Software in the City: Visual Guidance Through Large Scale Software Projects

Marc Schreiber
FH Aachen
University of Applied Sciences
marc.schreiber@fh-aachen.de

Stefan Hirtbach
Microsoft
Advanced Technology Labs Europe
stefah@microsoft.com

Bodo Kraft
FH Aachen
University of Applied Sciences
kraft@fh-aachen.de

Andreas Steinmetzler
Microsoft
Office:mac
andreast@microsoft.com

Abstract: The size of software projects at Microsoft are constantly increasing. This leads to the problem that developers and managers at Microsoft have trouble to comprehend and overview their own projects in detail. Regarding that there are some research projects at Microsoft with the goal to facilitate analyses on software projects. Those projects provide databases with metadata of the development process which developers, managers, and researchers can use. As an example, the data can be used for recommendation systems and bug analyses.

In the research field of visualization software there are a lot of approaches that try to visualize large software projects. One approach which seems to reach that goal is the visualization of software with real life metaphors. This paper combines existing research projects at Microsoft with a visualization approach that uses the city metaphor. The goal is to guide managers and developers in their day to day development decisions and to improve the comprehension of their software projects.

1 Introduction

Microsoft's software projects are generating a vast amount of data during their development process. Their Source Code Control Systems (SCCSs) alone are several terabyte large, in addition information is stored in bug repositories, drop shares, etc. With respect to that it is nearly impossible to keep a clear overview of the software project in various ways.

The pure size of the SCCSs makes it difficult for the single developer to understand the coherence of the source files, classes, functions, and other software components. Another example is that it is hard to tell which components (libraries and executables) rely on each other due to countless number of components inside of each Microsoft project. Those projects contain legacy code as well, which makes it harder to understand the code base, especially when the original author of legacy code left the company.

Within Microsoft Research there are several projects trying to help developers to manage the large scale projects. An example is the defect predictor of [NZZ$^+$10] which predicts code defects inside of Windows. The predictor identifies defects by certain consecutive changes on code in the SCCS history. The predictor claims to be the best with precision and recall above 90%.

Codebook is a social network based approach of [BKZ10] which helps Microsoft developers to orientate in large scale software projects. It is a platform which models the relationships among people, bugs, code, tests, builds, and specifications.

Those examples have in common that they rely on the data of the SCCS—there are more examples which also depend on other data like bugs: [AJL$^+$09], [BN12], [ZWDZ04], etc. The common set of requirements from those projects motivated the creation of a new data warehouse inside of Microsoft facilitating analyses on top of development processes. This data warehouse consists of multiple databases:

Core Database (CDB) Contains raw meta data information of SCCSs like Changelists, Filechanges, differences between files, etc.

Derived Database (DDB) Stores heuristic data which is generated on top of CDB data.

Binary Database (BDB) Saves information about binaries (types, functions, etc.).

Task Database (TDB) Includes data about workitems, bugs, and their states.

With the available metadata of the development process the Product Groups (PGs) at Microsoft started to generate visualizations of their software. These visualizations are simple and provide only hints on what is happening during the development. At first glance the simplification of the visualizations seems to confirms the hypothesis of [Bro87]; the author argues that software can not be visualized because large software projects are too complex.

But [KM99] contradicted [Bro87] and showed that *"Visualization is Possible"* if metaphors of the real life are considered. Many software visualizations are successful when city metaphors are used. Their advantages will be described briefly in Section 2 by demonstrating some existing approaches for visualizing software with the city metaphor.

Inspired by the existing visualizations, Section 3 will introduce the concepts of a new visualization tool named CodeMine City Tool (CMCT). This tool visualizes large software projects on top of the data warehouse with the goal to guide developers and managers in their day to day decisions and to make the comprehension of the software projects better.

Section 4 will show some results of visualized Microsoft projects. Following this Section 5 will describe some future prospects of the CMCT and Section 6 will give a conclusion talking about lessons learned and takeaways for developers.

2 State of the Art

In the research field of software visualization the consideration of the city metaphor claims to be successful—examples are the work of [KM00], [BNDL04], [WL07], and [SL10].

They are all using metaphors of the real life; [KM00], [WL07], and [SL10] use the city metaphor and [BNDL04] uses the landscape metaphor. Their visualizations are using similar concepts, for example, they are all using buildings to represent elements of software.

