Understanding API Usage to Support Informed Decision Making in Software Maintenance

Veronika Bauer Lars Heinemann
Technische Universität München, Germany
{*bauerv, heineman*}*@in.tum.de*

***Abstract*—Reuse of third-party libraries promises significant productivity improvements in software development. However, dependencies on external libraries and their APIs also introduce risks to a project and impact strategic decisions during development and maintenance. Informed decision making therefore requires a thorough understanding of the extent and nature of dependencies on external APIs. As realistically sized applications are often heavily entangled with various external APIs, gaining this understanding is infeasible with manual inspections only. To address this, we present an automated approach to analyze the dependencies of software projects on external APIs. The approach is supported by a static analysis tool featuring a visualization of the analysis results. We evaluate the approach as well as the tooling on multiple open source Java systems.**

***Keywords*-software reuse, API, library, software maintenance**

I. Introduction

Many software projects depend on external libraries and frameworks to deliver feature-rich software in a cost-efficient and timely manner. These libraries represent a great opportunity for productivity improvement and also can improve overall code quality [1], [2]. However, relying on third-party APIs entails loosing control over a part of a software system's functionality. This introduces potential risks for software maintenance: Firstly, APIs keep evolving, often introducing new functionality or providing bug fixes. Migrating to the new version is therefore often desirable. However, depending on the amount of changes, e.g., in case of a major new release of an API, backward-compatibility might not be guaranteed. Secondly, an external API might not yet be completely mature. Thus, it could introduce bugs into the current software project, which might be difficult to find and hard to fix. In this case, it would be necessary to replace the current API with a more reliable one, as soon as it is available. Thirdly, the provider of an API might decide to discontinue its support, such that maintainers can no longer rely on them for new functionality and bug fixes. Finally, the license of a library or a project might change, making it impossible to continue the use of a particular API. This, for instance, is the case when commercializing a software product built on non-commercial libraries. These risks are beyond the control of the maintainers of a system but need to be considered to make informed decisions about maintenance options of a software system. Consequently, it is necessary to understand the complexity of the dependencies to external APIs in detail. Without this knowledge, the effort required for many maintenance scenarios is hard to estimate. However, for realistically sized software systems, it is not feasible to assess API dependencies manually. Tool support is therefore needed to provide this information automatically.

Problem: Software development with external libraries poses multiple challenges to software maintenance. Maintainers need to address these challenges when planning and deciding maintenance activities for a system. This requires a detailed understanding of the API dependencies and their complexity. However, it is infeasible to manually retrieve this information for large projects. To effectively maintain projects with external libraries we, therefore, need dedicated support that allows for informed decisions regarding migration of libraries.

Contribution: We present an approach that automatically extracts information about library usage from the source code of a project and visualizes it to support decision making during software maintenance. It enables quick insight into how external libraries are used by a project and how complex the dependencies are. It thus aids in decision making regarding library migration scenarios. We identify multiple use cases and evaluate the approach for a number of open source Java systems.

Outline: The remainder of this paper is structured as follows: Section II describes the use cases, whereas Section III introduces related work and assesses its applicability with respect to the use cases. Section IV presents our approach, whilst Section V details the results of the evaluation. Sections VI and VII discuss outcomes and threats to validity. The paper proposes further research steps in Section VIII and concludes with Section IX.

II. Use cases

In the following paragraphs, we detail the use cases based on which we will evaluate our approach. We consider them to be typical maintenance scenarios.

A. API Evolution

A new release of an external library requires maintainers to assess whether the new release is relevant for their system,

i.e., contains desired new functionality or bug fixes for the parts of the API the system uses. If so, they need to decide if and when to migrate the system to the new version of the API. To allow a detailed estimation of the migration effort, maintainers need information about which parts of the system are currently using the old version of the API as well as how complex the dependencies are.

