Software Visualization

Abstract The Software Visualization (SV) discipline investigates approaches and techniques for static and dynamic graphical representations of algorithms, programs (code) and the processed data. SV is concerned primarily with analysis of programs and their development. The goal is to improve our understanding of inherently invisible and intangible software, particularly when dealing with large information spaces that characterize domains like software maintenance, reverse engineering, and collaborative development. The main challenge is to find effective mappings from different software aspects to graphical representations using visual metaphors. This paper provides an overview of the SV research, describes current research directions and includes an extensive list of recommended readings.

1 Introduction

Software Visualization (SV) can be defined as “a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration” [60]. SV refers to the visualization of computer programs and algorithms [108] and attempts to give physical shape to shapeless or intangible software that “disappears into disks.” The goal is to provide better comprehension of software artifacts [5].

The use of SV raises the questions [2]: “What can be visualized? How? For what reasons?” The effectiveness of SV is also a basic question. Research over the years has visualized different aspects of source code, the code itself, data flow and run-time behavior. SV has been applied for various areas like algorithm animation [20; 34], software engineering, concurrent program execution [23], static and dynamic visualizations of object oriented code [81; 89], fault diagnostics [1; 95], debugging [4] and requirement analysis [24], to name a few. An extensive compilation of research work relating to these fields can be found in [28; 58; 108].

A number of taxonomies have been developed that identify the properties of SV systems [79; 85; 86; 94]. Attributes defined by Roman and Price [86; 94] include:

- Scope and content: What is the aspect of the program being visualized?
- Abstraction: What kind of information is conveyed by the visualization?
- Form and technique: How is the graphical information being conveyed?
- Method: How is the visualization specified?
- Interaction: How can the user interact with the visualization?

Stasko and Patterson [106] identify an additional property, the level of automation provided for developing the SV system.

A task oriented view of SV [72; 75] uses the argument that no single SV tool or technique can address all visualization tasks. It is, therefore, necessary to identify the most appropriate visualization technique for the given SV task based on the SV dimensions [72]:

- Tasks: Why is the visualization needed?
- Audience: Who will use the visualization?
- Target: What is the data source to represent?
2 Evolution of software visualization

SV has progressed from using simple two-dimensional (2D) graphs [8; 80; 105; 107; 124] to three-dimensional (3D) representations [75; 76; 91] and more recently, virtual environments (VEs).

2.1 2D and 3D visualization

2D SV techniques typically involve graph or tree-like representations consisting of a large number of nodes and arcs [118]. A complex software system might include thousands of such nodes and arcs. To make conceptualization and comprehension easy for the user, visualizations of such systems present pieces of the graph in different views or different windows so that the user can focus on the level of detail he desires. The software system is therefore represented in multiple windows that present to the observer different characteristics of the system under consideration. Some examples of such visualization systems include Seesoft [38], SHriMP [112], GROOVE [59] and FIELD [108].

2D visualizations may lead to cluttering a plethora of information on a flat plane. Even though pan/zoom and fisheye views have been explored [112], visualizing software in 2D does introduce a cognitive overload by presenting too much information. Stasko [104] identifies the need for an extra spatial dimension in visualizations and states that "by adding an extra spatial dimension, we supply visualization designers with one more possibility for describing some aspect of a program or system." As an example of the advantages of using 3D visualizations, the Cone Tree concept developed at Xerox Parc [104; 121] can be considered. It has been claimed that the cone tree can display up to one thousand nodes without visual clutter, which is far beyond the capabilities of a 2D visualization. The developed 3D visualization presents structured information such as computer directories and project plans. In line with representing the execution time behavior of object-oriented code [59] in two dimensions, Stasko [104] discusses the development of a system called POLKA-3D to represent the same as a 3D animation.

Ware et al. [121] developed a system called GraphVisualizer3D to visualize object-oriented code in three dimensions. They suggest that perception is less error-prone if software objects are mapped to visual objects, as there is a natural mapping from the former to the latter. They present the results of experiments that analyzed perception in 2D and 3D and conclude that there is encouraging empirical evidence that error rates in perception are less in 3D visualization. One major advantage of 3D visualization is that it allows a user to perceive the depth of a presented structure. With 3D visualization, users can zoom-in or walk around structures or choose another angle (by rotating the design) and it is possible that hidden structures in a software system become evident. 3D visualization might help identify new metaphors, fostering new ideas with respect to design principles [41]. The hierarchy of relations and dependencies in design or source code would also become more readily apparent, because of the added depth. It can also help to faster develop the "mental model" in the mind of the user.

