SoftSphere: Dynamic Visualization of Coupling in Java Programs

Andrew Sutherland
Software Research Lab
Department of Computer Science
University of Saskatchewan
andrew.sutherland@usask.ca

ABSTRACT
The creation of an application for the effective visualization of software is a task requiring several steps. The steps required and the means for doing so are discussed. A prototype visualization application for visualizing software coupling is presented along with an assessment of the techniques presented therein. 3-D rendering is offered as the visualization medium for the prototype and answers are sought to a number of questions regarding the usefulness of this medium for visualizing software.

AUTHOR KEYWORDS
Software visualization, source code mining, three-dimensional visualization

1 INTRODUCTION

The development and maintenance of large software systems often involves performing complex tasks that require in-depth understanding and knowledge of the underlying code, documentation, and other design artifacts. The use of visual aids can simplify these tasks by removing the necessity to remember large amounts of information all at once.

Visualizations can be used to represent many different qualities or properties of software. Software visualizations allow a user to view various aspects of a system such as how a system evolves over time [4][18], the collaboration between its developers [14], execution of code at run-time [10], the logical structure of the software [2], or metrics pertaining to the code itself [5]. The purpose of representing these various aspects of software visually is to allow the user to recognize patterns and non-trivial relationships that without visual aids would have been too complex to recognize otherwise. For example, a program that visualizes collaboration between developers on a project may allow a manager to identify a particular developer that contributes to coupling between many of the components she has worked on. This conclusion may not have been as easily derived by looking at logs or written reports of what was changed.

The successful communication of information visually relies on determining the correct metaphor that is used to map software facts to visual entities.

1.1. Basic Stages of Visualization

Ware [17] states that all types of information visualization include four basic stages:

- The collection and storage of the data itself
- The preprocessing designed to transform the data into something we can understand
- The display hardware and the graphics algorithms that produce an image on the screen
- The human perceptual and cognitive system

The stages are interdependent on one another. For example, if the data collected and stored is incomplete or not suitable to the type of visualization required, the resulting visualization produced from the subsequent stages may not have the desired effect. Thus, in order to maximize the effectiveness of the visualization, it is necessary to take into account all four of these stages when creating a visualization application.

An initial implementation, SoftSphere, was constructed as an experimental platform on which various visualization techniques could be used. This paper discusses what visualization techniques were incorporated into the application and the basis behind them. Observations were made to determine what techniques could be improved and what additional techniques could be introduced for a more effective visualization.

3D rendering was the medium used to build the visualization. There is encouraging evidence that rendering in 3D does provide certain benefits [15], however it is still unclear what effects and techniques should be incorporated into a 3D visualization in order to maximize understanding. One of the goals of creating this application was to determine if 3D rendering is appropriate for representing software.

2 RELATED WORK

Building an effective visualization requires familiarity with a number of related topics. There are a number of areas that have strong ties to software visualization. This section introduces some of the prerequisite knowledge necessary to construct an effective visualization application.

2.1. Data Mining

Ware's first two stages of visualization involve collecting and storing the necessary information for visualization, followed by the transformation of the information into something we can understand. Presented here is an overview of how some visualization applications implement these first two important steps by mining data from CVS repositories and transforming the data into a representation that can be easily understood.

CVS (Concurrent Versioning System) is the version management method used for many open source projects. The information stored in CVS repositories contains information pertaining to file administration such as when files were last altered and by whom, and other information regarding what was changed with each commit. These qualities makes CVS repositories attractive targets for data mining as many types of useful information spanning a long time period can be found in a single, accessible place.

softChange [5][6] is a one such application that uses a CVS repository as one of its sources for software evolution information.

One of the difficulties in working with CVS repositories is that CVS does not keep track of what individual files were modified concurrently. The creators of softChange stress that it is important to know what files were modified at the same time, as concurrent change indicates a relationship between these files. softChange denotes a *modification request* (MR) as the set of files committed simultaneously in a single CVS commit. The application then analyzes source code contained in the files of each MR, and creates a factbase by listing the function, methods, and classes that have been added, modified, or removed from one MR to the next. softChange also cross-references these facts with other sources such as Bugzilla reports and mail archives in order to obtain a more concrete factbase.