B. API Replacement

At times, a library used by a project might have to be replaced by a completely different one, for instance, when a library is no longer supported. Maintainers of the project need to be able to determine the trade-off between staying with a discontinued library and migrating to a different library. Replacement of an API also could become necessary due to license conflicts: a library licensed under the GPL[1] might be convenient to use as long as the project is non-commercial. If, however, at some point the project should be turned into a commercial one or license policies of API producers change, the concerned libraries need to be exchanged. In both scenarios it is important for maintainers to know which parts of their projects depend on an API to which degree. They also need to know which functionality is used to be able to select a matching replacement, as well as to estimate the dimension of the migration.

C. System Integration

System integration refers to a maintenance scenario in which two software systems need to be integrated that are using different libraries for the same functionality, e.g. one of them uses Swing as GUI framework, the other one SWT. No constraints are given concerning which library should be used in the integrated system, however, at least one of the two systems needs to be migrated to the other library (it could also be the case that both systems are migrated to a third library). From a strategical point of view, an estimation about the effort of the integration is required to schedule the migration. Furthermore, maintainers need to decide on the API to migrate to, based on the complexity of the dependencies.

III. Related Work

Understanding the complexity of API dependencies involves combining approaches from two fields of research, impact analysis and library usage analysis. When building the tool support and visualization, we also dealt with issues of software visualization. In the following, we introduce related work for these areas.

A. Impact Analysis

The goal of impact analysis is to determine which artifacts are affected by a proposed change to the software. This information can then be used to plan and implement the changes more accurately [3] as well as estimating and directing test efforts [4]. Sangal et al. [5] employ dependency structure matrices (DSMs) to manage complex software architectures. A matrix visualizes the dependencies between the modules of a system (as given by the Java package structure), which can be used for impact analysis of proposed changes to certain system modules. The cells of a DSM contain a number that indicates the "strength" of a dependency, given by the number of references from one module to another. However, this does not yet provide a measurement for the complexity of the dependencies which we consider necessary in our use cases.

B. Library Usage Analysis

Lämmel et al. [6] analyze the API usage of 1,476 open source Java projects. Among others, they investigate the *footprint of API usage* on a project level, given by the number of APIs used and the number of (distinct) API methods called. They suggest that these numbers can be used as an indicator for *API-related complexity*. Their work provides a general overview on API-related complexity on a project level but does not foster the detailed understanding of dependencies, which we require to support maintainers of a system in the mentioned use cases. Acquiring this detailed information necessitates a very detailed report on the actual API-usage of a project, drilling down to package and class level.

Mileva et al. [7] analyze the usage frequency of API entities over time. If usage of an API element is decreasing, they infer that the item might be problematic, *e. g.*, exhibits a defect. Their goal is to utilize the *wisdom of the crowds* to produce recommendations telling that projects are increasingly avoiding a particular API element and proposing alternative ones. This could consequently trigger improvements to the user's code base. The authors thus attempt to provide support for API evolution scenarios. However, their goal is to detect and remove problematic API elements, whereas we focus on assessing the complexity of API dependencies and, therefore, need additional dependeny information to support our use cases.

Raemaekers et al. [9] analyze library usage in open source systems with the goal of automatically discovering the risks entailed by the use of third-party libraries. Among others, they determine how the usage of a library distributes over the system. If the usage is concentrated in a single component, the risks introduced by a library are more isolated compared to the situation where library references are scattered over the whole system. The authors, therefore, consider the distribution of API dependencies but do not assess their complexity in terms of the functionality used. The scope of the analysis is restricted to the package level on the basis of package imports, whereas our use cases require more detailed notions of API references to resolve dependencies in a more fine-granular way.

[1] The GNU General Public License, http://www.gnu.org/licenses/gpl.html

C. Software Visualization

Software visualization, as a form of information visualization, has the goal of transforming complex abstract information into a visual form, in order to support users in better understanding complex phenomena [10], [11]. Consequently, it is used for *intelligence amplification* [12], helping people that work with information to reason and communicate about it. Diehl defines software visualization as "the visualization of artifacts related to software and its development process" which is employed with the purpose to represent "the structure, behaviour, and evolution of software" [11]. Structural visualizations focus on information which can be extracted from a software system without running it whilst behavioral visualizations contain data extracted from program executions, like run-time traces. Price et al. [13] further differentiate between algorithmic and program visualization.