Another example of visualizing large nested graphs is the NV3D system [84] that has been tested with graphs containing more than 35,000 nodes and 100,000 relationships. The NV3D system uses techniques like rapid zooming, elision and 3D interactive visualization to display nested graphs. The NV3D system and the POLKA-3D system [104] analyze issues like spatial navigation, layout, semiotics and the common uses of the third dimension to represent characteristics like value, structure position, history of computation, state of computation and aesthetics to refine the appearance of a 3D visualization.

3D visualization have been explored for all areas where 2D visualizations were used, including metrics based visualization of object-oriented programs and visualization to track software errors, isolate problems and monitor progress of development [18; 19; 67]. 3D UML (Unified Modeling Language) representations have also been researched [37].

2.2 Virtual environments

VEs open possibilities of “immersion” and “navigation” that may help to explore software structure better. VEs enable the user to interact with a representation of something familiar, namely a world with familiar objects that he/she can interact with. The concept of “worlds” in a VE can be mapped to “entities” or “components” in object-oriented code or a software system. It is possible that all software artifacts from requirements to source code can be represented and linked in a VE to improve comprehension. VEs would enable users to navigate through these links faster and in a more intuitive manner than 2D representations or even 3D structures.

Software systems are large complex systems composed of multiple components. To effectively comprehend these systems, it is necessary to provide varying levels of detail. Any user attempting to understand the system must be able to zoom-out and zoom-in to each level of detail as necessary. 3D visualizations and VEs allow a user to concentrate on one aspect of the world in detail while providing a distant view of other aspects that are situated farther away. As the user moves close to each entity or visual component, it comes to "life" or presents a higher level of detail. This technique, called elision, is a major property of VEs, which abstracts distant objects and details closer objects. The user can move back and forth between objects or structures in this world, and rotate them around to view information that might be hidden from normal view.

Examples of SV systems that use VEs for representing object-oriented software systems are ImsoVision [71] and
Software World [60]. The former represents C++ code in an immersive VE while the latter does the same for static Java code. A major characteristic of both systems is the mapping of static properties of object-oriented code to objects in the VE. ImsoVision uses geometrical 3D shapes like platforms, spheres, horizontal and vertical columns as visual metaphors for the characteristics of C++ code, while Software World uses real-world metaphors like a world, countries, districts and buildings as visual metaphors for the various parts of Java code. An example of elision can be seen in the ImsoVision system [71], which hides the private attributes of an object under the platform that represents the object. The private attributes are visible only when the user rotates the platform around.

Both visualization systems visualize only static properties of code. They cannot be used to characterize the runtime behavior of an object-oriented system. While it is evident that VEs provide a far richer experience than 2D visualizations for a user attempting to comprehend a software system, it is necessary to further investigate metaphors and representations that allow us to move beyond visualizing static code [2].

2.3 Distributed virtual environments

A distributed (networked) VE (DVE or net-VE) is a software system in which multiple users interact with each other in real-time, even though those users may be physically located around the world [99]. The users have a shared sense of space, a shared sense of presence, a shared sense of time, a way to communicate and a way to share [99]. DVEs can be used for collaborative SV-based applications dealing with large and distributed software projects including coding, maintenance, and interactive visualization [2].

WYSIWIS (What You See Is What I See) is the basic abstraction that guides such kind of multi-user interface design provides the sense of teamwork. WYSIWIS is crucial for collaboration, however, some research has indicated that strict objectivity is too inflexible. It may actually hinder collaboration in some cases since the users are forced to agree on a common representation and can only see the same things, instead of being able to tailor their representation of the virtual scene to meet their needs [103; 109; 110].

