Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 29 (2005) 311–329
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

3D visualization techniques to support slicing-based program comprehension

Juergen Rilling, S.P. Mudur*

Department of Computer Science and Software Engineering, Concordia University, 1455, de Maisonneuve West, Montreal, Que, Canada H3G 1M8

Abstract

Graphic visuals derived from reverse engineered source code have long been recognized for their impact on improving the comprehensibility of structural and behavioral aspects of large software systems and their source code. A number of visualization techniques, primarily graph-based, do not scale. Some other proposed techniques based on 3D metaphors tend to obscure important structural relationships in the program. Multiple views displayed in overlapping windows are suggested as a possible solution, which more often than not results in problems of information overload and cognitive discontinuity. In this paper, we first present a comprehensive survey of related work in program comprehension and software visualization, and follow it up with a detailed description of our research which uses program slicing for deriving program structure-based attributes and 3D-metaball-based rendering techniques to help visualization-based analysis of source code structure. Metaballs, a 3D modeling technique, has already found extensive use for representing complex organic shapes and structural relationships in biology and chemistry. We have developed a metaball software visualization system in Java3D, named MetaViz. As proof of concept, using MetaViz, we demonstrate the creation of 3D visuals that are intuitively comprehensible and communicate information about relative component complexity and coupling among components and therefore enhance comprehension of the program structure.

Keywords: Software visualization; Program slicing; 3D modeling; Metaballs; Visual mapping

1. Introduction

Throughout a software product's life cycle, many different people are responsible for understanding the design details of the software code. Learning the structure of code developed by others is especially time consuming and effort intensive during the software maintenance phase. Maintenance programmers are often not involved in the original design and implementation and therefore must necessarily rely on design and testing documents to comprehend the design and the related code. However, after several maintenance cycles these documents are out of sync and often obsolete. Clearly, software maintenance, reengineering, and reuse involving large software systems are complex, costly, and risky tasks, as a direct consequence of the difficult and time-consuming task of program comprehension. Many reverse engineering tools have been built to help comprehension of large software systems. Software visualization is one approach suggested and being investigated worldwide for providing some assistance in program understanding. It should be recognized that

*Corresponding author.
E-mail addresses: rilling@cs.concordia.ca (J. Rilling), mudur@cs.concordia.ca (S.P. Mudur).

0097-8493/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2005.03.007

visualization is a complementary technique and is to be used in conjunction with other program understanding techniques such as software inspection, metrics, static and dynamic source code analysis.

1.1. Software comprehension

The increasing size and complexity of software systems introduces new challenges in comprehending overall program structure, their artifacts, and the behavioral relationships among these artifacts. Numerous theories have been formulated and empirical studies conducted to explain and document the problem-solving behavior of software engineers engaged in program comprehension [1–8]. The *bottom-up* approach reconstructs a high level of abstraction that can be derived through reverse engineering of source code. The *top-down* approach applies a goal-oriented method by utilizing domain/application specific knowledge to identify parts of the program that are necessary for identifying the relevant source code artifacts. In [4,7] an *opportunistic* approach is described that exploits both top-down and bottom-up cognitive approach.

However, for comprehension of large systems, it is not only impractical to attempt comprehension of the complete system, it might also be unnecessary. Often it suffices to obtain only some partial understanding related to the particular aspect of interest and to build a mental model one can rely on when performing a particular maintenance activity, or to create a model for locating places where such a change should be applied. Typically, a program performs a large set of functions/outputs. Rather than trying to comprehend all of a program's functionality, programmers will focus on selected functions (outputs) with the goal of identifying parts of the program that significantly influence a particular function or functions. One approach is to apply program slicing techniques to reduce the displayed data by including only those software entities (files, modules, classes, functions, statements and objects) that are relevant with respect to the computation of a specific program function of interest. Furthermore, the slicing approach can be applied in combination with software metrics to analyze and evaluate the quality of software systems based on their comprehensibility and maintainability.

1.2. Software metrics and program slicing for software comprehension

For a number of years now, software metrics have been used in the industry [1,9,10] to define, measure and analyze software quality. Large amounts of data are collected according to predefined analysis and quality models, and "analyzed" to find patterns showing design and code anomalies, etc. Extensive aggregation and filtering of data has to be performed before meaningful trends can be observed. The results of this analysis are traditionally presented in the form of textual tables and simple graphs.

Program slicing is a well-known decomposition technique that transforms a large program into a smaller one that contains only statements relevant to the computation of a selected program function (output). Weiser [8] defined a slice S as a reduced, executable program obtained from a program P by removing statements such that S replicates parts of the behavior of P. Being able to apply slicing in combination with metrics and synthesize partial views allows one to focus on computation of particular metrics for only those parts of the software that are of immediate interest. This in turn can therefore significantly enhance the usability of these metrics.

As in other disciplines of scientific visualization [11], it makes sense to combine these metrics with more sophisticated visualization techniques to assist the user in comprehending the data and, consequently, to find patterns/relationships that are not as obvious. For example, the cognitive complexity of an object-oriented system is of specific interest to all software engineers. A good deal of this complexity is reflected through the collaboration of classes in the system. Metrics can be applied to measure this complexity of systems and provide details about collaboration. Therefore a more refined view of the structure of the software system can be obtained. Using these metrics, relationships between software artifacts and also those between their components can be captured and then graphically represented.

1.3. Visualization of software structure and software metrics

Humans are limited in the density of textual information they can resolve and comprehend [7,12–16]. It is well established that a good visual representation will often serve as an excellent comprehension aid and facilitate study of complex problems in parallel [17]. A number of software visualization techniques have therefore evolved. Section 3 provides a review of published techniques.

Visualization of the internal structure of software systems could be used for different purposes, but primarily, it is to support program *comprehension*. The goal is to acquire sufficient knowledge about a software system by identifying program artifacts and understanding their relationships. As programs become more complex and larger, the sheer volume of information to be comprehended by the developers becomes daunting. It would be ideal to be able to simultaneously view and understand detailed information about a specific activity in a global context at all times for any size of program.

As Ben Shneiderman explains [18,19], the main goal of every visualization technique is "Overview first, zoom and filter, then details on demand". This means that visualization should first provide an overview then let the user restrict the source code on which the visualization is applied, and then create views that provide more details on the part of interest to the user.

Source code views, primarily based on some diagrammatic notation, have been evolving from the early days of computing [16,20,21]. However, for large, complex software systems, comprehension of such diagrammatic depictions is restricted by the resolution limits [22] of the visual medium (2D computer screen) and the limits of user's cognitive and perceptual capacities. One approach to overcome or reduce the limitations of the visual medium is to make use of a third dimension by mapping source code structures and program executions to a 3D space. Mapping these program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code. In this paper, we focus on the use of metaballs (also referred to as metablobs, soft objects or more generally implicit surfaces), a 3D modeling technique that has found extensive use in representing and visualizing complex organic shapes and structural relationships such as the DNA, humans, animals and other molecular surfaces [1,23–27] (cf. Fig. 1).

Earlier in [28], we proposed to extend the application domain of metaballs to include the visualization and comprehension of very large program artifacts. In this paper our emphasis is on demonstrating its potential and effectiveness in combination with program slicing-based metrics in creating 3D visuals for comprehending program structure. We demonstrate this using the Java 3D program we have developed for the same purpose. The extent of applicability of the metaball modeling and visualization technique in other domains has been such that virtually every significant commercially available 3D modeling software incorporates metaball modeling and rendering in some fashion or the other. Correspondingly, there are a large number of free software sites for packages supporting this technology [29]. However, to the best of our knowledge, ours is the first such attempt to apply the metaball metaphor in metrics-based software visualization.

1.4. Metaball metaphor

The metaball metaphor is a 3D object modeling and rendering technique which blends and transforms an assembly of particles with associated shapes into a more complex 3D shape, whose use is highly suitable for animal and other organic forms. This technique models particles in 3D space, which have energy (strength) and have a well defined, parametrically controlled influence over the surrounding and neighboring particles.

A metaball is defined by a so-called 3D variable density field, radiating from a given center point. The value of the field can vary linearly with distance from the center, or in any other way expressible via a mathematical formula. For example, a field can have a negative density distribution, or even an eccentric distribution. A metaball surface is constructed as the set of all points in the field with the same density value, which is given by the modeler or derived from the modeling context.

If two or more metaballs are constructed in close proximity to one another so that they overlap, they coalesce and their fields are added in a process called fusion to produce a composite field, which is then evaluated to produce a composite surface. Metaball fields can be transformed in a variety of ways to produce organic shapes necessary to represent, for example, the human form. Metaball surfaces are usually rendered as polygons. Metaballs have found extensive use in representing and visualizing complex organic shapes and structural relationships such as DNA, humans, animals, and other molecular surfaces. Extensions include grouping of particles, selective influence over other particles, hiding particles, etc.

By defining visually intuitive mappings between the entities or parameters in the software slices and metaball models, we can create a 3D virtual environment in which it is possible to walk around these entities, to see what significantly influences the entity of interest, or to hide insignificant influences or zoom into entity-groups for understanding more detailed interactions. By mapping different entities in multiple views, say object or functional, it becomes possible to use the same 3D metaphor to help understand software from different viewpoints. Mapping entity type to shape gives us the potential to visually differentiate, for example, free functions from member functions in a C++ program. Interacting with a complex metaball model, by moving an entity of interest closer to clusters of other entities and observing the animated response, could further help in visualizing the more dynamic aspects of a large software program. In short, the metaball metaphor gives us a constantly moving micro-universe of entities (metaballs), which can be dynamically altered to model program parameters and can be interactively walked through for various reverse engineering purposes, such as design evaluation, maintenance and testing.