In [KM00] the authors developed a tool called *Software World* which visualizes Java code with the city metaphor. The tool was a result of their previous paper [KM99] where they showed that the visualization of software can be meaningful even when the size of software projects prevents a two-dimensional visualization (graphs, UML, etc).

Software World displays a whole software system with multiple cities where each city represents one source file. The cities are subdivided into districts representing classes of the source files and the buildings in the districts visualize the methods of the classes. The building height is defined by the number of source code lines and the building color indicates if the access is public or private.

The paper [WL07] introduced a similar concept to [KM00] for Java software visualization: A city represents a whole Java project, the districts represent Java packages, and the buildings represent Java classes. The dimensions of the buildings (area, height) are determined by the number of attributes respectively number of methods of each class.

An innovation of [WL07] is the introduction of the district topology which displays the Java package hierarchy. Each district is located on a level according to its nesting in the package hierarchy. A subdistrict of a district is placed one level higher. This topology creates a hilly landscape which helps to realize the package structure.

The authors of [BNDL04] focused on the structure of software projects. In contrast to [KM00] and [WL07] buildings are used to symbolize the existence of attributes and methods distinct by different shapes. Another difference is that no districts are used to group classes. Instead the authors used nested spheres to symbolize the package structure. So inside of a sphere there are other spheres (the subpackages of a package) or cities where each city represents one class and the buildings inside the attributes and methods of that class.

An important aspect of [BNDL04] is to display the dependencies among attributes and/or methods. Rather then using straight lines for direct connections, which could cause occlusions and overlappings, the authors propose the *Hierarchical Net*. This solution routes the connections between depending attributes/methods according to the hierarchy levels of software elements.

[SL10][1] uses the city metaphor as well by focusing on visualizing the development history. Like the other introduced papers [SL10] takes advantage of the hierarchical structure of software represented by a hierarchical system of streets. For visualizing the history of software they use a layout algorithm which places components along the streets but keeps their initial positions: New components are attached to the end of a street, growing components cause to shift other elements, and the space of removed component stay empty.

A second approach for visualizing the history is the introduction of a topology to the system. An important aspect of the development history is the age of components which is represented by the level of each component. *"The older an element is, the more its*

[1] [LS09] presents the approach as well.

representation will be elevated in the visualization." [2] For a better age comparison of different heights in the topology they also used contour lines. This concept makes it is for the user to tell how old components of the software are.

3 Concept of CodeMine City

The basic idea of the CMCT is to provide managers and developers of software projects with a visualization which guides them in their development decisions. Another purpose of the tool is to help managers and developers to comprehend their software better, e. g. by visualizing library dependencies.

As seen in Section 2 three dimensional visualizations which use a metaphor of the real world—especially city metaphors—can help to gather the vast amount of data of software projects. So to guide users of CMCT in their decisions and to help them comprehending their software better the tool will visualize data in a city metaphor, like in [WL07, page 2]. But the tool will have three main differences:

(A) Instead of visualizing classes as buildings the CMCT will visualize binaries as buildings.

(B) The CMCT will characterize its cityspace by the dependency graph of the binaries.

(C) The user can decide which metrics will determine the dimensions of the buildings like area or height.

The reason for visualizing binaries as buildings (point A) is the scale of projects in Microsoft. If classes would from the basis for buildings the cityspace would explode. The number of buildings in this approach would cause quite a bit of confusion.

	Small scale project	Large scale project
Binaries	6×10^1	3.6×10^2
Source Files	5.2×10^2	5.7×10^4
Classes	8.2×10^2	9.6×10^4

Table 1: Compare Buildings Count of Microsoft Products on the Basis of Binaries, Source Files, and Classes

As Table 1 shows the amount of buildings for a large scale project would be about 160 times bigger if the cityspace would be based on the source files. Even in a small scale project the city would be about ten times bigger. If the buildings were based on classes, it would be even worse. This indicates that the traditional approaches as seen in Section 2 will lead to confusing cityspaces.

Those approaches could also not be applied if a project is based on C code alone because the buildings must be based on functions. But what would happen when a project is based

[2] Source: [SL10]

on C and C++? Those arguments underline the decision for point A because it is generic approach for software projects in general.

The main argument for point B is that the user can test applied changes to a binary to understand the affect on other binaries in the system. This point is very important in large scale software projects because inter binary dependencies are often non-obvious. For example: A new developer in a team does not know the structure of the software project. If he has to fix bugs he can determine with the tool if that fix could harm other binaries and he can ask his team mates for advice.