Indeed, collaboration in the real world often proceeds without the participants having access to the same information. This has lead to the development of the Relaxed-WYSIWIS concept. Snowdon introduced the term “subjective views” for the concept of multiple perspectives in VEs [103]. A subjective VE can give the user ability to control the presentation style to best suit their working needs.

SOLVEN is a model to support subjective views [100]. The core feature of SOLVEN is an access matrix, which defines the representation of individual objects for individual users. The matrix defines an object’s view in terms of two independent factors, appearance (differing geometric definition) and modifier (highlighting and de-emphasizing abilities).

VR-VIBE is a multi-user 3D visualization of a collection of documents or document references [102]. The visualization is structured using a 3D spatial framework of keywords, called Points of Interest or POIs.

Griminstead et al. describe the use of a distributed, collaborative grid enabled visualization environment [48]. The Resource-Aware Visualization Environment (RAVE) was deployed as Web Services and accessed using a simple PDA. Experiments like this one clear the path for large scale, multi-user SV based collaborative applications.

3 Metaphors in software visualization

Building on ideas stated in the previous section, a metaphor can be defined as “a rhetoric figure whose essence is understanding and experiencing one kind of thing in terms of another” [63]. Metaphors in the medium of representation affect the expressiveness of the visualization. Metaphors might be abstract geometrical shapes (as in ImsoVision, NV3D, GraphVisualizer3D) or they might be real-world entities (as in Software World). While it is true that a user would be more familiar with a real-world metaphor, the visual complexity of the metaphor should not affect the effectiveness of the visualization. Roman and Cox [93] represent the role of the visual metaphor in a program visualization as shown in Figure 1 [2].

Fig. 1 Mapping software to a graphical representation

Evidently, a metaphor is the entity that gives shape to the different faces of intangible software [2]. The next questions in investigating VE visualization techniques are: What are the characteristics of an effective metaphor for a VE and What are the desirable characteristics of a VE for visualizing an object-oriented system? It is necessary to state the desirable properties of a metaphor for SV in 2D, 3D or VEs. While considering the properties of a metaphor, issues that arise regarding the characteristics of the SV system can also be discussed.

Mackinlay discusses graphical design issues on the basis of two criteria: expressiveness and effectiveness [70]. Expressiveness is the medium used to express the graphical representation and effectiveness is the extent to which the representation is effective for comprehension of the visualized information. These two criteria form the basis on which we propose our design issues for effective visualizations. To be effective and meaningful, any visualization system should consider the following key areas.

1. **Scope of the representation**: Visualizing complex, real-time systems can create chaos if the scope of the SV is
not defined. Scope, as identified by Price et al. [86], is
isolating the characteristics of the system that the vi-
sualization will address. The SV might choose to ad-
address static or dynamic features of the software. It might
choose to represent control flow or data flow or depen-
dencies or all three. For example, visualization of Java
source code might address the classes in other packages
that a particular class depends on or inheritance hierar-
chies for a class or interface dependencies for a class, to
state a few.

2. **Medium of representation:** The type of information be-
ing visualized and the level of detail required in the visu-
alization are just two factors that dictate the type of out-
put medium needed. If the system to be visualized is rel-
atively small and if detail like complexity of the source
code, version history, detailed dependency navigation or
linking of the graphical representation to source code is
not needed. A simple 2D graph is sufficient. If however,
the system to be visualized should state in detail infor-
mation like security vulnerabilities in the code and de-
sign or if the representation should present varying levels
of information about the system without overwhelming
the user, then 3D visualizations might be considered.

3. **The Visual Metaphor:** Metaphors in the medium of rep-
resentation affect the expressiveness of the visualization.
Metaphors might be abstract geometrical shapes (as in
ImsoVision, NV3D, GraphVisualizer3D and other 2D rep-
resentations) or they might be real-world entities (as in
Software World). The visual complexity of the metaphor
should not affect the expressiveness of the visualization.
However, in case of DVEs [10], users might feel more
comfortable interacting with their colleagues in a real-
world immersive VE.

(a) **Consistency of the Metaphor:** The metaphor or the
mapping from software artifacts to the representa-
tions should be consistent throughout the visualiza-
tion. Multiple software artifacts cannot be mapped to
the same metaphor. Similarly, a software artifact cannot be
mapped to multiple metaphors. In a VE, the metaphor
should be consistent with the world it is present in.