The drawback to the method used by softChange is that is unlikely that a modification request by their definition will always contain related files. A single MR may contain files from multiple tasks that happened to be performed in between commits. This condition will result in otherwise unrelated files being logically coupled. The opposite may also occur, where files with some logical connection are committed separately, resulting in missed file relations. In summary, the procedure employed by softChange may make too large of an assumption about how developers use CVS to manage open source projects.

ProjectWatcher [14] is another application that mines information obtained from a CVS repository. However, the approach taken by ProjectWatcher differs significantly from the approach taken by softChange. ProjectWatcher tracks local interaction history of developers to support awareness in team-based software development. To do this, a shadow CVS repository is used to track changes as each developer makes them. A factbase of user edits is kept up to date as auto-commits are performed on the shadow repository. A TXL [16] program is used to uniquely identify all entities in the software (such as modules, methods, variables, etc.). Activity information can then generated by cross-referencing the shadow repository with the name factbase generated by the TXL program. This approach has a significant advantage over the modification request method used by softChange as it does not rely on assumptions made about the relationships between entities.

2.2. Visualizing in 3D

Ware's final two stages of visualization [17] involve using graphics algorithms to produce an image on the screen that affords better understanding of the underlying factual data. There is evidence that using 3D graphics algorithms to produce these images has certain benefits related to how the human visual system perceives and processes information [7].

Radfelder et al. [11] attempt to improve understanding of UML diagrams by introducing a third dimension. They claim that traditional 2D UML diagrams can be irritating to the user if two related entities of interest are located on opposing ends of the diagram. This problem occurs especially in complex class diagrams with many relations between entities. Their solution to this problem consists of dynamically bringing the related entities to the foreground of the visualization and moving the other entities to the background. The authors claim that this form of transformation – moving things of interest smoothly to the front and moving things which have lost their particularity to the background – is closer to the way human beings interact with physical objects. That is, real-world objects we are not interested in typically do not disappear, but are moved aside to our peripheral vision.

Balzer et al. [2] use an interesting 3D representation of software in their application, Software Landscapes (see Figure 1). Balzer claims that information density in a single view should be maximized under the constraint of comprehensibility. That is, there should not be more visual data in a single view than can be easily comprehended by the human visual system. The method proposed by Balzer realizes this rule by allowing the user to easily move between levels of abstraction and to different parts of the visualized system. The claim is that object-oriented software can be naturally mapped to the landscape metaphor due the hierarchical nature of software. Balzer also argues that because human beings naturally know how to navigate through a landscape (i.e. enter/exit structures, move around obstacles) that this metaphor will allow users to more easy familiarize themselves with the application, which will in turn allow the user to understand the software more effectively.

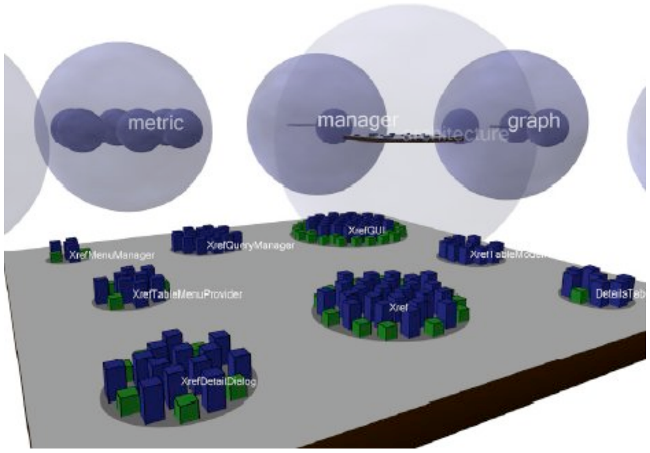

Figure 1 - Software Landscapes

2.3. Clustering and Visualization Layout

Spatial arrangement of visual entities is often used to convey relationships between the correlating entities in the software. This is because is it a natural human tendency to group related objects, so it makes sense to group objects visually in order to express their logical relationship. Determining a layout for visual entities can be assisted by using a clustering algorithm. Clustering in itself is a very sophisticated reverse engineering technique [1] and a complete summary of the field would be beyond the scope of this paper. Discussed here is some of the literature related to clustering and how it can be applied to software visualization.