A second reason for point B is that on the basis of the cityspace it is also possible to identify the role of a binary: Binaries at the center (core binaries) of the city are referenced more often within the software project, whereas peripheral binaries are referenced less because they make use of the core binaries.

The visualizations in [WL07] and [KM00] do not let the user choose between different metrics[3]. That is a disadvantage because decisions are rarely based on a single metric like lines of code which supports point C. The CMCT allows to select different metrics[4] which makes it possible to detect binaries with the help of their shape or color. Section 3.2 will explain the details of that argument why point C is important and how shape and color can help.

3.1 Setting of CMCT Cityspace

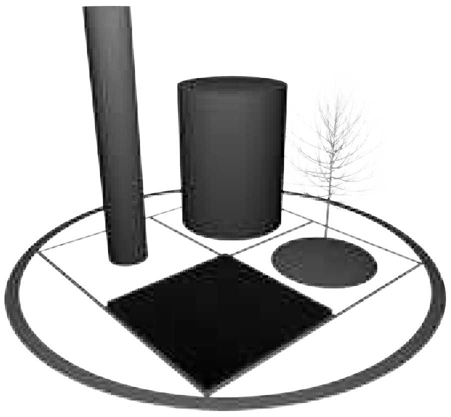
Figure 1: Elements of CMCT

The cityspace of the CMCT consists of four different elements (see Figure 1). As seen at the beginning of Section 3 binaries will be visualized by buildings and the dimensions are determined by metrics which the user is able to choose. If no metrics could be applied to a binary, because no source files are available for the binary (external binary), then the binary will be symbolized by a tree. This makes it easy for the user to distinguish between internal and external binaries.

The binaries are organized in districts surrounded by a roundabout traffic and each district contains the binaries of one directory. In the district the buildings are arranged on a grid structure. If a districts contains n buildings and n is not a square number, then some carets are empty. From the bird's eye view it is hard to differentiate if a caret is really empty or if it contains a building with a very small area. Meadows filling empty spaces and help to differentiate these cases.

[3]In [BNDL04] there is not even one metric displayed.

[4]The metrics work the data warehouse which mean that they are using data from CDB, DDB, BDB, and TDB.

One concept of the CMCT is to visualize the dependencies among the districts. This is supported by streets connecting dependent districts with each other. Single binary dependencies can be resolved through the selection feature of CMCT: The selected binary will be highlighted with blue color, all binaries which reference the selected one will be highlighted with orange color, and all binaries which are referenced by the selected one will be highlighted with green color. In addition the CMCT will show the dependency route with orange or green lines on the connecting streets.

The arrangement of the cityspace or more precisely of the districts happens through two algorithms: At first the districts are arranged by the algorithm described in [KK89] and afterwards overlappings of the districts are removed with the algorithm of [DMS05].

A property of the algorithm in [KK89] is that it produces a relative small number of edge and node crossings. However crossings are not eliminated and because of this fact the CMCT has to avoid that districts are crossed by streets[5]—streets can be crossed by other streets—which is supported by tunnels passing underneath the districts.

The tunnels help the user to identify the dependencies in an easier way. If the crossing of district would be avoided by the usage of the roundabout traffic (crossing streets enters the roundabout traffic on one side and leaves it on the opposite side), it could create the assumption that two districts depend on each other even if they do not.

3.2 User Stories

Clean and Organize Code in Projects Over time the source code of projects grows and grows. Often code will be refactored or functions become obsolete resulting in dead code. To help developers cleaning up the code base the CMCT can display a texture with cracks for each building. The more cracks are visible the more dead code the building contains.

The CMCT can also help to organize libraries. If there are a lot of functions which are only called by one or a few other libraries, it makes no sense to provide those functions in the core library. A better solution is to make those functions part of a new library or to move them into existing ones. Therefor the CMCT can display, with a pie chart texture, how much code is used by none, few, or many other binaries.

What Depends on a Library/Source File? A new developer at the team was told to implement a new feature. For that he has to work on code which he has never touched and so far he does not know which other binaries could be impacted by his changes.

With the CMCT he can search for the file which he will modify and the CMCT will directly show all binaries depending on it. If no dependencies exists, he can go on implementing the feature. Otherwise he can investigate who is working on dependent binaries and coordinate the work

[5]Crossing of districts would cause occlusions and overlappings, see [BNDL04].