1.5. Organization of this paper

The main focus of this paper is to show how 3D modeling and rendering techniques can be applied to assist in the task of program comprehension by creating suitable 3D visual representations of the software and its

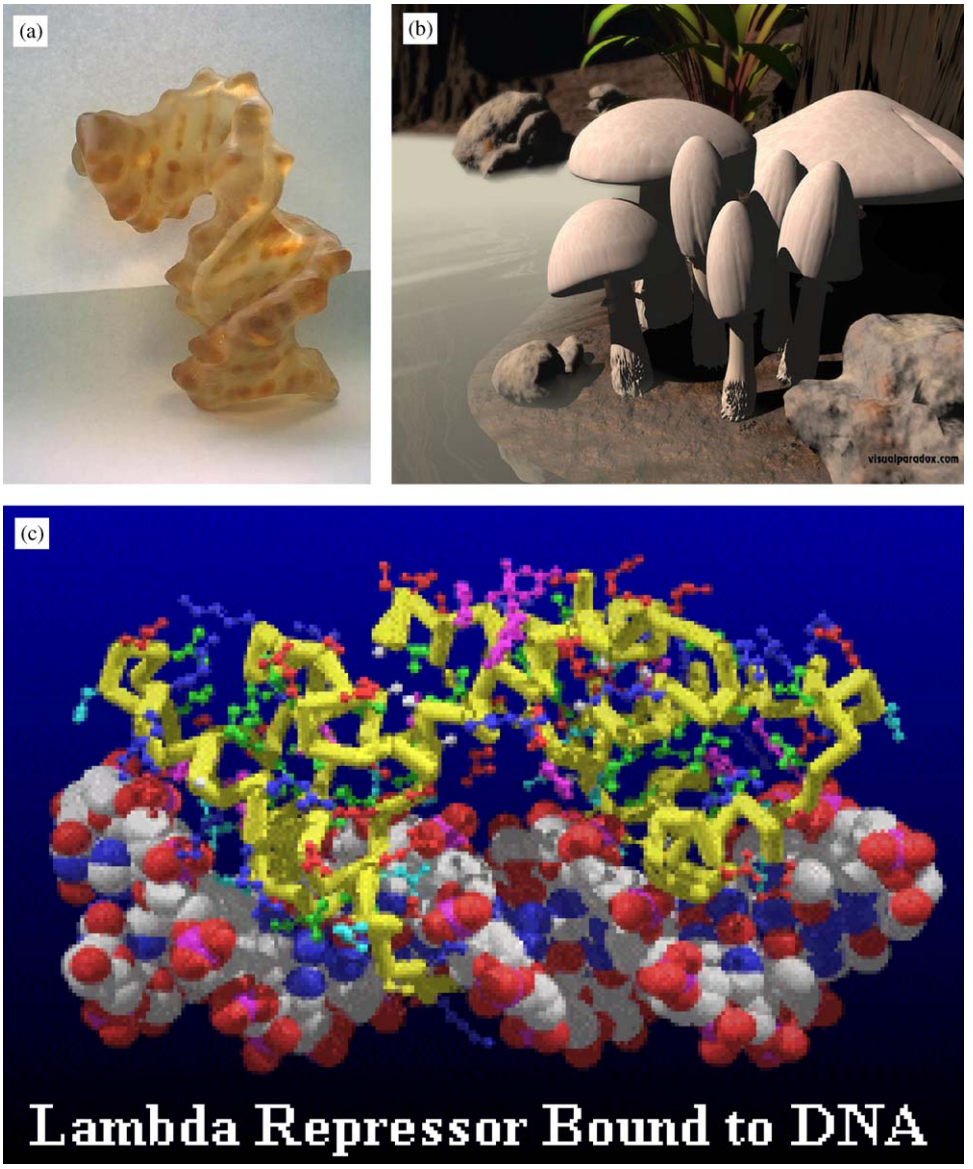

Fig. 1. Traditional applications of meatballs: (a) DNA structure (www.scripps.edu/pub/olson-web); (b) organic visual (www.visualparadox.com); (c) molecular images (The Scripps Research Institute).

internal structure. Specifically, we combine metrics derived from program slicing techniques with metaball representation to visually depict various aspects of software, particularly relative sizes of software entities, interrelationships, program slice membership, etc. For this we define intuitive visual mappings for different types of entities/metrics. The remainder of this paper is organized as follows: Section 2 reviews program slicing and metrics and their role in program comprehension. Section 3 discusses the current state of the art in software visualization, while Section 4 discusses application of metaballs in combination with program slicing for typical software comprehension tasks and also intuitive visual mappings for various aspects of large software. In Section 5, we demonstrate the applicability and effectiveness of this technique by creating metaball visuals of a Java 3D program called as MetaViz that has been developed by us for software visualization purposes. Lastly, in Section 6, we present our conclusions based on our experience with the metaball metaphor in visualizing program structures and discuss future planned extensions to this work.

2. Software metrics and program slicing—a survey

2.1. Software metrics

The term software metrics is not uniquely defined. In literature, software measure, software measurement and software metrics are often used interchangeably. IEEE does provide the following definition: *The application of a systematic*, *disciplined*, *quantifiable approach to the development*, *operation*, *and maintenance of software*; *that is*, *the application of engineering to software*. While the use of metrics is commonplace in the traditional engineering world, it has yet to become mainstream in the software domain, even though it has been shown that software metrics can provide software engineers and maintainers with guidance in analyzing the quality of their design and code and its future maintainability [1,9,30–34]. Software metrics could address many aspects of the software life cycle—process, product, people, quality, design maintenance etc. Li and Henry [32] concluded after using their metrics to evaluate two software systems that there "is a strong relationship between metrics and maintenance effort in object-oriented systems" and that "Maintenance effort can be predicted from combinations of metrics collected from source code". The remainder of this section presents the design metrics that we used for visualizing system structures using our metaball visualization.

2.2. Software design metrics

Several design metrics are presented in the literature [31,32,34,35] to evaluate the design and quality of software systems, enabling the early identification of maintenance and reuse issues in existing systems. Two major categories of design metrics can be distinguished: cohesion (internal aspects) and coupling (structural aspects) design metrics. Within this paper, we focus on coupling metrics that are traditionally used to evaluate the structural quality of software systems design.

2.3. Coupling measurements

Coupling models the relationship and interaction between modules or classes and is based traditionally on the information flow. Besides cohesion, coupling is one of the most significant quality attributes of every software application or module design. Coupling, as defined by Briand [36], is "the measure of the strength of association established by a connection of one module to another." Strong coupling makes a system complex, since a module is harder to understand, change, or correct by itself if it is highly interrelated to other modules. Coupling is studied simultaneously with cohesion in structured programming and more recently in object-oriented programming [31,36,37]. Ideally, interacting objects should be as loosely coupled to one another as possible for the following reasons: (1) fewer interconnections between modules reduce the chance that a fault in one module will cause a failure in other modules, (2) fewer interconnections between modules reduce the chance that changes in one module cause problems in other modules and (3) fewer interconnections between modules reduce programmer time to understand the details of other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.

A large number of metrics are introduced and discussed in object-oriented programming. The most widely used and evaluated metric suites are from Chidamber [37] and Li [32]. For example, *coupling between objects* (*CBO*), introduced by Chidamber and Kemerer [37], for a class corresponds to the count of non-inheritance related couples with other classes. Another way of expressing CBO is that an object is coupled to another object if two objects act upon each other, i.e., methods of one use methods or instance variables of another. In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. Larger the number of couples, higher is the sensitivity to changes in other parts of the design. As a result maintenance and testing are more difficult.

Similarly, *Response for a Class* [37] measures communication among classes through message passing. A message can cause an object to "behave" in a particular manner by invoking a particular method. The response set is a set of methods available to the object and its cardinality is a measure of the attributes of an object. Yet another, *message passing coupling* (*MPC*) was introduced by Li and Henry [33] as part of their extension to the original metrics suite presented by Chidamber and Kemerer. MPC measures coupling between classes by message passing and is calculated at the class level. MPC is the static number of send statements defined in a class, where a send statement is a message sent out from a method of class a to class b. Program slicing techniques can help in obtaining these metrics from the source code of programs.

2.4. Program slicing

Weiser's original research [8] on program slicing was motivated by the need to help students understand and debug their programs. Weiser discovered that the mental process used by programmers when debugging their code was slicing and tried to formally define this process and the output by introducing his first algorithm.

Weiser [8] defined a static program slice as those parts of a program P that potentially could affect the value of

a variable v at a point of interest. A large number of extensions to the original slicing algorithms have been presented in the literature (e.g. [38–44]).

Korel and Laski [45] introduced the notion of dynamic slicing that can be seen as a refinement of the static approach by utilizing additional information derived from program executions on some specific program input. The dynamic slice preserves the program behavior for a *specific* input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which a program terminates. Later extensions of the dynamic slicing algorithms to include hybrid algorithms [46–49] that combine static and dynamic information for the slice computation were introduced in [50,21].

The reason for this diversity of slicing techniques and criteria is that different applications require different properties of slices. Program slicing is not only used in software debugging [8,39,48,50–53], but also in software maintenance and testing [5,34,43,53]. There are two major approaches to locating places of low design quality in an existing design. The first one is a systematic approach that requires a complete understanding of the program behavior before any code modification. The second one is an "as needed" approach that can be adopted as it requires only a partial understanding of the program to locate as quickly as possible certain code segments that need to be changed, tested, or maintained for desired enhancement or bug-fixing.

2.5. Slicing-based coupling measures

The usefulness of traditional coupling measurements like CBO, RFC and MPC has already been shown in a number of empirical studies [1,9,30,31,36,47]. Some of the application areas of these metrics are in testing and maintenance activities for object-oriented systems. The results of these empirical studies show that there is a strong relationship between metrics and maintenance effort in object-oriented systems and that maintenance effort can be predicted from combinations of metrics collected from source code.