(b) **Semantic Richness of the Metaphor and complexity:**
The metaphor chosen should be rich enough to pro-
vide mappings for all aspects of the software that
need to be visualized. The scope of the representa-
tion determines to a certain extent the nature of the
metaphor to be chosen. There should be enough ob-
jects or equivalent representations in the metaphor
for the software entities that need to be visualized.
The SV should not divert the user from the infor-
mation that the SV system attempts to convey. The VE
should provide pertinent representations without giving
the user the impression of immersion in endless space. Similar views can be found in [123].

4. **Abstractness:** The user of the visualization system
should be able to focus away from certain parts of the
representation and focus in detail on certain parts of the
representation. This is the property of elision (used in
NV3D [84]) that permits different users to focus on the
level of detail they desire. For example, if a visualiza-
tion system should aid an evaluator in discovering secu-
rity vulnerabilities, the evaluator would look for different
levels of detail (say, low-level representations that map
to source code) as opposed to a user who will be inter-
ested only in visualizing if any security problems exist in
the system. This ability to zoom-in and zoom-out is what
makes navigation through a 3D system easier than under-
standing a 2D representation. Roman and Cox [93] iden-
tify different levels of abstractedness, namely direct rep-
resentation, structural representation, synthesized repre-
sentation and analytical representation.

5. **Ease of Navigation and Interaction:** Ease of naviga-
tion is obviously a major design issue when constructing
a visualization. The user should understand what is pre-
sented and what level of abstraction in the system the
user is currently at. It should be easy for the user to
move back and forth between different views or differ-
ent worlds (in the case of VEs). Also, the nature of the
medium of representation would affect the level of nav-
gitation a user expects to have in the visualization. 3D
visualizations should allow users to rotate the entities
around for different angles of view. It should be possi-
able to hide or “close” objects that are not of interest by
clicking on them or interacting with them in other ways.

6. **Level of Automation:** Automation specifies the level
that the construction of the software visualization sys-
tem is automatic. Effective visualizations would need to
be fully automated for SV to be more widely used.

### 4 Software visualization tools and applications

Based on concepts and developments in information visual-
ization [14; 117], usability [77] and software engineer-
ing [16; 42; 68], new SV frameworks, notations [22], query
languages [87] and techniques are proposed [26; 29; 32].
New SV models enable interactive, on-line configuration of
SV views and mappings (Vizz3D [83]) and better support
for software comprehension [12; 82; 116].

The *rube* framework presents models (multimodels) and
their visualizations that are based on user-specified metaphors
and aesthetics [53]. The *RiOT* framework can be used to
manage testing, and provide dynamic visualization of het-
erogeneous distributed component-based applications [46].
The *Source Viewer 3D* (*sv3D*) uses a 3D metaphor to rep-
resent software system and analysis data that extends the
Seesoft pixel metaphor by rendering the visualization in a
3D space [73].

There are many SV tools developed for specific aspects
of software design and development [113; 115]. CodeCrawler
is an example of a lightweight software visualization tool
which combines metrics information with its visualizations
[64; 65]. SV tools can be integrated within an integrated de-
velopment environment (IDE) such as Eclipse [69]. SV tools
can also be accessed on the Web [31] and presented as Web services [33].

Object-oriented aspects are often a topic of SV research [51]. That includes evolution of class hierarchies [47], versioning [11; 96], run-time visualization [101] metrics [57] and component-based software [40]. That also includes C++ [62] and Java [17; 45; 88] programming languages, as well as UML [54; 74].

Other SV areas include formalisms [3], metrics [9; 66; 98], slicing [27; 92; 90], and XML [52; 78; 114], to name a few. The remainder of this section discusses SV for software evolution, software security, data mining in software systems, algorithms and software engineering education.

4.1 Software evolution

Software is continually changing and evolving [39]. Today’s typical software system is a complex beast spanning millions of lines of code. Manually analyzing the effects and impacts of changes [56] to a software system is a labor-intensive and often error-prone task. Visualizing the evolution of the system may be accomplished, in part, through visualizing the version history of a software system. Visualizing version history typically involves visualizing metrics such as number of lines of code in a particular version of a file included in the system, the percentage of growth and the percentage of change in the system, defect density and change complexity measures [44]. This section discusses advances in version history visualization.