Our approach presents a structural program visualization, displaying the hierarchical composition of the software project in an interactive table view. The visual representation of the analysis results was inspired by the extended SeeSoft View [14] which encodes different characteristics of source code, such as code age, into colored bars.

IV. Approach

Our approach automatically analyzes software systems with the goal of determining the degree of dependence to its included libraries. We statically analyze the code to determine the dependencies and use the extracted information to produce a visualization to gain a quick overview of the library dependencies. To determine the degree of API dependence and complexity, we measure three values: firstly, we determine the total number of all method calls to external APIs in order to find out which libraries are contributing to a project and to which extent. This allows to rank APIs with respect to their contribution. To take into account the variety of API functionality which the project actually employs, we secondly extract for each external API the number of distinct method calls. Thirdly, our visualization encodes the proportion of each distinct method call with respect to all method calls.

A. Implementation

The approach is implemented in Java on top of the open source software quality assessment toolkit ConQAT[2], a modular toolkit for creating quality dashboards which integrate the results of multiple quality analyses. The current implementation analyzes the API usage of Java programs, however, it could be transferred to other programming languages. The approach consists of two principal parts: fact extraction and visualization. Both are presented in detail in the following sections.

[2]http://www.conqat.org/

B. Fact Extraction

We use the Eclipse Java Compiler (ECJ) to obtain the abstract syntax tree (AST) for Java code. We traverse the AST to extract the API references of the source code of software projects. We approximate the degree of dependence to an API with the number of API method calls. For each class, we determine the total number of API calls, the number of distinct API methods called for each included library as well as their proportion and aggregate the data hierarchically along the package structure. We consider every Java Archive File (JAR) contained in the project as included library.

C. Visualization

We extend the HTML output and tree table visualization capabilities of ConQAT to present the results of the fact extraction. Figure 1 shows an exemplary screenshot of an analysis output. The columns, labelled *A*, list all external APIs, to which the project has dependencies. They are ordered decreasingly by the number of overall API calls from left to right. The table rows, labelled *B*, contain an interactive tree that reflects the package structure of the analyzed system. The table cells show the (aggregated) total number of method calls, *#total*, from a system package to a certain API, as well as the number of distinct method calls, *#dist*. The width of the colored bars visualizes the total number of API calls, *#total*. Each color corresponds to a distinct API method and the width of the colored stripe (*PDist*) encodes proportionally how often it was called compared to the other API methods.

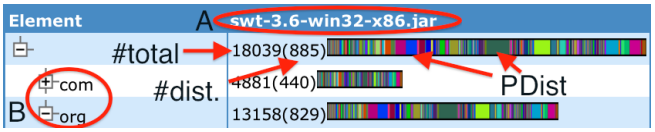

Figure 1. Excerpt of the API-Dependence Visualization, displaying the column of one external library, labelled A, the rows with the system composition, labelled B, and the values for number of total and distinct API calls. The number of total calls and the proportion of the distinct calls are encoded in the colored bar.

The top-level row allows to gain a quick insight into the role a particular API plays within the project in terms of overall usage. Expanding the tree reveals how packages of the system use the APIs. It allows to understand to which extent a package is dependent on APIs and also how the dependencies of a certain API span over the system architecture. The table also allows to compare the degree of API dependence between packages. Furthermore, the tree enables a drill-down to the class level of the system, thus allowing to pinpoint the API dependencies.

V. Evaluation

In the following, we qualitatively evaluate our approach by answering questions typically raised in the use case scenarios of Section II. We illustrate the findings on three study objects, which are listed in Table I. All of them are open source Java projects, which use external libraries. The extent to which these projects depend on external libraries, however, differs notably in terms of complexity.