As stated earlier, one might not always want to analyze the whole software design and quality of the system, but rather focus on the analysis of a particular variable, function or feature. Traditional software design metrics like, CBO, RFC, and MPC can be combined with program slicing to capture design issues related to a particular slicing criterion. These measurements will therefore consider only those software artifacts in the measurements that are included in the slice. It should be noted that the slicing criterion might not be restricted to the traditional variable/statement level; it can also be applied for different levels of abstractions, e.g. method, class or feature level [32] and therefore provide different levels of granularity. The provision of different levels of granularity allows the user to select the level of abstraction that corresponds closest to the mental models that one forms during a particular comprehension or maintenance task [4]. Using slicing we can obtain not only a more focused, but also a more precise measure of design quality with respect to a particular slicing criterion. It also allows for further detailed analysis of the level of information flow among modules compared to traditional coupling metrics. A detailed discussion of the applicability and use of different slicing-based coupling and cohesion measurements can be found in [34,43,54–56]. They identified testing, comprehension and re-engineering as some of the potential application areas for program slicing-based measurements.

2.6. Slicing-based message passing coupling (SMPC)

In this paper, we focus on software visualization using slicing-based MPC [32,57] metrics. We use these to illustrate the applicability of the combination of 3D visualization and program slicing to refine existing software measurements. MPC measures the number of methods that can be invoked from a class through messages. MPC among classes is calculated at the class level. MPC is a static measurement of send statements defined in a module (class), where a send statement is a message sent out from a method of module M_1 to module M_2. An object/function call is included in the slice if at least one statement within this program artifact is included in a program slice. Similarly, a call relationship connecting line between two modules M_1 and M_2 (where M_1 calls M_2) is included in the slice if at least one "call M_2" statement inside of module M_1 belongs to the program slice. Larger the number of invoked messages (MPC), greater is the complexity of the class and correspondingly greater is the testing and debugging effort required to comprehend module interactions. This method often results in a large number of static couplings (method calls) that leads to high MPC measures, making meaningful analysis difficult.

Slicing-based MPC metric not only reduces complexity of the metric itself, it also correlates the value of the metric to the context of a particular program slice. Coupling measurements can also be used to group and organize software systems in loosely and highly coupled subsystems. Clearly, slicing-based coupling metrics can play a significant role in identifying coupling complexity of software systems at different levels of granularity [57]. In combination with suitable visualization techniques they provide a powerful mechanism to assist in program comprehension on an "as needed" basis, and are therefore an essential approach for program comprehension. Section 5 illustrates the use of program slicing-based MPC metrics-based visualization.

3. Software visualization

3.1. Limitations of 2D graph-based visualization

Visualization in the form of reverse engineered 2D diagrams (e.g., collaboration diagrams, call-graphs, sequence diagrams, etc.) and models (UML class models) have been suggested in the literature [12] to provide users with higher abstraction views of the software under investigation (cf. Fig. 2). The principal aim is to ease program comprehension by enabling a person to comprehend complex internal structure and entity relationships through appropriate visual mappings. For large software systems, however, it becomes increasingly difficult to comprehend these diagrams for several reasons:

(1) The visualization technique does not scale up causing increased clutter in the diagram because of the large amount of information (entities and entity relationships) to be displayed, essentially resulting in information overload problems. Often, 2D inheritance trees or call graphs of several thousands of entities create space filling and incomprehensible visuals.
(2) Awkward layout techniques provided by the chosen visualization mapping tend to obscure important patterns and relationships in the software from the user.
(3) Navigation tools are non-intuitive; pan-zoom and overlapping multiple windows are typically the kinds of navigation tools supported causing cognitive discontinuity problems. There are some visualization mappings such as fisheye-views [10], perspective information walls [58] and hyperbolic trees [59], which offer some solutions for focus versus detail. They assist the user in not getting lost in the visual space, but even these do not easily scale to very large software.
(4) Often their scope is rather specialized to depict only certain program artifacts and their relationships.

3.2. 3D versus 2D visualization

As mentioned earlier, over the last decade, programs have become larger and more complex, creating new challenges to the programmer in visualizing these complex and large source code structures. Providing a few predefined views might not be sufficient as users are still dealing with a large amount of information and data. Also, not every visualization technique is equally adept in displaying a particular aspect of software structure [60]. The visualization technique might lack an appropriate navigation support or may not allow effective reduction of the amount of information displayed through a choice of predefined views. The

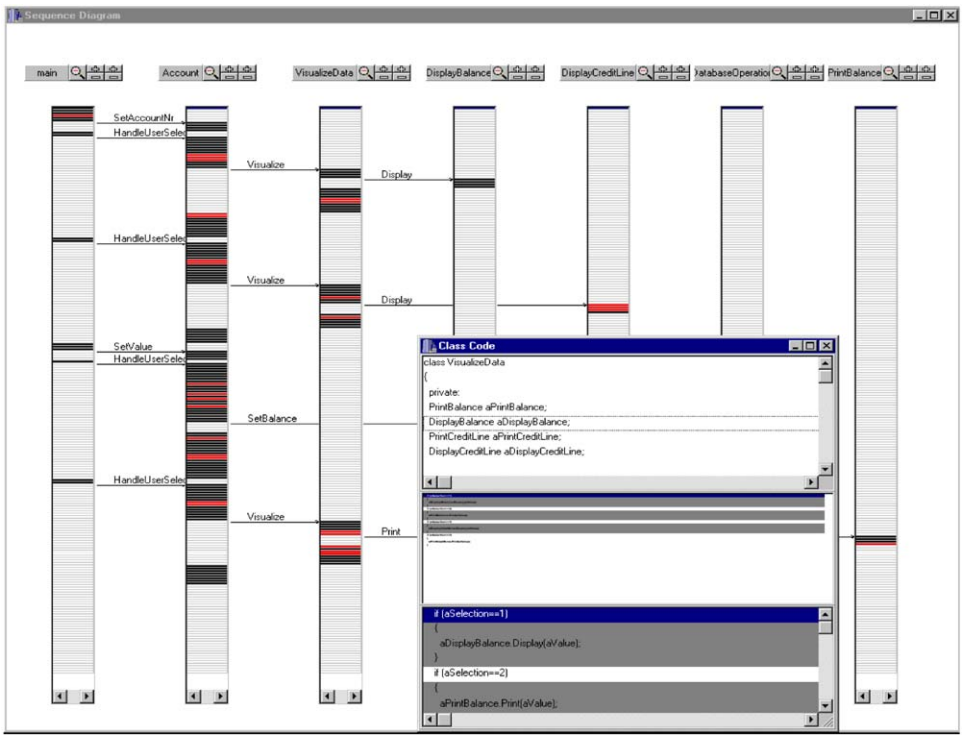

Fig. 2. Sequence diagram to visualize program executions—an example of a graph-based visualization (Screen capture).

disadvantages of most of the commonly used high-level abstractions such as call-graph, UML class models, and collaboration diagrams. Have been discussed by other authors as well [52].

Software visualization of source code structures and execution behaviors could consist of both static views and dynamic views [61,18,62]. Dynamic views are based on information from the analysis of recorded or monitored program executions. During recording of a program execution, a large amount of data may be collected. Although this is not a new problem, rapid increases in the quantity of information available and a growing need for more highly optimized solutions have both added to the pressure to make good and effective use of this information [63].

3D visual representations are often suggested and presented as a solution to provide just this required extra space and resulting ease of use in navigation and abstraction level. While the advantages of adding a third dimension are initially obvious, these are realizable only if truly distinct and effective use is made of the added dimension. However, most of the current approaches are just transforming established 2D visualizing techniques into a 3D space [16]. 3D software structure visualizations are still centered on creating standard call-graphs within a 3D space. For example, the usage of 3D call-graphs does offer a greater working volume for the graphs thereby increasing the capacity for readability. However, at the same time, they introduce undesirable effects that significantly affect the gain from the added dimension. Problems that might be introduced by 3D visualization techniques include significant objects being obscured, disorientation, and spatial complexity. To some limited extent, these issues can be resolved by 3D interaction techniques where the viewpoint of the 3D graph is actually within the graph structure; otherwise, the 3D visualization is limited to merely a 2D picture of a 3D structure (cf. Fig. 3).

3.3. 3D metaphors used in software visualization

Ultimately, for 3D visualizations to be effective, techniques other than just mapping 2D models onto the 3D space are required. These techniques must introduce a more meaningful and abstract program representation that makes full use of the 3D

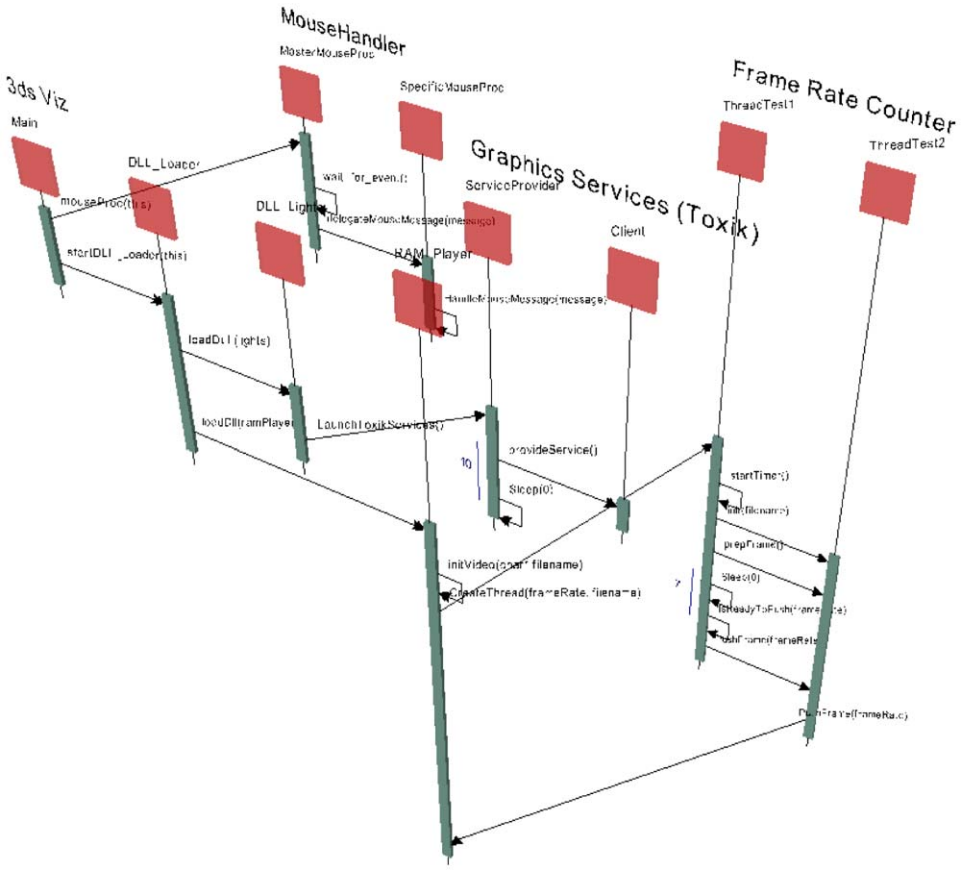

Fig. 3. Mapping 2D sequence diagram notation into 3D space (Screen capture).

