

[VT04] Brian C. Verrelli and Sarah A. Tishkoff. Signatures of selection and gene conversion associated with human color vision variation. *American Journal of Human Genetics*, 75(4):363–375, 2004.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: Visualization of code evolution. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS '05), St. Louis, MI*, pages 47–56, New York, NY, 2005. ACM Press.

[VW93] Paul F. Velleman and Leland Wilkinson. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician*, 47(1):65–72, February 1993.

[vWvdW99] Jarke J. van Wijk and Huub van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proceedings of IEEE Symposium on Information Visualization INFOVIS'99*, pages 73–78, San Francisco, 1999.

[W3C] World Wide Web Consortium W3C. Web Services Architecture Specification. `http://www.w3.org/2002/ws`.

[War00] Colin Ware. *Information Visualization — Perception for Design*. Morgan Kaufmann, San Francisco, CA, 2000.

[Wat93] Alan Watt. *3D Computer Graphics*. Addison-Wesley, Reading, MA, 1993.

[Wer23] Max Wertheimer. Untersuchungen zur Lehr von der Gestalt (in German). *Psychologische Forschung*, 4:301–350, 1923.

[WF94] Colin Ware and Glenn Franck. Viewing a graph in a virtual reality display is three times as good as a 2D diagram. In *Proceedings of the IEEE Symposium on Visual Languages (VL'94), St. Louis, MO*, pages 182–183, Washington, DC, 1994. IEEE Computer Society Press.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley Longman, Redwood City, CA, 1995.

[WMS02] Reinhard Wilhelm, Tomasz Müldner, and Raimund Seidel. Algorithm explanation: Visualizing abstract states and invariants. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 381–394. 2002.

[Wol98] Jeremy M. Wolfe. Visual memory: What do you know about what you saw? *Current Biology*, 8:303–304, 1998.

[WPCM02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.

[WRLP94] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. The cognitive walkthrough: A practitioner's guide. In Jakob Nielsen and Robert L. Mack, editors, *Usability Inspection Methods*, pages 105–139. Wiley, New York, NY, 1994.

[XRa] XRadar. `http://xradar.sourceforge.net/`.

[Xsl] X-slice. `http://xsuds.argreenhouse.com/html-man/xslice.html`.

[ZB89] Horst Zuse and Peter Bollmann. Software metrics – using measurement theory to describe the properties and scales of static software complexity metrics. *SIGPLAN Notices*, 24(8):23–33, 1989.

[ZDZ03] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution IWPSE September 2003, Helsinki, Finland*, pages 73–83, Washington, DC, 2003. IEEE Computer Society Press.

[Zel01] Andreas Zeller. Datenstrukturen visualisieren und animieren mit DDD (in German). *Informatik – Forschung und Entwicklung*, 16(2):65–75, 2001.

[Zha03] Kang Zhang, editor. *Software visualization – From theory to practice*, volume 734 of *International Series in Engineering and Computer Science, volume 134*. Springer, Berlin, Heidelberg, New York, 2003.

[ZL96] Andreas Zeller and Dorothea Lütkehaus. DDD – A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.

[ZWDZ05] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

[ZZ02] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Proceedings of Dagstuhl Seminar on Software Visualization [Die02b]*, pages 191–204. 2002.

Index