Anquetil et al. [1] performed a comparative study of clustering algorithms and their application to software remoduralisation. Their definition of clustering is an activity in reverse engineering that consists of gathering software entities (modules, routines, etc.) that compose the system into meaningful (highly cohesive) and independent (loosely coupled) groups. Anquetil states that is necessary to define a description of each entity such that it can be clustered according to some scheme, and to define a coupling between entities. The coupling between entities determines when two entities form a cohesive cluster. Coupling may be defined as a dependency – i.e. a direct link, or it may be defined by similar behaviour – i.e. a sibling link. For example, in the case of Java classes, if coupling was defined as external method calls, this would be a direct link coupling. The more method calls between two classes, the higher the coupling. If coupling surpasses a certain threshold, these classes should be in the same cluster. On the other hand, if coupling is defined as the number of times classes call the same methods of another class; this would be sibling link coupling.

Layout of visualizations for software often incorporate clustering as described above to arrange entities visually. GEVOL [4] visualizes software evolution using a series of graphs representing the state of the software at various points in lifetime of the software. Nodes represent entities (classes, methods, etc.) in a Java program and edges represent inheritance or calls. Coupling between the entities is represented as weights assigned to each node. The weight of each node is determined by the number of times it appears over the total number of versions of the software. The weighted nodes are fed into a force algorithm, automatically determining the layout of the graph.

Noack et al. [9] use a layout algorithm based on gravitation. Nodes representing entities with high cohesion are "attracted" to one another, while nodes that are more loosely coupled are "repelled". The result is a self-organizing system of nodes that automatically arrange themselves into clusters.

2.4. Visualization of Software Evolution

The changes software must undergo over its lifetime due to changing requirements and routine maintenance cause software systems to evolve over time [8]. Visualization techniques can be useful for extracting useful evolution data that is hidden among source releases, CVS repositiories, and maintenance logs.

Wu et al. [18] have adapted spectrographs to create a visualization application that conveys evolution information. By mapping software versions against software components and using colour to represent recent change, a notion is gained of what versions involved major changes to the system. This approach is useful for identifying points in the lifetime of the software where new functionality was added, or refactoring took place. Wu et al. denote these points as *software punctuations.* Their visualization is tested on a number of open source projects. Most of these projects had at least one or two release versions where the majority of the system had undergone substantial change.

GEVOL [4] visualizes software evolution by showing a series of snapshots in linear order. Each snapshot is representative of a particular aspect of the system at that time. Inheritance graphs, call-graphs, and control-flow graphs were some of the properties of the software that were visualized. Colour is used to represent recently added entities or modification to existing entities.

Software evolution and other temporal qualities of software present a special challenge to software visualization. Careful thought needs to be given on how to effectively visualize change.

3 PROBLEM

Software development is often performed with large development teams over a longer period of time. Even after the software is deployed, maintenance must be performed to ensure the continuing fulfillment of requirements. With object-oriented software it is desirable to retain modularity. That is, it is desirable to limit the dependencies between components. The evolution process that all software systems must undergo can often cause components to lose their modularity and become logically linked to other components. This has the unfortunate side-effect of making changes to software more complicated than they need to be, as changes in one component may cause unintended changes in behaviour in a dependent component.

4 APPROACH

A visualization of the software that clearly represents coupling and cohesion between software components may aid software developers in determining what changes to the system contributed to the coupling. Once the cause of the increased cohesion is identified, corrective action could then be taken to reduce the unwanted cohesion.

The approach taken by Wu et al. [18] with their evolution spectrographs allowed developers to quickly identify what versions of the system resulted in software punctuations. A similar approach could be used to determine what versions of the system resulted in increased coupling.

SoftSphere is an attempt at visualizing coupling between software entities in a Java program. The evolution of the system in terms of coupling is also visualized. While far from a complete development application, SoftSphere provides a framework to determine what visualization techniques can be used to visualize aspects of software such as coupling, cohesion, and software evolution.