environment and, thus, the engineer's natural intuition and perceptual skills [4,7]. 3D visualization techniques for software are all based on the use of a suitable metaphor which helps a user to mentally associate a visual/spatial aspect to a corresponding aspect of the software. For example, size or height/width of a visual object may be directly related to the inherent complexity of the software entity it represents; spatial proximity may be related to a measure of cohesion, etc. A number of 3D metaphors have been proposed. Each has its own specific comprehension objectives and its own advantages/disadvantages. These metaphors include the following:

- cone-trees [64], ideally suited for inheritance/inclusion kind of relationships;
- world cities [20,65], again well suited for object-oriented package/class inheritance relationships;
- human agents [66], primarily introduced for visualizing tasks in an operating system like software;
- Immersive VR [63] emphasizes the use of a virtual environment in which 3D representations of software are viewed in computer supported collaborative environment. Each user can obtain an independent view of the same software.
- 3D graphs [67], entities are represented using basic shapes like spheres connected by 3D lines and the metaphor of geometric distance is used to enable recognition of structures/patterns, etc.
- Metaballs [28], introduced by the authors using the metaphor of particle with attributes and energy influencing its surrounding particles.

3.4. Related work in metrics-based software visualization

Demeyer et al. have presented in their paper [68] a hybrid approach combining metrics and visualization. They combine software metrics with traditional graph-based visualization techniques, by mapping metrics values to color or size of nodes. A major limitation of their approach is that it utilizes different types of 2D graphs as visuals and would entail context switching. Further, scaling to large software would pose the type of problems discussed earlier. Systä et al. [69] present their research on applying OO metrics with graphical reverse engineering tools [69,70].

Their work focused on creating new visualization scripts for Rigi and to perform a case study on their subject Java system. Again, their visualization techniques were limited to more traditional 2D visualization approaches. Lewerentz and Simon [45] presented a 3D metrics-based visualization of OO programs. In their research, they focus on mapping program structures into a 3D graphs with nodes of different shapes. The primary metaphor is that of geometric distance. Common to all existing work in visualization of software metrics is the attempt to visualize the overall structure of the software system under investigation. So far, most are based on traditional graph-based visualization techniques, providing programmers with large quantities of information to be analyzed and interpreted. Often, these visualization techniques do not support an intuitive simultaneous mapping between the visual components and metrics, complicating the analysis of the results even further.

The next section discusses further the metaball metaphor for visualizing software structure with intuitive visual mappings for software artifacts/aspects, abstraction and navigation techniques.

4. Application of metaball metaphor to software visualization

Comprehension of OO programs is simplified if the relationships that exist between classes and other parts of the program are easily understood. Diagrammatic notations that have evolved certainly help. Thus, UML-based static and dynamic visualization techniques such as class models, sequence and collaboration diagrams can be applied for smaller software systems to provide an overview of the relationships in a program. However, for large software systems these diagrams do not provide adequate abstraction to visualize all the dependencies.

Particles in the metaball metaphor can be mapped to software structures, with spherical or elliptical blobs representing an object or a function (distinguished by different shapes for particles) that are created dynamically during a program execution and cylindrical blobs connecting these software entities for representing the interrelationship amongst them. The potential energy surrounding a blob has traditionally been used to indicate the influence the blob has with respect to its surroundings. This can be very intuitively used to visualize the strength of the coupling among program artifacts. For example, the number of function invocations performed among objects could be one of the parameters used to indicate the relationship (coupling) among the blobs (cf. Fig. 4).

The dimensions or size of a blob can be used to indicate a desired measure of the software entity, for example, the number of statements in a function. Blobs will be spatially located in clusters with spatial nearness indicating an identifiable association between the software entities that are mapped. We expect that a programmer preferred spatial configuration of entities is maintained in a persistent manner. This will enable the programmer to retain the visual association with software entities with very little effort.

One of the problems faced in most graph-based visual representations is that association with source text, say class/method/property names are difficult to maintain, in the graph-based visual format. The metaball surface

can be easily texture mapped with a label to enable this association with source code.

Fig. 5 shows a UML class diagram and a corresponding 3D metaball visual. For this example, the number of classes is small, and both visualizations appear to serve the intended purpose. But it is obvious that with limits on screen space the UML diagram does not scale. On the other hand, with the ability to navigate in 3D space and view the 3D structure from any viewpoint, the available space for pictorially depicting entities and relationships is vastly expanded. Further, combinations in which UML diagrams of metaball sub-clusters can be shown on demand would further enhance information communication.

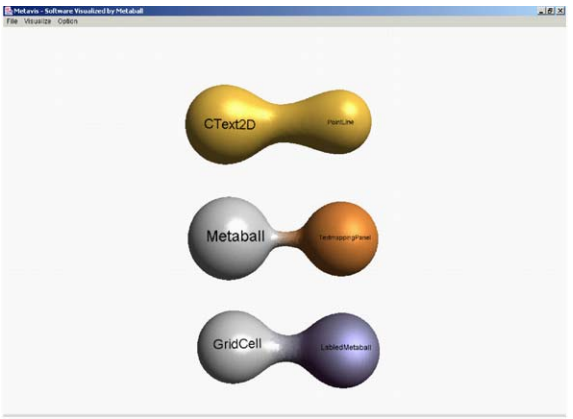

Fig. 4. Metaballs in visualizing software interaction. Top: shows high coupling between two entities, middle: shows some coupling between two entities and bottom: shows medium coupling between two entities.

Frequently, it is advantageous to reduce the number of visible elements. Limiting the number of visual elements to be displayed improves the clarity and simultaneously increases performance of layout and rendering [66]. Various ''abstraction'' and ''reduction'' techniques have been applied by researchers in order to reduce the visual complexity of graph-like structures. One approach is to perform clustering.

Clustering can be described as the process of discovering groupings or classes of data, based on chosen semantics. Clustering techniques have been referred to in the literature as *cluster analysis*, *grouping*, *clumping*, *classification*, *and unsupervised pattern recognition* [66]. Engineering illustrations have traditionally used the technique of explosion/implosion to depict/coalesce the internal details of complex engineering artifacts. For a visualization technique to be scalable to represent large amounts of data, the metaphor has to support similar techniques such as collapsing, hiding, and expanding parts of the diagram, thereby giving the user the ability to select the view granularity and consequently the amount of information that has to be displayed. Similar techniques are easily supported by metaball models.

Collapsing/expanding particle clusters enables us to visualize abstractions at different hierarchical levels, mainly to provide users with an option to summarize and analyze a program structure or a program execution. Such clustering techniques are less applicable in visualizing dynamic changes (particularly for large

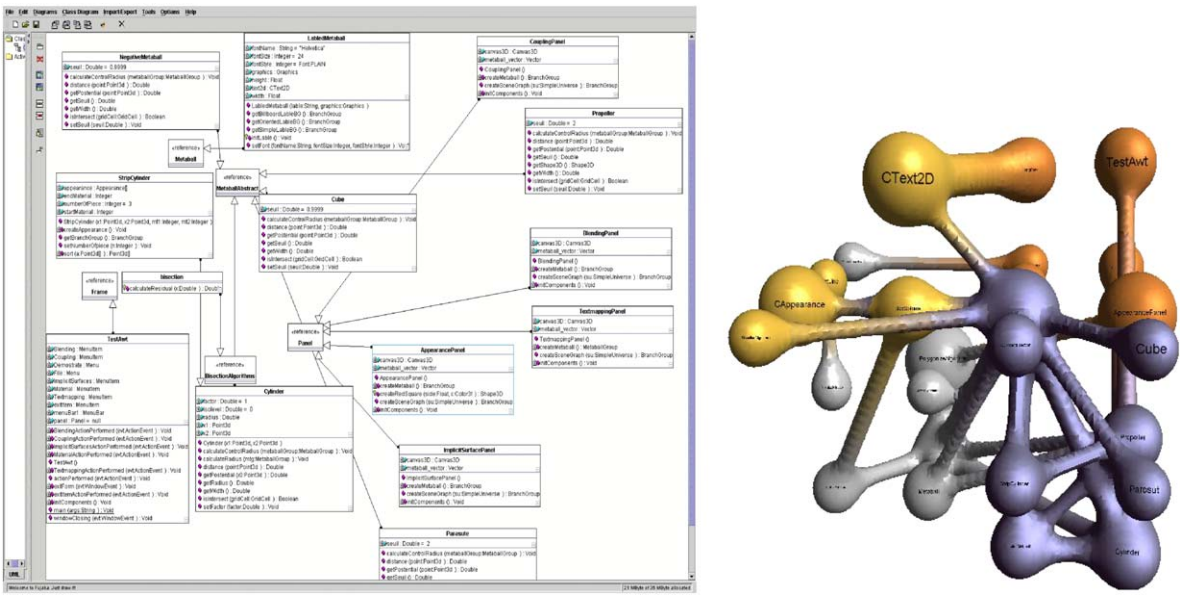

Fig. 5. Combination of classical UML diagram with metaball visualization (Screen capture).