What Files Are Easy To Edit and Which Are Not According to [SZZ05] it is possible to calculate how difficult it is to edit source files in a software project. This metric is interesting for managers of software projects because they can assign work to rookies or the experienced developers with respect to the difficulty of source files.

Combination of Metrics A wide variety of combinations of metrics for the different dimensions are conceivable. Here is one example: For the building area the user can choose the number of code lines and for the building height the number of methods. Buildings which appear as flat slices indicate that there is are lot of long methods in it—it is bad smell, see [Fow99].

3.3 Status of Implementation

The basic concepts of the CMCT are fully implemented and the tool is ready to use. The tool is configurable so that it can be used for all Microsoft projects whose data is available in the data warehouse. The CMCT provides a set of metrics to the user and the implementation is open to be extended with more metrics. The CMCT is implemented with C# and the rendering of the cityspace is done by WPF 3D (see [Pet07] as reference).

The CMCT is an internal project and will not be available outside of Microsoft. Part of the reason for this is its dependency on the internal data warehouse (CDB, DDB, etc.) tailored to Microsoft specific requirements.

4 Cityspaces of Microsoft Projects

Finally this section shows some cityspaces of software projects in Microsoft. The first example in Figure 2 is a large scale project showing a structured cityspace. In spite of the large number of binaries the cityspace is well-arranged. (1) The core district, on which a lot of binaries of the software project depend, is located at the center. Furthermore there is also a second core district near the center of the city. This district contains only one external binary on which almost every binary depends.

The districts which surround the core districts depend on them. (2) At the edge of the city there are some smaller groups of districts with interdependencies. All of those districts are co-located.

The highlighting of the dependencies works very well too. Through the selection it is possible to detect interrelated buildings immediately. Figure 2 also illustrates that the highlighting confirms the role of the district at the center as a core district.

All trees and meadows contribute to a good overview for the user, even from a far distance. By looking at Figure 2 it is very clear which carets of the districts are empty and which binaries are external to the project. It is also clear that most binaries are implemented by the project itself given the lack of trees in the city.

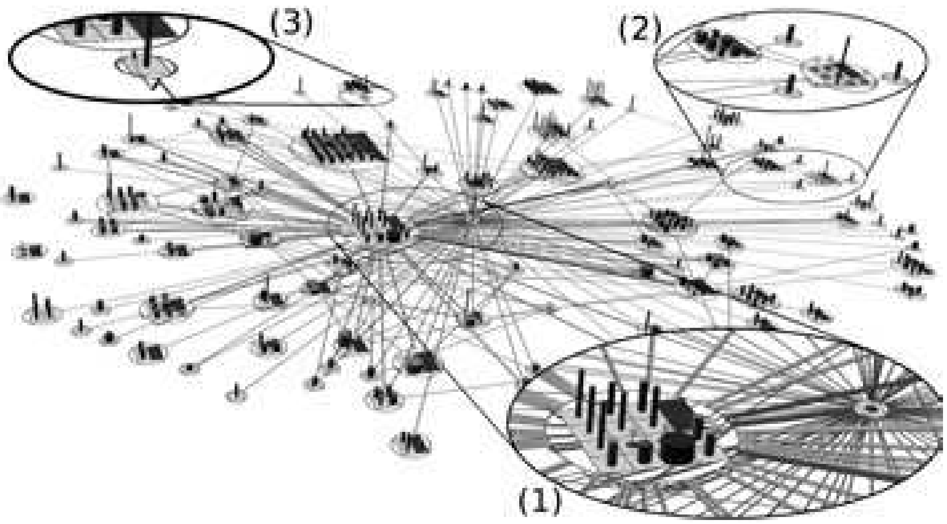

Figure 2: Cityspace of Large Scale Project

(3) From the distance it also possible to see if a district is passed by a tunnel. This and the fact that the route of a dependency between two districts never turns off the road makes it easy for the user to recognize if two districts depends on each other.

Figure 3 presents the cityspace of a small scale project. The cityspace of the project is well-arranged and it has one core district. There is a second core district but it is only referenced by the other core district, so they are forming the city center—the second core district looks like a park because it contains only external binaries and an open area.