5 IMPLEMENTATION

SoftSphere was designed to extract the syntactical structure of Java programs and visually represent the various entities and relationships composing the software. The goal is to use data mining techniques similar to what was described in Section 2 to generate a factbase. The resulting factbase is structured such that it is a simple task to transform the facts into visual primitives. A simple clustering algorithm is used to arrange the visual primitives allowing the user to quickly perceive what entities are more closely coupled. That is, the spatial position of visual entities reflects the strength of the relationship between their correlating entities in the software. Finally, the evolution of the system can be viewed by allowing the user to iterate through subsequent versions of the system.

5.1. Generating the Factbase

A source transformation language, TXL [16], is used to generate a factbase listing the various entities and relationships that compose a piece of Java software. A separate TXL program was used to derive each type of fact from the source code. The type of entity facts that were extracted from the source code include:

- Package entities
- Class entities
- Method entities
- Global Field entities
- Local Field entities

The types of relationship facts extracted from the source code include:

- Import facts
- Local Method Call facts
- External Method Call facts

The facts are written in text format. A different text file is used for each type of fact. The application parses the text file and creates an internal hierarchical structure of the software in memory. Each fact contains information relating where in the hierarchy it should be placed.

Package coupling	Imports
Class coupling	External Method Calls, Use of Type
Method coupling	Local Method Calls

Table 1- Definitions of coupling for various entities

The relationship facts are used to build a measure of coupling between entities, similar to the description of entity coupling given by Anquetil et al [1]. Coupling between package entities is defined as the number of imports between packages. That is, two packages are defined to be highly coupled when one of the packages imports many classes from the other package. The same concept is used for coupling between classes. Coupling for classes is determined by the number external method calls between classes. Local method calls determine the coupling between method entities. Table 1 summarizes the different definitions of coupling used for each software entity.

TXL provides an efficient means of gathering syntactical data from the software. It is possible to generate a new factbase each time the visualization is run, at the small expense of a few seconds of overhead processing time.

5.2. Visual Representation

The mapping of software entities and relationships to visual representation is performed in OpenGL. A number of features offered by OpenGL are used to create an effective 3D representation of the software's syntactical structure.

The scene is rendered using a perspective viewing projection. This viewing projection is the closest approximation to how the human vision system actually perceives reality. The basic concept behind this view is that distant objects appear smaller than objects that are closer.

This gives a more believable impression of a three dimensional scene.

Standard OpenGL lighting was used to automatically shade the objects, further enhancing the three-dimensional feel to the scene. Figure 2 demonstrates how these two effects used together create a three-dimensional representation of software.

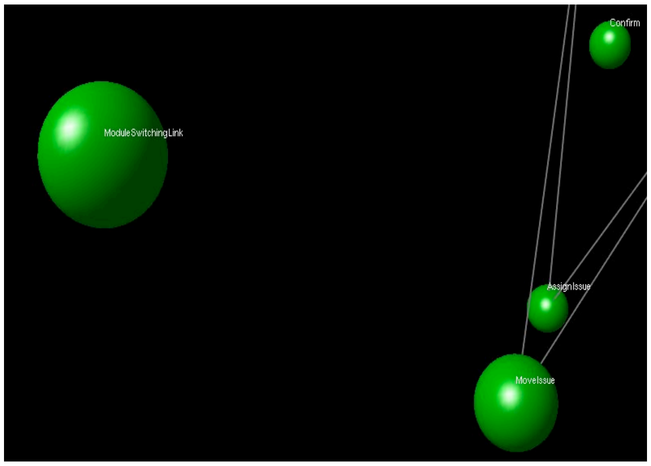

Figure 2 - 3D Effect achieved with lighting and perspective viewing projection

As evident from the title of the prototype, the various software entities are represented by spheres. In the current implementation of SoftSphere software attributes were not mapped to the shape of the entity. The sphere seemed to be the most aesthetically pleasing choice for this type of visualization. Relationships between the software entities were represented by edges between their corresponding spheres.

The user can navigate through several views of the software. Each view pertains to a different syntactical level of the software. The default level is the package-level view. The user can view the packages and their relationships to other packages (defined by Import facts). By selecting a particular package, the user can switch to a class-level view. Classes that are contained by the selected package will be displayed along with their relations to other classes in that package (defined by external method calls, and field types). Selecting a particular class will switch to the method-level view which display methods and global fields present in the selected class.