Table 1
Visual mappings

Program artifact	Metaball property
Software entities	Particles
Entity types	Blob shape
Entity measure (e.g. no. of statements in function)	Blob dimensions
Entity association	Particle clustering
Entity relationship (e.g. hierarchy, coupling)	Cylindrical blob connecting two entity blobs
Relationship strength	Energy potential amongst entity blobs and connection blobs
Hierarchic levels of abstraction	Particle collapsing/expansion
Different dynamic behavioral aspects	Blob colors, brightness, shininess, animated change in connections, etc.

software systems) because of navigation or orientation issues. Grouping, and therefore, changing the layout dynamically would distort a user's ability to correlate the clusters with a particular program structure or content. Dynamically changing associations are visually depicted by animating changes in appearance properties of blobs/connections. Similarly, hiding or dimming blobs that are insignificant or less significant to a particular comprehension task gives us the ability to present only details that matter. Transparency and bounding spheres may be used to depict encapsulation. Table 1 shows a partial list of visual mappings that are supported by MetaViz, the Java 3D software that we have developed. This program implements the metaball metaphor for software visualization.

One basic requirement to enhance acceptance of visualization techniques is to provide programmers with navigation tools that provide easy navigation and a clear perspective about how the current program part relates to the overall structure of the software system. Browsing through the source code is akin to navigating in this 3D metaball space. Essentially one is provided full 3D virtual space navigation capabilities, including collision with software entities. Hence, the user can bring into focus any desired software entity/cluster/relationship. Additionally it is quite straightforward to provide rapid moves to entities far away from the current view, more like non-linear browsing of source code. For example, a click on a particular metaball could bring that metaball into focus, or the user could simply select the entity by name. Consider another scenario. Let us assume that a user would like to look at all the large software entities. This could be done by automatically moving the view from one large software entity to another.

4.1. Content-based clustering

The use of the semantic data associated with metaballs to perform clustering could be termed *content-based* clustering. Content-based clustering can yield groupings that are most appropriate for a particular application and can even be combined with structure-based clustering. Content-based clustering requires application of specific data and knowledge. It is important to note that clustering can be used for functions such as filtering and search. In visualization terms, filtering usually refers to the de-emphasis or removal of elements from the view, while searching usually refers to the emphasis of an element or group of elements. Both filtering and search can be accomplished by partitioning elements into two or more groups, and then emphasizing one of the groups by graphics techniques such as highlighting. The reader is referred to an example of this later in Fig. 15 which illustrates visual aids in clustering, utilizing program slicing.

4.2. Entity layout

Clustering data can also be used to arrive at the final layout of the entities in the geometric space. The layout algorithm should avoid cluttering of the metaballs. At the same time, it should use the space optimally so that appropriate detail is presented in all views. The general 3D layout problem is complex and layout for software visualization has been studied to a limited extent by a number of researchers [71,72]. Our MetaViz tool consists of three major parts: the grid-layout, a clustering and grouping algorithm and the metaball rendering engine (see Fig. 6). The grid-layout part uses an XML file as an input, which describes the software artifacts and their internal relationships. It is based on a traditional hill climbing optimizing approach. MetaViz provides the user with continuous feedback about progress made in the layout optimization by displaying snapshots of the current layout (cf. Fig. 7) in the form of a sphere-line 3D graph. Once a certain optimum is reached, the layout is complete. The resulting layout will be saved in an XML file to be rendered using the metaball technique.

The rendering engine part of MetaViz reads the XML file as input to generate and render the metaballs in 3D space by mapping the properties of software to the

properties of metaballs. After completion of the rendering process, the user can navigate through the visuals and apply an overview, select and zoom approach to refine the current view.

Since one of the major goals of software visualization is to guide people during comprehension of software systems, we use readability as the major criteria to evaluate the quality of our layout algorithm. Similar readability criteria as already applied in information visualization can be applied for software visualization.

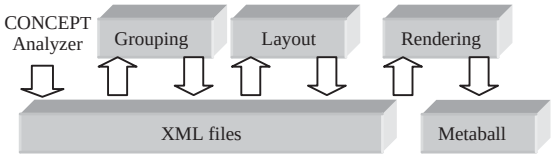

Fig. 6. MetaViz pipeline.

Fig. 8 shows the importance of the different criteria for the readability of a visualization technique, in general.

Clearly, the metaball metaphor provides us with a visually rich environment to depict entities in a software system along with visual techniques that enable mapping of software structure and dynamic behavior onto highly intuitive visual renderings.

5. Metaball visualization of a Java 3D program

To demonstrate the applicability and effectiveness of 3D metaball visuals in combination with program slicing, we created 3D visuals for the application itself—MetaViz, our Java 3D metaball visualization program. This program consists of four packages and 26 classes and with a total of 10,000 lines of source code. Fig. 9 shows a zoomed-in view of MetaViz as visualized using itself and using different colors for related groups

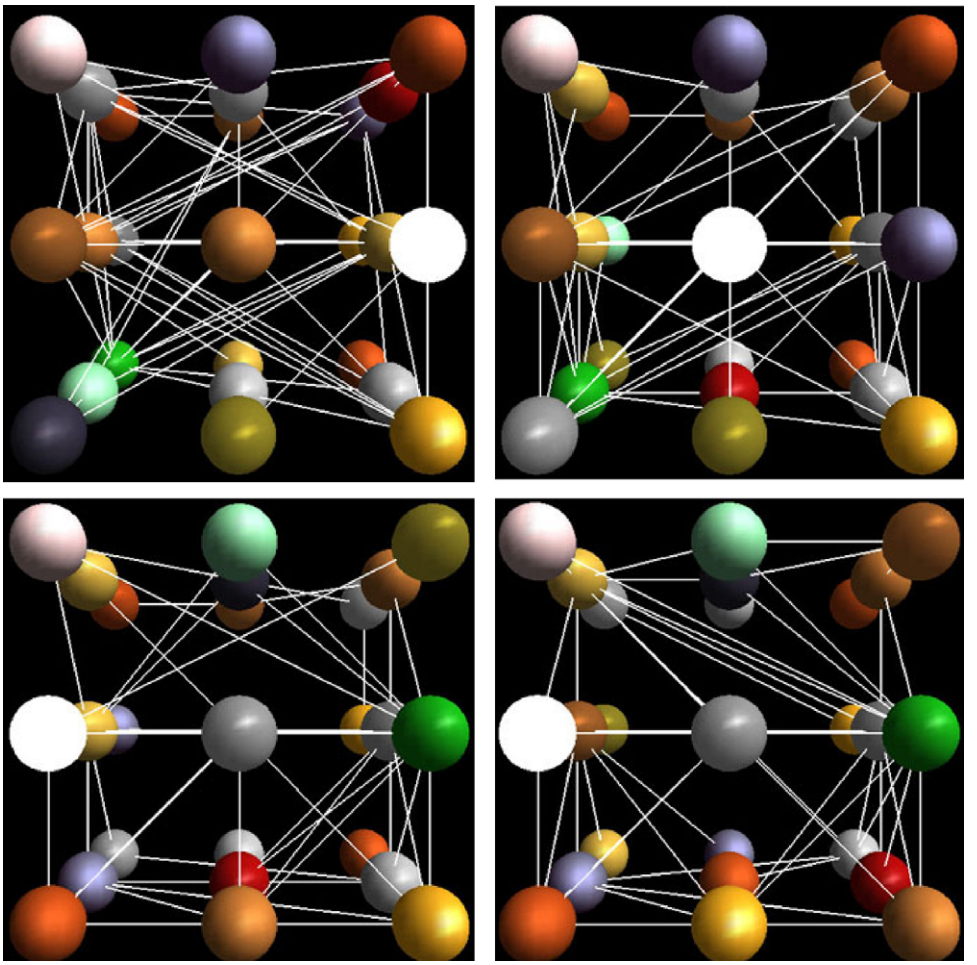

Fig. 7. Different stages in the layout process.

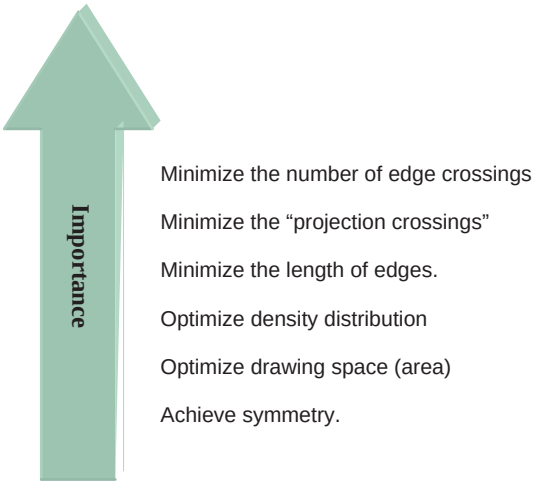

Fig. 8. Visualization criteria affecting layout of entities.