The last example shows a cityspace of a medium scale project of Microsoft (see Figure 4). The city contains only of a few districts, where two of them contain 87% of all buildings. The remaining districts contain only a few buildings. Figure 4 shows that a lot of the binaries are external binaries symbolized by trees—an ecologically beneficial city.

The large district at the center contains 71 binaries. Its location and the amount of buildings in it create the impression that this district is the city center, but appearances are deceiving. The district contains an external binary on which almost all internal binaries depend. There is only one binary referenced outside of the district. The real city center is the district on left side of the big district. But that situation was quickly revealed with a few clicks and shows the power of CMCT.

5 Future Prospects

There are some aspects of the CMCT which can be improved. For example: As Figure 2 shows there are a lot of streets which connect districts at the edge with one district at the center. Reducing the number of streets would lead to a more clear cityspace. This could

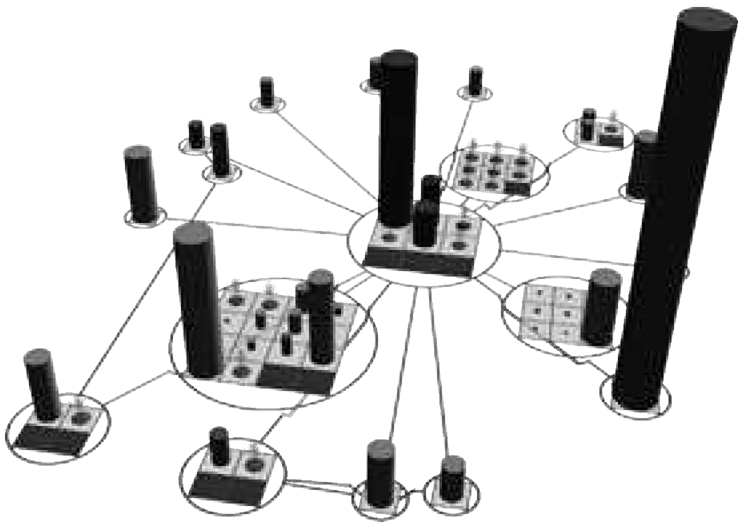

Figure 3: Cityspace of Small Scale Project

be reached with highways. The streets starting in the same area and ending at the same district would form a highway and this highway connects the districts.

A feature for comparing branches of software projects could be implemented for the CMCT. Two or more cities would be displayed side by side (maybe separated by a river) so that the user is able to compare the activities among multiple branches. Code integration from one branch to another could be symbolized with traffic.

6 Conclusion

As this paper shows—and especially Section 4—the conceived concept elements of Section 3 are helpful to comprehend software projects better. It is easy to spot dependencies by the CMCT and it provides a well-arranged cityspace to the user independent of the project size.

During the development of the CMCT we discovered some oddities in the cityspace of the data warehouse project—the cityspace did not match the expected cityspace. By inspecting why the cityspace did not look like expected we discovered that some build files of the project were configured incorrectly. This shows again the power of the CMCT. Developers who have a good knowledge of a software project are able to detect issues they were not aware of.

Finally we summarize the lessons learned from our project: The approach of using the binaries as basic element for visualizing the structure of large scale software projects is

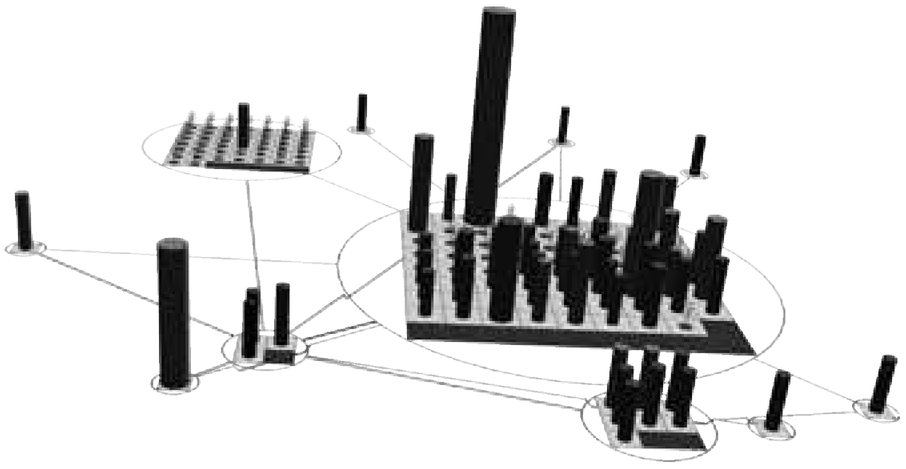

Figure 4: Cityspace of Medium Scale Project

appropriate. Especially for large scale projects this approach reduces the number of visual elements in the city space (see Table 1).