5.3. Visualization Layout

Finding an effective method of arranging visual entities in the viewing space can be difficult. The entities must be arranged such that there is a limited amount of occlusion and clutter. On top of this requirement is the desire to use spatial position of entities such that it conveys the coupling between entities in an intuitive manner.

A gravity-based clustering algorithm was devised in order to meet the two above requirements. The algorithm is based on the following two rules:

- Highly coupled nodes attract each other to a minimum distance based upon the strength of the coupling.
- Nodes with no or little coupling are repelled to a pre-defined distance.

So in effect, there are two types of force determining the locations of each node in the graph. Attraction is used to move coupled nodes towards one another, and stop when they have reached a suitable distance. Repulsion is used to move unrelated nodes away from one another, reducing clutter in the layout, and placing further emphasis on the clusters generated by attracted nodes. Initially, the nodes are place in a pseudo-random arrangement that ensures that no two nodes will be placed in approximately the same position. The clustering algorithm is then put into effect until the arrangement stabilizes itself. Figure 3 shows an example arrangement of a system generated by this algorithm. In the lower right of the arrangement is a group of highly coupled nodes.

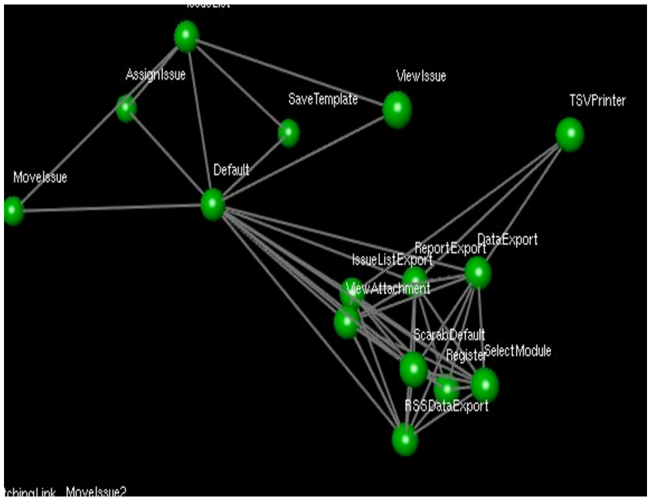

Figure 3 - Gravity based clustering algorithm to visualize coupling

5.4. Visualizing Software Evolution

To visualize the evolution of the software, each version of the software is loaded into the visualization. The user can iterate through the subsequent versions of the software. The visualization will automatically rearrange itself to reflect the changes made in each version. One of the drawbacks of systems such as GEVOL or Wu's spectrographs was that it was difficult to track changes from version to version of the software. The method of visualizing evolution employed by SoftSphere allows the user to dynamically view how the software changes over time, instead of viewing individual snapshots or the entire