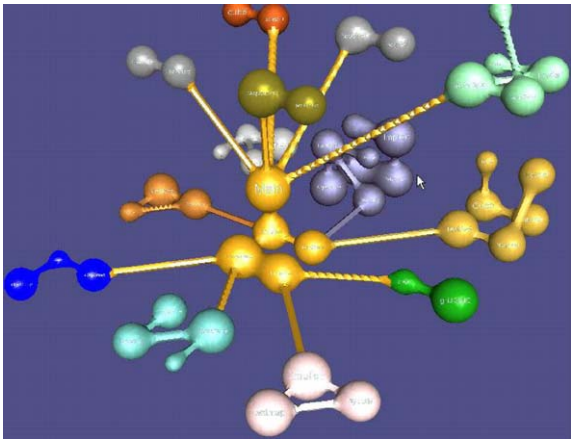

Fig. 9. A clip of the MetaViz source program visualized using MetaViz itself (Screen capture).

of classes. We use this not as an example of a large software system, but as a program which allows us to demonstrate how the presented techniques can benefit the comprehension of software systems and their structures.

5.1. Hierarchic view

One of the first tasks in program understanding is to obtain an overview of the software entities, their sizes and static structural relationships such as inheritance. The metaball inheritance network is a 3D graphic structure that corresponds to a network depicting the program's inheritance hierarchy. Nodes denote packages/classes and branches denote the inheritance relationship. The size of the node is indicative of the entity size and the color is indicative of the package to which that the class belongs. With suitable text call outs associated with a software entity, this inheritance network makes it easier for a user to gauge the maximum/minimum sizes of classes and also their numbers. It also enables simple analysis such as identification of classes which can be potential cases, say, for refactoring a class into their super classes, or break up highly bloated classes which have far too many methods, etc.

Fig. 10 (left) shows a hierarchical representation of the MetaViz system as provided by Sun One IDE (or for that matter most other IDEs). The view provides the list of classes and packages included in the software system and its class hierarchy. Limitations of this view are that for large systems it will require scrolling through the listing making it difficult to identify structural dependencies among the different parts of the program.

Fig. 10 (right) shows the same class hierarchy using our metaball approach. At the top level, we see the various packages of the system. Packages are identified by their names and an associated unique color.

The size of the metaballs is proportional to the contained code size of the corresponding package/class. Connected to the packages are the classes associated with a particular package. The color and size scheme also applies to metaballs on the class level. Other options include the possibility to select packages of interest. The selection process will lead to a customized view, showing the classes that belong to the packages and dimming the classes that do not belong. We will now demonstrate how the metaball technique can be applied to visualize program slices and program slicing-based method coupling metrics (SMCP).

5.2. Design evaluation

Software systems have to be flexible in order to cope with evolving requirements [2,4,5,73]. Although good software engineering practice encourages programmers to plan for future modifications, not every future design change can be predicted. Traditional software metrics are used for this purpose, but are often limited by their textual or tabular representation. For example, Fig. 11 shows a partial snapshot of different coupling measurements for MetaViz program that were computed using Sun One metrics add-on.

One of the major shortcomings of a tabular representation is that it does not intuitively correspond to a programmer's mental model of a software design. Typically, a software design is represented and documented on a graphical abstraction level using, for example, UML class models rather than a textual representation. Additionally, from a user perspective,

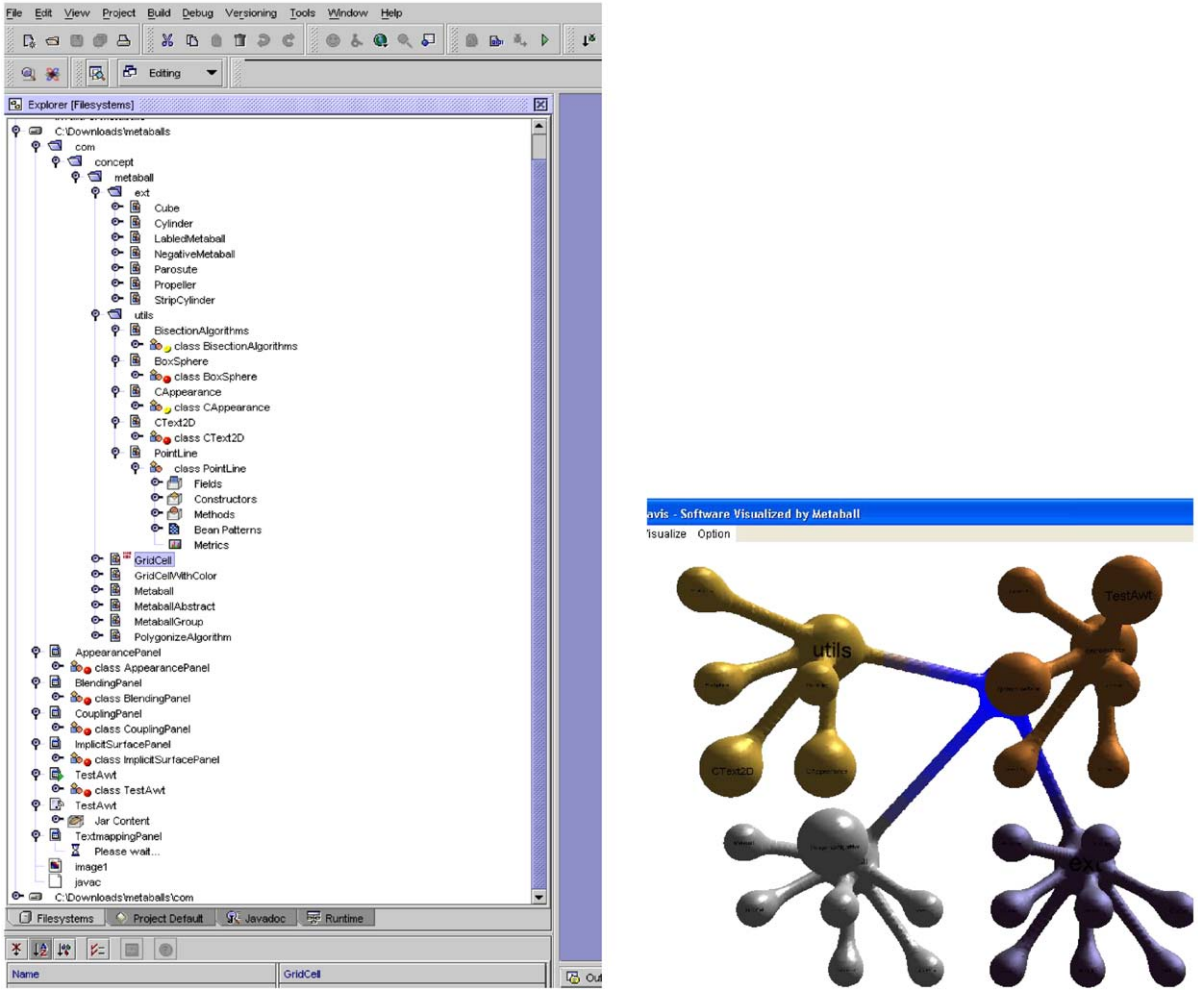

Fig. 10. Hierarchical structure in SunONE (left); same structure using Metaviz's inheritance network (right).

these measurements are somewhat disconnected making it difficult for a programmer to interpret and correlate the metrics within the context of the overall design and structure of a software system.

For the purpose of demonstration, we focused on the MPC metric. We applied the MPC metric for our MetaViz environment. Fig. 12 shows a snapshot of a tabular representation of the MPC measurements. The table illustrates the MPC that exists between a pair of classes. These metrics are then used to visualize the information using metaballs, with particular interest in the influence over neighboring entities.

The metaball diameter is indicative of class size; the fusion between two classes indicates the existing MPC among two classes and the strength of the coupling among classes corresponds to the diameter of the connecting cylinders among the entities. An example of strong coupling is shown in Fig. 13, where the two classes CText2D and Text are strongly coupled with each other. The example illustrates how the metaball approach can be applied intuitively, not only to pictorially visualize a program structure but also to represent and visualize design measurements. It facilitates the task of analyzing metrics directly in the context of the overall program structure.

5.3. Combining program slicing with MPC

For large software systems, the metaball visualization technique if directly used has the same problems as the more conventional graph-based visualization techniques in its ability to scale and visualize large amount of information. Usually, programmers tend to apply an as needed comprehension approach, to comprehend only

Metrics

Class Name	WMC	CBO	RFC	MPC	DIT
AppearancePanel	10	6	71	111	4
BlendingPanel	2	5	51	67	4
CouplingPanel	2	6	63	86	4
ImplicitSurfacePanel	2	6	54	70	4
TestAwt	7	12	35	78	5
TextmappingPanel	4	5	58	76	4
com.concept.metaball.GridCell	13	11	19	34	1
com.concept.metaball.GridCellWithColor	8	3	9	10	2
com.concept.metaball.Metaball	19	6	26	16	2
com.concept.metaball.Metaball.bisection	4	2	3	1	2
com.concept.metaball.MetaballAbstract	21	10	45	43	1
com.concept.metaball.MetaballGroup	26	15	37	40	4
com.concept.metaball.PolygonizeAlgorithm	92	3	26	67	1
com.concept.metaball.ext.Cube	10	3	13	8	2
com.concept.metaball.ext.Cylinder	11	5	20	19	

Fig. 11. Snapshot of coupling measurements for MetaViz program.

	1.	2.	3.	4.	5.	6.	7.	8.	9.	10	
1.		18	10		18				10		5
2.				12	7				10		7
3.				2	13		10		5		5
4.					5	10	7		5		8
5.						4			10		
6.							5				

Fig. 12. MPC measurement snapshot for MetaViz.

those parts of the systems that are relevant for the particular task. Software metrics-based on program slicing can be used to address this problem, by reducing comprehension complexity, by focusing on a specific comprehension aspect rather than the overall comprehension of the system. Additionally, the metaball metaphor can also be extended to apply an as "needed approach" to the visualization of software structures, by displaying only those parts of the system that are relevant for the current comprehension task. Combining software visualization with source code analysis, in this case program slicing, can enhance and improve the applicability of these techniques. Specifically, we create metaball visuals to display the SMPC for our MetaViz system. Fig. 14 shows a snapshot of the SMPC measurements. The classes that are included in the slice and considered in the SMPC are highlighted in yellow. Additionally, MPC metrics for the classes included in the slice were updated, including only those MPC measurements that are included in the program slice, resulting in a reduction of the MPC values for these classes.