An important construct in most of the works discussed below is that of a Modification Request or Maintenance Request (MR). A software system is assumed to consist of subsystems. Each of these subsystems has a number of modules. The modules include the program elements, which may be a collection of one or more source files, and an MR is the information representing work to be done to each module. Deltas are part of an MR, representing editing changes made to individual files in order to complete an MR. A file can be checked out, edited and then checked in [6; 43]. Note that, this terminology works well with version control systems that can record the parent MR for each delta list, along with the number of lines added, deleted and modified by that change. Alternatively, as in the case of CVS, there is no concept of a MR. Changes made to files are recorded as part of a checkout or update of modules.

The forerunner to most of the attempts at version history visualization can be seen in Seesoft, developed by Eick et al. [38]. Seesoft is a tool for visualization of line-oriented software statistics. Seesoft can visualize up to 50,000 lines of code and provides information about various statistics like the number of lines under version control, the age of each line of code in a file, the number of lines of code in each file, the MR that touched a particular line of code in a file and the number of times the line was executed during testing. Seesoft uses a row-column metaphor. Each column represents a file and the rows in each column represent the number of lines of code in the file. It allows user interaction to decipher interesting patterns in the version history. It also provides information about the dates of changes, the reasons for changes, the developer who changed the code, etc.

Another significant effort in version history visualization is presented by Gall et al. in [44]. Their work uses color and the third dimension effectively to visualize software release histories. The metrics that they visualize include size of the system in terms of lines of code, age in terms of version numbers and error-proneness in terms of defect density. Their Software Release History visualization is composed of three entities:

- Time: The visualization is grouped according to the Release Sequence Number (RSN). A snapshot of the system at the time of each release enables the end-user to see the evolution of files between releases. Addition, deletion and modification of files between releases are clearly visible.
- Structure: The system is decomposed into subsystems. Each subsystem is decomposed into modules and each module comprises the source code files.
- Attributes: These include version number, size, change complexity and defect density.

The visualization was created using Java and Virtual Reality Modeling Language (VRML) to render and navigate the 3D spaces. The end user can navigate through the visualization and use the mouse to extract information about the structure of the entire system for a release, or focus on a particular subsystem and extract the values of the modules in the subsystem. The paper concludes with the suggestion that other metrics like lines of code, complexity measures and defect density can be visualized. It also suggested the detection automatic detection of change patterns to identify module dependencies. The type of change pattern to be investigated could be input by the user.

Gall et al. discuss another application of version history visualization [43]. They present an approach that uses information in the release history of a system to uncover logical dependencies and change patterns among modules. They have developed a technique that automatically extracts information about the logical dependencies among the modules of a system. These logical dependencies are different from the syntactic dependencies that are evident through source code analysis. They propose the idea of Change Sequence Analysis and Change Report Analysis to identify logical dependencies. The Change Sequence Analysis lists the releases in which a module has been changed. Different modules can be compared on the basis of such change sequences and common change patterns can be identified.

Lanza and Ducasse in [35; 36] study the evolution of classes in a system using a combination of software visualization and software metrics. The visualization is 2D, with rows representing the classes in a system and columns denoting the version of the system. The first column would represent version 1 of the system, the second version 2 and so on. The number of methods in the class decides the width of each rectangle representing a class, while the number of
instance variables in the class decides the height of the rectangle. The authors suggest that other metrics can also be used effectively to represent a class. This metaphor allows easy visualization of the number of class in the system, the most recent classes that have been added to the system and growth and stagnation phases in the evolution. An innovative technique here is the classification of classes based on the kind of changes made to them over the different versions of the system.

Koike et al. in [61] present a 3D visualization framework (VRCS), by means of which a user can interact with a version control system. Versions of the files in a system are represented as cubes arranged along the z-axis, ordered by time. Releases that link versions of various files together are represented as circles. VRCS has been implemented using OpenGL/C and serves as an interface to RCS. Users can “check out,” edit and “check in” files, view differences between two cubes/versions of a file, retrieve all the files that compose a release and even build the executable file for a release. The authors also suggest some mechanism that enables the user to select the amount of graphical information presented. VRCS can only be applied to single-user systems.