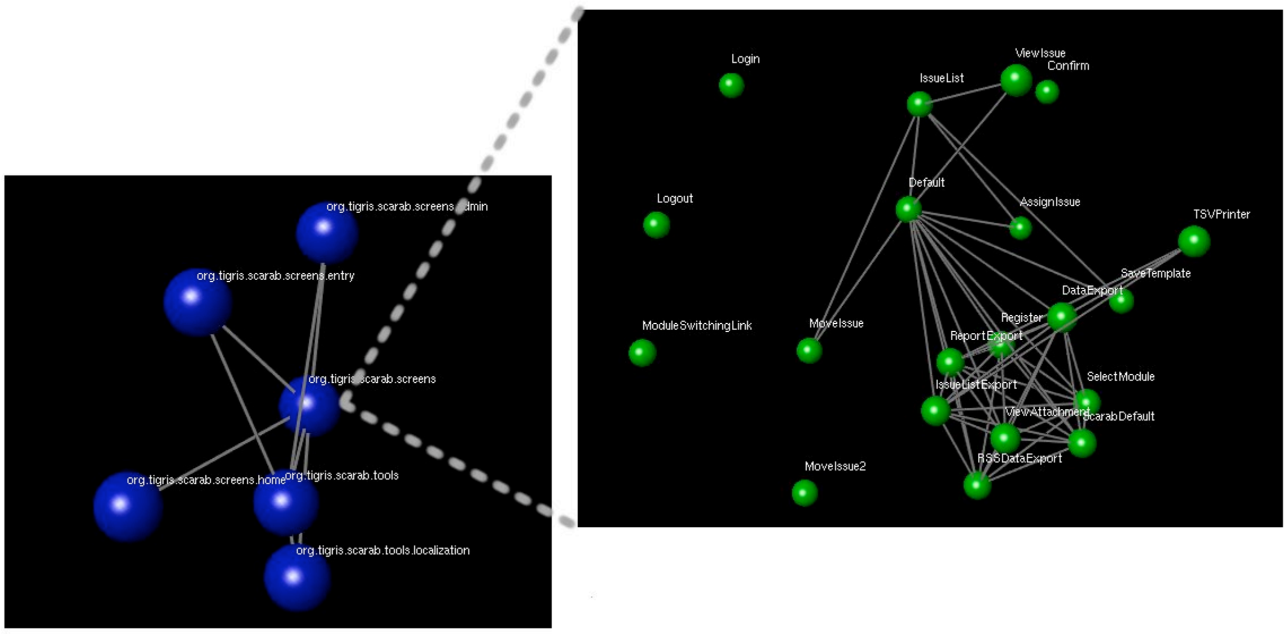

Figure 4 – Visualization of coupling between packages and classes for the Scarab Issue Tracking Application [13]

course of change all at once. A mental map of the software is preserved through the use of animation.

5.5. Interactivity

A certain level of interactivity was necessary in order to make full use of the three-dimensional rendering of the visualization. While 3D rendering may promote cognitive understanding, it also has the potential to increase the effects of occlusion. Since a sense of depth has now been introduced, background objects are more likely to be occluded by objects in the foreground. This problem presents the need for interactive techniques that allow the user to manipulate the visualization or the viewpoint.

The ability to rotate, zoom, and pan the camera view provide the user with a means to customize the viewpoint. The level of abstraction with which the visualization is viewed can also be changed. The software can be viewed at the package, class, method, or field level by selecting an entity and viewing the contents. For example, selecting a package entity will show the inner classes of that package. Figure 4 shows the expansion of a package node, allowing the interior classes of the package to become visible.

6 OBSERVATIONS

The design and implementation of SoftSphere was performed to see if it was viable to represent coupling and software evolution in a three-dimension rendering using a gravity-based clustering algorithm. The effectiveness of each technique was informally evaluated.

To verify that coupling was being represented properly, an open source project consisting of several packages and a moderate amount of classes per package was visualized using SoftSphere. Figure 4 shows a visualization of the open source project Scarab [13]. Scarab is a issue-tracking system implemented entirely in Java. Looking at the visualization of package coupling on the left, it is apparent that the *screens* package and the *tools* and *tools.localization* packages are all highly coupled. This would indicate to a developer that before making changes to any of these packages it would be wise to carefully consider the effects those changes would have on the other packages.

Expanding the *screens* shows approximately 20 classes inside of that package. An obvious cluster has formed in the lower right of the visualization. These classes all pertain to the export of data. Therefore, it makes sense that there is some degree of cohesion between these classes.

The use of node position to relate coupling works to give a general idea of coupling. However, detailed information on the nature of the coupling may be more useful. Integrating a feature that allows the user to obtain specific details about the code contributing to the coupling of the components could be increase the usefulness of the visualization significantly.

The use of three-dimensional rendering for SoftSphere allowed experimentation with interactive techniques that otherwise would not have been possible in a 2D rendering. It also presented subtle annoyances and problems that would not have been present otherwise. For example, occlusion becomes more of a problem, especially when using the perspective view. If viewed from an inappropriate angle, a single graph may have most of its entities occluded by a few entities at the forefront.

By introducing a three-dimensional perspective rendering, the effectiveness of mapping software attributes to the size of the visual entities is also reduced if not eliminated entirely. As the visible size of entities is reduced the farther back in the view they are placed, it is difficult to convey any meaning by physically making objects larger or smaller.

Determining if 3D rendering actually increases cognitive understanding is not a straight-forward task. By having people observe the visualizations and interact with the application can give a rough idea of how effective the visualization is at communicating software coupling, but formal usability study would be necessary to validate any positive or negative result.