For visualization of the MPC metrics, we update the fusion among the different metaballs by reflecting only the MPC included in the slice. One approach to display a slice is by highlighting the modules (blobs) and call relationships that belong to the slice in the original metaball diagram, showing the complete program. Another approach is to display a metaball sub-diagram that is constructed from the original metaball diagram by hiding/dimming all modules (blobs) and their calling relationships that do not belong to the slice (cf. Fig. 15).

6. Conclusions

It is a well-known fact that a major share of systems development effort goes into the comprehension of large systems and their source code, about which we usually know very little [4]. The large and complex programs developed and maintained in current software environments are the ones that can most benefit from the visualization and source code analysis techniques presented in this research. Our paper first presented a comprehensive review of techniques for program comprehension through software metrics and visualization. This was followed by introducing a novel approach of applying the metaball metaphor to visualize source code and source code analysis information. Specifically, in combination with program slicing, this technique provides a rich, powerful, and intuitive method of visual presentations that can considerably enhance and accelerate program comprehension. Combined with program slicing, it allows for a reduction of the information to be displayed, and it also enables us to provide additional source code insights that can be applied to a variety of source code-based comprehension tasks (e.g. debugging, testing, performance analysis, etc.). We have illustrated how the metaball approach

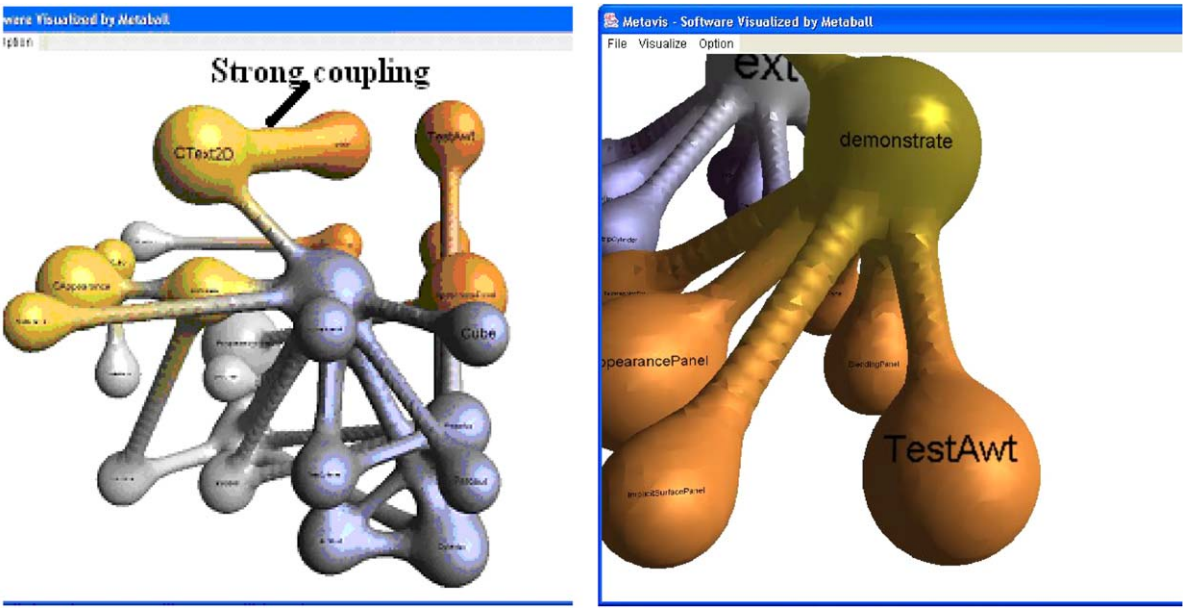

Fig. 13. Metaball visuals of MPC measurements (Fig. 12).

	1.	2.	3.	4.	5.	6.	7.	8.	9.
1.		18	10		18				10
2.				12	7				10
3.				2	13		10		5
4.					5	10	7		5
5.						4			10

Fig. 14. Slicing-based MPC measurement for MetaViz (Screen capture).

can be combined with program slicing to visualize software metrics.

Given the complexity of software and the different problem solving characteristics of programmers, it is now well recognized that there is unlikely to be any one single visualization metaphor that can be considered most optimal for software visualization. Instead, different metaphors may be better suited to specific program comprehension purposes and for particular types of analyses results. In our opinion, the metaball metaphor is rich and has the potential to be a very good candidate for a number of software reverse engineering tasks. This is largely because it is highly effective in visually capturing the relationships between software entities such as coupling, relevancy, and influence relationships that play a vital role in virtually all program comprehension and reverse engineering tasks. While there is a large body of powerful software available for visualizing metaballs both commercial and public domain, all of it is tailored toward applications in domains such as molecular modeling, animation, and electronic gaming. Our use of metaballs is similar but not identical to those domains. It was important to develop MetaViz, our own metaball visualization software. MetaViz is specifically tailored to producing the kind of visuals and 3D interactions that have been elaborated upon in earlier sections. Presently it includes a JDBC interface to a database to obtain the data/values resulting from source code analysis. These are then modeled and rendered as 3D visuals.

MetaViz development and application have just commenced. A number of tasks/extensions are planned. The more important of these are listed below:

- We are investigating the possibility of undertaking actual case studies in comprehension of large programs from industry/external organizations. These case studies will allow us to properly validate the effectiveness and applicability of the metaball metaphor for large software visualization.
- Usability studies are another important aspect. The effectiveness of software visualization tools must be validated beyond merely anecdotal evidence. Usability of these tools must be evaluated for functional, practical, aesthetic and problem solving support aspects.
- Presently, MetaViz is a stand alone tool that uses JDBC to access data created within CONCEPT [52], a software engineering automation environment being developed here. We would attempt the integration of MetaViz into the CONCEPT environment

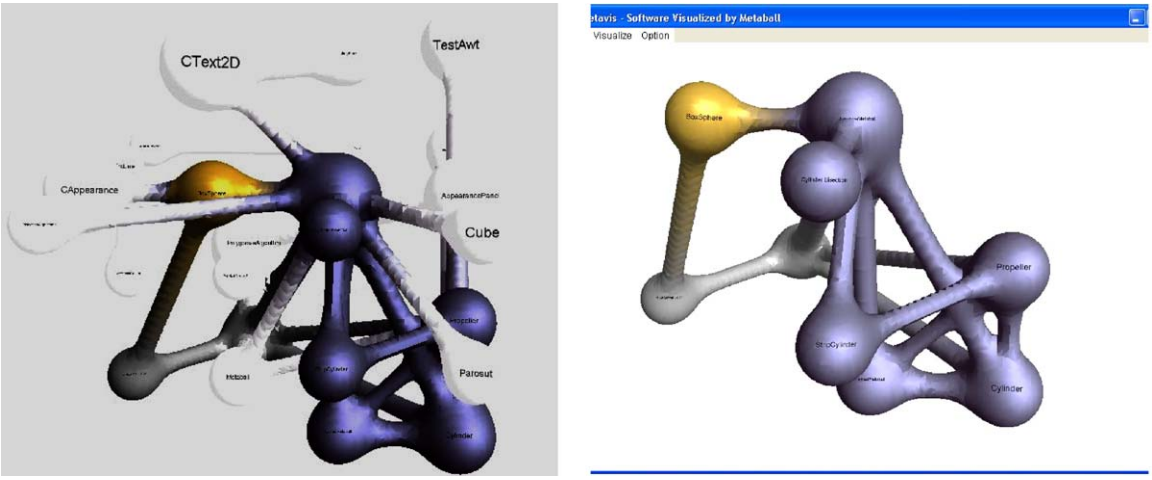

Fig. 15. Slicing-based MPC measurement for MetaViz (Screen capture).

with the intention of supporting round trip software engineering activities.

Acknowledgements

We gratefully acknowledge partial funding support through research Grants from NSERC, Canada and Concordia University ENCS Faculty Research Funds. We also thank our Master's student Jian Qun Wang for his help in carrying out the software implementation.

References

[1] Bassil S, Keller RK. Software visualization tools: survey and analysis, Proceedings of the IEEE ninth international workshop on program comprehension (IWPC'01), 2002. p 7–17.

[2] Favre JM. G^{SEE}: a generic software exploration environment. Proceedings of the ninth international workshop on program comprehension (IWPC'2001), Toronto, Canada, May 2001, p. 233–44.

[3] Harel D. Biting the silver bullet, toward a brighter future for system development. IEEE Computer 1992;25(1):8–20.

[4] Mayrhauser A, Vans AM. Program understanding behavior during adaptation of large scale software. Proceedings of the sixth international workshop on program comprehension, IWPC '98, Ischia, Italy, June 1998. p. 164–72.

[5] Rilling J. Maximizing functional cohesion of comprehension environments by integrating user and task knowledge. Proceedings of the eighth IEEE working conference on reverse engineering (WCRE 2001), Stuttgart, Germany, October 2001. p. 157–65.

[6] Sanlaville R, Favre JM, Ledru Y. Helping various stakeholders to understand a very large software product. Proceedings of the European conference on component-based software engineering, September 2001.

[7] Storey M-A, Fracchia F, Müller H. Cognitive design elements to support the construction of a mental model during software exploration. Journal of Software Systems 1999;44:171–85 (special issue on program comprehension).

[8] Weiser M. Program slicing. IEEE Transactions on Software Engineering 1982;SE-10(4):352–7.

[9] Basili V, Briand L, Melo W. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical report, CS-TR-3395, University of Maryland, Department of Computer Science, 1995.

[10] Furnas G. Generalized Fisheye views. Proceedings of the SIGCHI human factor in computing, 1986. p. 18–23.