Finally, CVSscan [120] is an integrated multiview environment that helps users to better understand the status, history and structure of the sources code, as well as for instance the roles played by various contributors.

4.2 Software security

One possible application of SV is in the area of software security analysis. For example, visualizing the results of dependency analysis and traceability analysis in a software system can help identify the potential security vulnerabilities if proposed changes to a system are implemented.

Conti and Abdulla in [21] discuss use of SV for security analysis. The authors examine the visual fingerprints left by a wide variety of popular network attack software tools to provide better understanding of the specific methodologies used by attackers as well as the identifiable characteristics of the tools themselves. The techniques used in the paper are entirely passive in nature making them virtually undetectable by the attackers. The paper explored the application of several visualization techniques including parallel coordinate plots and scrolling plots for their usefulness toward identification of attack tools, without the typical automated intrusion detection system’s signatures and statistical anomalies. These visualizations were tested using a wide range of popular network security tools and the results showed that in many cases, the specific tool can be identified.

While [21] focused on attack tools fingerprints, Yao in [122] studied virus fingerprints. The paper focused on visualizing Windows executable viruses using self-organizing maps (SOMs) without using virus specific signature information as a prior stage of detecting computer viruses. SOMs are visualized using the unified distance matrix. The paper addresses the fact that each virus has its own character to be distinguished although it is inserted in the executable file. Yoo observed that the virus features cannot be hidden through the SOM visualization; these features are like a DNA that determines a person’s unique genetic code. The research studied how virus codes effect the whole program projection. Without each virus signature, and presented how a virus pattern in Windows executable files indicate its family. The paper also shows that variants of a virus also can be covered with the specific virus’ mask, which is produced by SOM.

4.3 Data mining in software systems

Visualization is employed in data mining to visually present already discovered patterns and to discover new patterns visually. Success in both tasks depends on the ability of presenting abstract patterns as simple visual patterns [119].

SV is used in [15] for mining software archives. A software archive is comprised of the information stored by a configuration management system and related tools. This information includes versions, changes, bug databases and electronic mail. The authors claim that the relevance of many of the software engineering rules published in the literature to a project or a set of projects is unclear; either the rules are too general, or results of the case studies cannot be transferred, because the constraints of the case studies are not well documented. The authors use visual data mining for extracting rules from software archives for validation of the application of these rules and also for discovering new project-specific rules. The authors developed EPOSee to visualize n-ary association and sequence rules and study software evolution and relations based on hierarchically ordered items. EPOSee uses pixelmaps and parallel coordinates views, and provide visualizations that conform to Ben Sheiderman’s visualization mantra; “Overview first, zoom and filter, then details on demand” [97]. As an example, the paper studied the large software archive of the Mozilla open source project.

Vityaev and Kovalerchuk in [119] propose a technique called inverse visualization (IV) to address the problem of visualizing complex patterns. Their approach does not use data “as is” and does not follow a traditional sequence: discover pattern — visualize pattern. Instead, the sequence proposed in [119] is: convert data to visualizable form — discover patterns with predefined visualization. IV is based on specially designed data preprocessing which permits the discovery of abstract patterns that can be presented by simple visual patterns. In the paper, the feasibility of solving inverse visualization tasks is illustrated on functional non-linear additive dependencies that are transformed into simple and intuitive visual patterns.

4.4 Algorithms and software engineering education

Algorithm and software engineering visualization can help instructors to explain and learners to understand algorithms
and software engineering principles and practices [55]. For example, an algorithm can be animated showing relevant parameters and variables, the current state and a visual representation of the objects being manipulated, and also an animated formal description of the algorithm. Complex model structures are simplified at a high level of abstraction to highlight only the important aspects. Details can then be shown at lower levels of abstraction by omitting irrelevant details. For better comprehension, the designer scales down data to coarser structures and slows down algorithms that process data. Smooth transitions between different states of moving objects can make it easier to follow the way the algorithm works on graphical representations of data structures.