Another advantage of this approach is the independence of a specific technology like in the approaches of [WL07], [SL10], or [KM00] which are limited to Java applications. Inside Microsoft software projects depend on many different technologies (e. g. C, C++, .NET Framework, scripts languages, etc.). For these heterogeneous projects it is hard to find a common element for the visualization (like classes or functions) and if such an element exists for one software project, it is not guaranteed to be the common element in other projects as well.

A major takeaway of the CMCT for software engineers is that the CMCT creates a mental image—[SL10, page 1] shows that a mental image of software is important for the communication among developers. Such images help software engineers to improve their communication inside of software projects. For example, the CMCT was used to explain the structure of the small scale project (see Figure 3) to new developers in the team. Immediately new developers knew the coherence of the single modules and they were able to put their work into the context of the projects architecture.

The CMCT makes refactoring hints available to the engineers as well. As seen in Section 3.2 combinations of different metrics provide those hints (e. g. bad smells). Other combination could point to *AntiPatterns* (see [BMSMM98])—*The Blob* could be identified by a combination of lines of code and number of attributes. It is also possible to extends the collection of metrics with new ones which would forebode to other *AntiPatterns* as well. Addressing other specific software engineering problems with new metrics is possible as well.

Considering the user story "*Clean and Organize Code in Projects*" software engineers are able to evaluate and comprehend the architecture of their projects. How important comprehension of software is showed a Principal Development Lead at Microsoft by approving the value of CMCT: *"There's value in enabling faster code understanding."* In his first job at Microsoft he had to understand 200k lines of code. The CMCT could have helped him to understand all the code in a faster way.

Acknowledgments

We would like to acknowledge to the following persons who have made the completion of this paper possible:

The team (Rich Hanbidge, Olivier Nano) we worked with at the Microsoft Advanced Technology Labs Europe in Aachen, for the support during our research and for making it possible to use the development data of some Microsoft PGs.

Jacek Czerwonka, for providing us with an early development version of the BDB which helped us to develop the CMCT.

References

[AJL$^+$09] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 373 – 382. ACM, 2009.

[BKZ10] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125 – 134. ACM, 2010.

[BMSMM98] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1 edition, April 1998.

[BN12] Christian Bird and Nachiappan Nagappan. Who? What? Where? Examining Distributed Development in Two Large Open Source Projects. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 237 – 246, 2012.

[BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems. In *Proceedings of the Joint Eurographics – IEEE TCVG Symposium on Visualization*, pages 261 – 266, 2004.

[Bro87] Frederick P. Brooks, Jr. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10 – 19, April 1987.

[DMS05] Tim Dwyer, Kim Marriott, and Peter J. Stuckey. Fast Node Overlap Removal. In *Graph Drawing*, pages 153 – 164. Springer, 2005.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[KK89] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Inf. Process. Lett.*, 31(1):7 – 15, April 1989.

[KM99] Claire Knight and Malcolm Munroe. Visualizing Software - A Key Research Area. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, Washington, DC, USA, 1999. IEEE Computer Society.

[KM00] Claire Knight and Malcolm Munro. Virtual but Visible Software. In *Proceedings of the International Conference on Information Visualisation*, pages 198 – 205, Washington, DC, USA, 2000. IEEE Computer Society.

[LS09] Claus Lewerentz and Frank Steinbrückner. SoftUrbs: Visualizing Software Systems as Urban Structures. Technical report, BTU Cottbus, February 2009.

[NZZ$^+$10] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change Bursts as Defect Predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, pages 309 – 318, November 2010.

[Pet07] Charles Petzold. *3D Programming For Windows*. Microsoft Press, Redmond, WA, USA, 2007.

[SL10] Frank Steinbrückner and Claus Lewerentz. Representing Development History in Software Cities. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 193–202, 2010.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. HATARI: Raising Risk Awareness (Research Demonstration). In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107 – 110. ACM, September 2005.

[WL07] Richard Wettel and Michele Lanza, editors. *Visualizing Software Systems as Cities*, 2007.

[ZWDZ04] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563 – 572. IEEE Computer Society, May 2004.