[11] Nielson GM, Hagen H, Mueller H. Scientific visualization: overviews, methodologies, and techniques. Silver Spring, MD: IEEE Computer Society Press; 1997.

[12] Ball T, Eick SG. Software visualization in the large. IEEE Computer 1996;29(4):33–43.

[13] Baker MJ, Eick SG. Space-filling software visualization. Journal of Visual Languages and Computing 1995;6: 119–33.

[14] Kreuseler M, Schuman H. Information visualization using a new focus + context technique in combination with dynamic clustering of information space. Proceedings of the ACM workshop on new paradigms in information visualization and manipulation, Kansas city, 1999. p. 1–5.

[15] Michaud J, Storey MAD, Muller HA. Programs, integrating information sources for visualizing java. Proceedings of the international conference of software maintenance (ICSM'2002), Italy, 2001.

[16] Price B, Baecker R, Small I. A principled taxonomy of software visualization. Journal of Visual Languages and Computing 1994. p. 211–66.

[17] Reiss SP, Cruz IS. Practical software visualization. CHI'94 workshop on software visualization.

[18] Shneiderman B. Tree visualization with tree-maps: a 2-D space-filling approach. In: ACM transactions of the computer–human interaction, vol. 11(1), 1992, p. 92–9.

[19] Shneiderman B. Designing the user interface, 3rd ed. Reading, MA: Addison-Wesley; 1997.
[20] Knight C, Munro M. Visualising software—a key research area. Proceedings of the international conference on software maintenance; ICSM'99. New York: IEEE Press; 1999.
[21] Rilling J, Karanth B. A hybrid program slicing framework. IEEE international workshop on source code analysis and manipulation SCAM 2001, Florence, Italy, November 2001.
[22] Blinn JF. A generalisation of algebraic surface drawing. ACM Transactions on Graphics 1982;1(3):135–256.
[23] Bloomenthal J. Polygonization of implicit surfaces. Computer Aided Geometric Design 1988;5(4):341–55.
[24] Stasko J, Domingue J, Brown MH, Price BA, editors. Software visualization: programming as a multimedia experience. Cambridge, MA: MIT Press; 1998.
[25] Wyvill G, McPheeters C, Wyvill B. Data structure for soft objects. The Visual Computer 1986;2(4):227–34.
[26] Wyvill G, McPheeters C, Wyvill B. Animating soft objects. The Visual Computer 1986;2(4):235–42.
[27] Wyvill B, Wyvill G. Field functions for implicit surfaces. Visual Computer 1989;5:75–82.
[28] Rilling J, Mudur SP. On the use of metaballs to visually map source code structures and analysis results onto 3D space. Proceedings of the IEEE WRCE 2002.
[29] 3D ARK, 3D related software list, http://www.3dark.com/resources/products/softwarelist.htm.
[30] Basili VR, Briand LC, Melow WL. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 1996;22(10):751–61.
[31] Fenton N. Software metrics: a rigorous approach. London: Chapman & Hall; 1991.
[32] Li B. A hierarchical slice-based framework for object-oriented coupling measurement. TUCS technical report no. 415, Turku Centre for Computer Science, Turku, Finland, July 2001.
[33] Li W, Henry S. Object-oriented metrics that predict maintainability. Journal of Systems and Software 1993; 23(2):111–22.
[34] Martin R. OO design quality metrics—an analysis of dependencies. Position paper, workshop on pragmatic and theoretical directions in object-oriented software metrics, OOPSLA'94, October 1994.
[35] Hitz M, Montazeri B. Chidamber & Kemerer's metrics suite: a measurement theory perspective. IEEE Transactions on Software Engineering 1996;22(4):267–71.
[36] Briand L, Devanbu P, Melo W. An investigation into coupling measures for C++. Technical Report ISERN 96-08, IEEE ICSE'97, Boston, USA, May 1997.
[37] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 1994;20(6):476–93.
[38] Agrawal H, Horgan, J. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN'90 conference on programming language design and implementation, SIGPLAN notices, vol. 25(6), 1990; p. 246–56.
[39] Agrawal H, DeMillo R, Spafford E. Debugging with dynamic slicing and backtracking. Software—Practice and Experience 1993;23(6):589–616.
[40] Harman M, Hierons RM, Danicic S, Laurence M, Howroyd J, Fox C. Pre/post conditioned slicing. Proceedings of the IEEE international conference on software maintenance (ICSM'2001), Florence, Italy, 2001.
[41] Harman M, Danicic SA. A new algorithm for slicing unstructured programs. Journal of Software Maintenance 1998;10(6):415–41.
[42] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 1990;12(1):26–61.
[43] Horwitz S, Reps T. The use of program dependence graphs in software engineering. In: Proceedings of the 14th International conference on software engineering, Melbourne, Australia, 1992. p. 392–411.
[44] Larsen LD, Harrold MJ. Slicing object oriented software. In: Proceeding of the 18th international conference on software engineering, March 1996.
[45] Korel B, Laski J. Dynamic program slicing. Information Processing Letters 1988;29(3):155–63.
[46] Gopal R. Dynamic program slicing-based on dependence relations. In: Proceedings of the conference on software maintenance, 1991. p. 191–200.
[47] Kamkar M, Krajina P. Dynamic slicing of distributed programs. In: Proceedings of the international conference on software maintenance, October 1995. p. 222–9.
[48] Korel B. Computation of dynamic slices for unstructured programs. IEEE Transactions on Software Engineering 1997;23(1):17–34.
[49] Zhao J. Dynamic slicing of object-oriented programs. Technical-Report SE-98-119, Information Processing Society of Japan, May 1998. p. 17–23.
[50] Gupta R, Soffa M, Howard J. Hybrid slicing: integrating dynamic information with static analysis. ACM Transactions on Software Engineering and Methodology 1997; 6(4):370–97.
[51] Lyle J, Weiser M. Experiments on slicing-based debugging tools. Proceedings of the first conference on empirical studies of programming, 1986. p. 187–97.
[52] Rilling J, Seffah A. The CONCEPT project—applying source code analysis to reduce information complexity of static and dynamic visualization techniques. IEEE VISSOFT workshop, Paris, 2002.
[53] Tip F. A survey of program slicing techniques. Journal of Programming Languages 9/1995;3(3):121–89.
[54] Harman M, Okulawon M, Sivagurunathan B, Danicic S. Slice-based measurement of coupling. IEEE/ACM ICSE workshop on process modeling and empirical studies of software evolution (PMESSE'97), Boston, MA, 17th–23rd May 1997. p. 28–32.
[55] Ott L, Thuss J. Slice-based metrics for estimating cohesion. Proceedings of the IEEE-CS international software metrics symposium, 1993. p. 71–81.
[56] Ott L, Thuss J. The relationship between slices and module cohesion. Proceedings of 11th international conference on software engineering 1989. p. 192–204.
[57] Rilling J, Meng WJ, Ormandjieva O. Context driven slicing-based coupling measures. Proceedings of the 20th IEEE international conference on software maintenance (ICSM'04), Vol. 00, Chicago, IL, 2004. p. 532.
[58] Mackinlay J, Robertson G, Card S. The perspective wall: detail and context smoothly integrated. Proceedings of the SIGCHI human factors in computing, 1991. p. 173–9.

[59] Lamping J, Rao R, Pirolli P. A focus + context technique based on hyperbolic geometry for visualizing large hierarchies. Proceedings of the SIGCHI human factors in computing systems, 1995. p. 401–8.
[60] Nielsen J. 2D is better than 3D. AlertBox. http://useit.com/alerbox/981115.html; 1998.
[61] Pirolli P, Card SK, Van Der Wege, Mija M. Visual information foraging in a focus + context visualization. In: Proceedings of the ACM conference on human factor in computing systems (CHI-01), Seattle, 2001. p. 506–13.
[62] Walker RJ, Murphy GC, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing dynamic software system information through high-level models. Proceedings of the OOPSLA'98, SIGPLAN notices 33(10), October 1998. p. 271–83.
[63] Maletic JI, Leigh J, Marcus A, Dunlap G. Visualizing object-oriented software in virtual reality. Proceedings of the ninth international workshop on program comprehension (IWPC 2001), Toronto, Canada, May 12–13, 2001. p. 26–35.
[64] Robertson GG, Mackinlay JD, Card SK. Cone trees: animated 3D visualizations of hierarchical information. Proceedings of the CHI'91 conference on human factors in computing systems 1991. p. 189–94.
[65] Knight C, Munro M. Visualising the non-existing. Proceedings of the IASTED international conference: computer graphics and imaging, Hawaii, USA, 2001.
[66] Hopkins J, Fishwick PA. A three-dimensional human agent metaphor for modeling and simulation. Proceedings of the IEEE 2001;89(2):131–47.
[67] Dwyer T. Three dimensional UML using force directed layout. Proceedings of the Australian symposium on information visualization, 2001.
[68] Demeyer S, Ducasse S, Lanza M. A hybrid reverse engineering platform combining metrics and program visualization. In: Proceedings of the WCRE'99, New York: IEEE Press; 1999. p. 175–87.
[69] Systä T, Koskimies K, Müller HA. Shimba—an environment for reverse engineering Java software systems. Software—Practice and Experience (SPE) 2001;31:371–94.
[70] http://www.rigi.csc.uvic.ca/.
[71] Franck G, Sardesai M, Ware C. Layout and structuring object oriented software in three dimensions. Proceedings of the CASCON 1995.
[72] Sugiyama K, Misue K. Visualization of structural information: automatic drawing of compound digraphs. IEEE Transactions on Systems, Man and Cybernetics 1991;21(4):876–92.
[73] Van Deursen A, Kuipers T. Building documentation generators. In: Proceedings of the international conference on software maintenance (ICSM'99). Silver Spring, MD: IEEE Computer Society Press; 1999. p. 40–9.