A recent proposal by Baloian et al. [7] concerns an approach to develop algorithm visualization that seeks to construct context-dependent interfaces allowing the learner to interactively control and explore the actions in the algorithm. The proposed approach replaces standard control using mouse clicks on graphic displays with a concept called Concept Keyboards (CK) mirroring the inherent logical structures of the algorithm under investigation. The CK concept separates control elements, data input and visual output objects by means of an adequate concept keyboard application to be used to configure keyboards, collect startup data and visualize user actions.

A key on a concept keyboard has a special meaning (concept) associated with it instead of just a label. Each key of the CK will be mapped to the execution of an existing method available in the algorithm implementation. In order to choose the interesting events (those which are crucial for understanding the algorithm), the designer has a simple GUI displaying the available actions and allowing them to select the relevant ones. CKs are used to trigger more complex semantic actions over the system in which they have been implemented. The special software supplied allows the user to redefine the function of each key and to regroup keys into fields of differing sizes. The user’s attempts at manipulation of algorithms and data structures are reflected by changes in the visualization or another form of output like textual or acoustic information. This provides users, including people with sensory disabilities, with suitable interfaces that may enhance the comprehension of the algorithm being presented.

The GRASP, and its successor jGRASP, where developed in [25; 50] with the goal to enhance software system comprehension efficiency and effectiveness. The developed visualization tools support well-defined cognitive processes employed during a comprehension task, such as top-down, bottom-up and mixed comprehension models. Grissom et al in [49] measured the effect of varying levels of student engagement with algorithm visualization to learn simple sorting algorithms. Their results showed that learning increases as the level of student engagement increases. They concluded that algorithm visualization has a bigger impact on learning when students go beyond merely viewing a visualization and are required to engage in additional activities structured around the visualization.

5 Conclusions

Advances in software visualization are leading to its pervasive adoption for better comprehension, engineering, and consequently enhancements in algorithm animation, software evolution and software metrics. Development of secure software and software engineering education are also major beneficiaries.

Interactive visualization can be coupled with other modalities, such as sensing or predictive methods, to provide powerful new capabilities for software visualization as well as other visualization domains. In addition, the fusion of visualization techniques with other areas such as data mining, grid computing, and web services are promoting broad-based advances, particularly in the emerging areas of visual analytics and mobile visualization. Another promising area of SV advancement is collaborative VEs that will lead to better understanding of collaborative software engineering processes.

Indeed, the importance of SV is growing, both in academia and industry [13]. A recent survey of software maintenance, re-engineering and reverse engineering researches [30; 29] shows that 40 % of researchers considers SV “absolutely necessary for their work” while 42 % of researchers considers it “important but not critical.” In addition, a significant increase in SV research is apparent in the plethora of recent conferences, workshops, and symposia on software visualization. For a wide spectrum of new ideas and approaches, the reader is referred to the Dagstuhl Seminar “Software Visualization” (2001), the ACM Symposium on SoftwareVisualization (2003, 2005) and the IEEE International Workshop on Visualizing Software for Understanding and Analysis (2002, 2003, 2005).