Table I
The studied Java applications

System	Version	Application Domain	LOC
Azureus/Vuze	4504	P2P File Sharing	786,865
DrJava	20100913-r5387	Java Programming	160,256
OpenProj	1.4	Project Management	151,910

Our approach answers the following questions:

Which external libraries does the system depend on?: This question is relevant for the use case *API Evolution*, as deciding if one is concerned by a new API release requires knowing all of a projects' external libraries.

The Figures 2, 3, and 4 show the visualization of the analysis results for OpenProject, Azureus and DrJava. All external libraries which are used by the systems are listed in the top row of the corresponding Figure[3].

Which libraries are used most by the system? Is there a package depending on a large number of external libraries?: This information helps to estimate the criticality of a migration, concerning the use cases *API Evolution* and *API Replacement*. If a central API is concerned, migration might be more urgent than for a hardly used API.

The external libraries are ordered decreasingly according to their share of contribution to the system. Therefore, at one glance, the most important libraries can be determined for each of them (forms.jar for OpenProject, swt.jar for Azureus, and plt.jar for DrJava). Also patterns of reuse can be seen. In Figure 4 most of the dependencies are bundled in the package "edu.rice.cs.drjava.model", which indicates the package as important for maintenance scenarios.

How many calls to a specific external library are present in the system? How complex are the dependencies?: This question is central to all use cases as it provides information about the complexity of the dependence on a specific library.

For each of the external libraries, the number of total and distinct references is reported in the matching column in the top row, which corresponds to the top level of the system representation. Putting the numbers of total and distinct references in relation allows an estimate of the complexity of the dependencies. Take as example Figure 2: forms.jar, which is most frequently referenced in the project (942 API references), accounts for only 31 distinct references, whereas jasperreports.jar (298 total references) accounts for 115 distinct references. The colored stripes encode the proportion of the distinct method calls. This allows to assess one more aspect of the dependency complexity.

How are API calls distributed over a system? Which packages are affected? Who needs to be notified of the API migration?: This question applies to all use cases: together with the question above, its results help to estimate the dimension of a migration. After deciding to migrate a project to a new version or a different API, maintainers need to determine whom to notify of the changes. They need to find out which parts of the system are affected by a migration, as well as finding the programmers responsible for the respective packages.

Expanding the interactive tree allows to immediately spot which packages or classes are dependent on a library. In Figure 3, expanding the root node of the tree shows that about a quarter of the references to the swt.jar are in the package "com" and three quarters in "org". Expanding "org" to investigate the further distribution of the API references shows that only two packages, "org.eclipse" and "org.gudy", are including SWT-functionality. The significant share of dependencies falls within the "org.gudy" packages.

[3]Note that Figure 2 is split in two rows to retain readability.

VI. Discussion

In many projects, third-party libraries are an integral part of software development. A number of maintenance scenarios caused by evolving APIs require detailed knowledge about the extent and complexity of the dependencies to these APIs. Our approach addresses this by analyzing and visualizing the extent and complexity of the library dependencies in a software project. The evaluation shows that the central questions raised during the identified usage scenarios can be answered by our approach. Since the approach is completely automated, gaining this knowledge comes at little cost. It is thus an effective way to support informed decisions during software maintenance.

Our approach aims at getting a quick overview over the library dependences of a software project. We believe that the proposed approach complements existing work on *impact analysis*, where the focus is to narrow down the parts of a system that are affected by a given change as precise as possible, for instance to direct testing efforts. Our goal is a more abstract and aggregated view and our analysis is specifically designed to focus on the dependencies to external libraries and their influence on maintenance activities.

We envision the presented analysis and visualization to be part of a quality analysis dashboard that allows to monitor the dependence status to included libraries in a continuous manner, for instance as a part of a nightly build.