Informal demonstrations of SoftSphere have produced encouraging reactions. Observers have expressed their desire to visualize their own programs with the application, and found interacting with the visualization to be an engaging experience. While far from solid evidence that the techniques have merit, it does provide some encouragement for continuing work on the prototype.

7 CONCLUSION

SoftSphere was presented as an application that incorporated a series of techniques to aid in creating an effective visualization of coupling between software entities such as Java packages, classes, methods, and fields. SoftSphere is not a full-fledged software development tool, but a prototype that tests various visualization techniques that could be incorporated into such a tool.

The four stages of information visualization suggested by Ware et al. were kept in mind throughout the creation of the application. TXL programs were used to gather software facts directly from the code and store them in a format suitable as a basis for a hierarchical representation of the software. A three-dimensional rendering of the scene was constructed using a perspective viewing projection and lighting. Interaction techniques supported the ability to interact with the software similar to how a person would interact with a physical object in the real world. A notion of the coupling between the various software entities was obtained through the use of a gravity-based clustering algorithm. A limited sense of how the software evolved in terms of coupling was gained by iterating through subsequent versions of the software and dynamically updating the visualization to reflect any changes made.

Possible avenues for future work include expanding the feature set of SoftSphere to include transition effects so that when changing the level-of-abstraction (e.g. switching from package level to class level) the selected entity can be shown in context to the level above it. The would further preserve the mental map the user has of the software and provide a more aesthetic experience when using the software.

Other definitions of coupling should also be considered. Instead of visualizing syntactical coupling, logical coupling (as done in softChange [5]) could be calculated using a variety of methods and visualized in a similar fashion.

8 REFERENCES

[1] N. Anquetil and T.C. Lethbridge. Comparitive Study of Clustering Algorithms and Abstract Representations for Software Remodularisation. *IEE Proceedings – Software*, 2003.

[2] M. Balzer, A. Noack, O. Deussen, C. Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems. *Joint EUROGRAPHICS – IEEE TCVG Symposium on Visualization,* Konstanz, Germany, 2004.

[3] M. Burch, S. Diehl, and P. Weißgerber. Visual Data Mining in Software Archives. *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization,* New York, 2005.

[4] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A System for Graph-Based Visualization of the Evolution of Software. *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization,* San Diego, California, 2003.

[5] D.M. German. Mining CVS Repositories, the softChange Experience. *Proceedings SEKE 2004 The 16th Internation Conference on Software*, Banff, 2004.

[6] D.M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. *Journal of Software Engineering Knowledge Engineering*, Edinburgh, 2004.

[7] P. Irani, M. Tingley and C. Ware. Using Perceptual Syntax to Enhance Semantic Content in Diagrams. *IEEE Comput. Graph. Appl.*, 2001.

[8] M.M. Lehman and L.A.Belady. Program Evolution – Process of Software Change. *Academic Press*, London UK, 1985.

[9] A. Noack and C. Lewerentz. A Space of Layout Styles for Hierarchical Graph Models of Software Systems. *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, New York, 2005.

[10] A. Orso, J.A. Jones, M.J. Harrold, and J. Stasko. Gammatella: Visualization of Program-Execution Data for Deployed Software. *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.

[11] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. *AVI '00: Proceedings of the working conference on Advanced visual interfaces, Palmero*, Italy, 2000.

[12] G. Robles, J.M. Gonzalez-Barahona, and R.A. Ghosh. GlueTheos: Automating the Retrieval and Analaysis of Data from Publicly Available Software Repositories. *MSR 2004*, Edinburgh, 2004.

[13] Tigris.org: Open Source Software Engineering Tools. April 28th. <http://scarab.tigris.org/>.

[14] K.A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a Software Developer's Local Interaction History. *MSR 2004*, Edinburgh, 2004.

[15] M. Tavanti and M. Lind. 2D vs. 3D, Implications on Spatial Memory. *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization,* 2001.

[16] TXL Home Page. April 1st, 2005 <http://www.txl.ca/>.

[17] C. Ware. Information Visualization: Perception for Design. San Francisco, CA. *Morgan Kauffman,* 2000.

[18] J. Wu and R.C. Holt and A.E. Hassan. Exploring Software Evolution Using Spectrographs, *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, 2004.