References

on Visualization (VIS’04), IEEE Computer Society, Washington, DC, USA, p 598.4
the visualization of software metrics. In: Proceedings of the 2005
ACM symposium on Software Visualization (SoftVis’05), ACM Press,
New York, NY, USA, pp 164–172.214
vey and analysis. In: Proceedings of the 9th International Work-
shop on Program Comprehension (IWPC 2001), Toronto, Ont.,
Canada, pp 7–17
change-proneness in OO software through visualization. In: Pro-
cedings of the 11th IEEE International Workshop on Program
Comprehension (IWPC 2003), Toronto, Ont., Canada, pp 414–421
12. Boisvert C (2004) Supporting program development comprehen-
sion by visualising iterative design. In: Proceedings of the Eighth
International Conference on Information Visualisation (IV’04),
pp 717–722
support in industry. In: Proceedings of the International Confer-
ence on Software Maintenance (ICSM’03), pp 348–357
on Software Visualization (SoftVis’05), ACM Press, New York,
NY, USA, pp 37–46
neering. Communications of the ACM 47(9):53–58
Interactive software Visualization Environment. In: Proceedings of
the 2004 IEEE Symposium on Visual Languages and Human
Centric Computing (VLC’04), pp 41–43
In: Proceedings of the Fifth International Workshop on Program
Comprehension (IWPC’97), pp 183–191
management data. IEEE Computer Graphics and Applications
18(4):24–29
to support the use of evolutionary algorithms. Journal of Visual
Languages and Computing 14(2):123–130
network attack tools. In: Proceedings of the 2004 ACM Work-
shop on Visualization and Data Mining for Computer Security
(VizSEC/DMS’04), pp 45–54
modeling and implementing visual notations with applications
to software engineering. ACM Transactions on Software Engi-
nering Methodology 13(4):431–487
symposium on Software Visualization (SoftVis’05), ACM Press,
New York, NY, USA, pp 135–144
able visualizations to support reverse engineering: A framework for
evaluation. In: Proceedings of the Fifth Working Conference on
Reverse Engineering, pp 201–209
ASEE/IEEE Frontiers in Education Conference, pp 12b1/5–
12b1/10
software visualization. In: Proceedings of the 23rd International
Workshop on Software Engineering (ICSE’01), pp 758–759
Proceedings of the 9th International Workshop on Program Com-
prehension (IWPC’01), pp 50–59
nar, Dagstuhl Castle, Germany, May 20–25, 2001, Revised Pa-
ers. Lecture Notes in Computer Science 2269, Springer, Berlin
27th international conference on Software engineering (ICSE’05), ACM Press, New York, NY, USA, pp 718–719
30. Diehl S, Kerren A (2002) Refinement of program points for vi-
sual execution. In: Proceedings of the First International Work-
shop on Visualizing Software for Understanding and Analysis
(VISSOFT’02), pp 100–109
Visual Languages, pp 364–371
32. Eick J, Sutinen E (2002) Software visualization — edito-
33. Dong J, Yang S, Zhang K (2005) VisDP: A web service for vi-
sualizing design patterns on demand. In: Proceedings of the In-
ternational Conference on Information Technology: Coding and
Computing (ITCC’05), vol 2, pp 385–391
cedings of the 11th IEEE International Symposium on Visual
Languages, pp 342–349
35. Ducasse S, Lanza M (2005) The class blueprint: Visually sup-
porting the understanding of classes. IEEE Transactions on Soft-
ware Engineering 31(1):75–90
Reengineering (CSMR’04), pp 309–318
layout. In: Proceedings of the Australian Symposium on Infor-
mation Visualisation 2001 (invis.au), Sydney, Australia, pp 77–
85
sualizing software changes. IEEE Transactions on Software En-
gineering 28(4):396–412
software. In: Proceedings of the First International Workshop on
Visualizing Software for Understanding and Analysis (VIS-
SOFT’02), pp 51–60
41. Feijs L, Jong RD (1998) 3D visualization of software architec-
tures. Communications of the ACM 41(12):73–78
42. Fiutem R, Merlo E, Antonio G, P’Tornella (1996) Understand-
ing the architecture of software systems. In: Proceedings of the
Fourth Workshop on Program Comprehension (WPC’06), pp
287–296
based on product release history. In: Proceedings of the Interna-
tional Conference on Software Maintenance (ICSM’98), IEEE
Computer Society, Washington, DC, USA, p 190
44. Gall H, Jazayeri M, Riva C (1999) Visualizing software release
histories: The use of color and third dimension. In: Proceedings of
the IEEE International Conference on Software Maintenance
(ICSM’99)
45. Gestwicki P, Jayaraman B (2005) Methodology and architecture of
JIVE. In: Proceedings of the 2005 ACM symposium on Soft-
ware Visualization (SoftVis’05), ACM Press, New York, NY,
USA, pp 95–104
ment and software visualization framework for heterogeneous
distributed applications. In: Proceedings of the 6th IEEE Inter-
national Symposium on High Assurance Systems Engineering
(HASE’01), pp 106–116
47. Girba T, Lanza M, Ducasse S (2005) Characterizing the evo-
lution of class hierarchies. In: Proceedings of the Ninth Euro-
pean Conference on Software Maintenance and Reengineering