VII. Threats to Validity

With our approach we intend to measure and visualize the complexity of a project's dependence to external libraries. The metrics we chose are the total number and the number

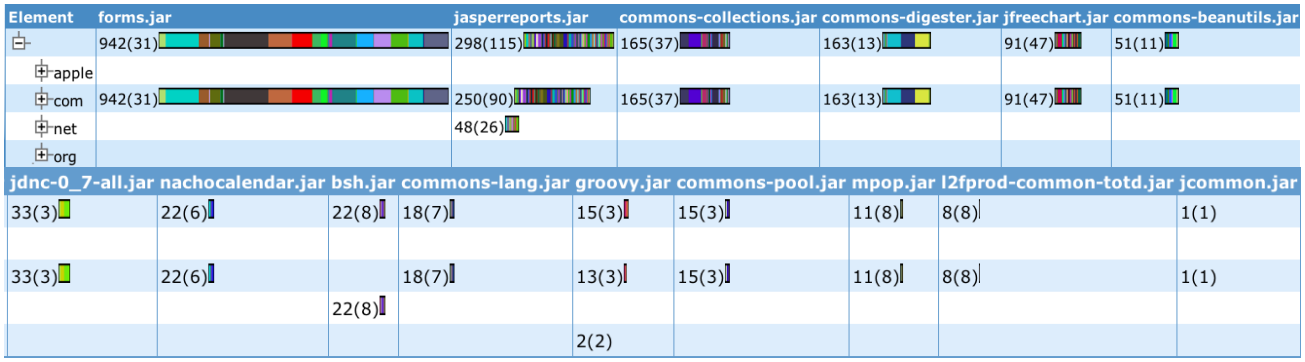

Figure 2. Dependency visualization for OpenProj, displaying the global system overview in the first row. The following rows show the API dependencies on the first package level.

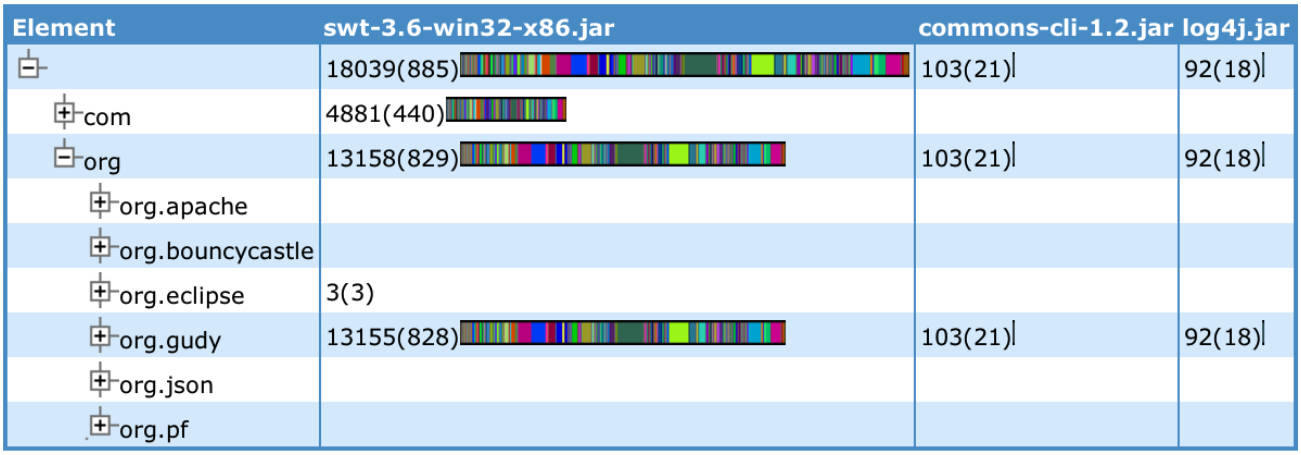

Figure 3. Dependency visualization for Azureus. This example demonstrates the localization of packages depending on SWT. Furthermore, it shows at one glance to which library the system has the strongest dependencies.

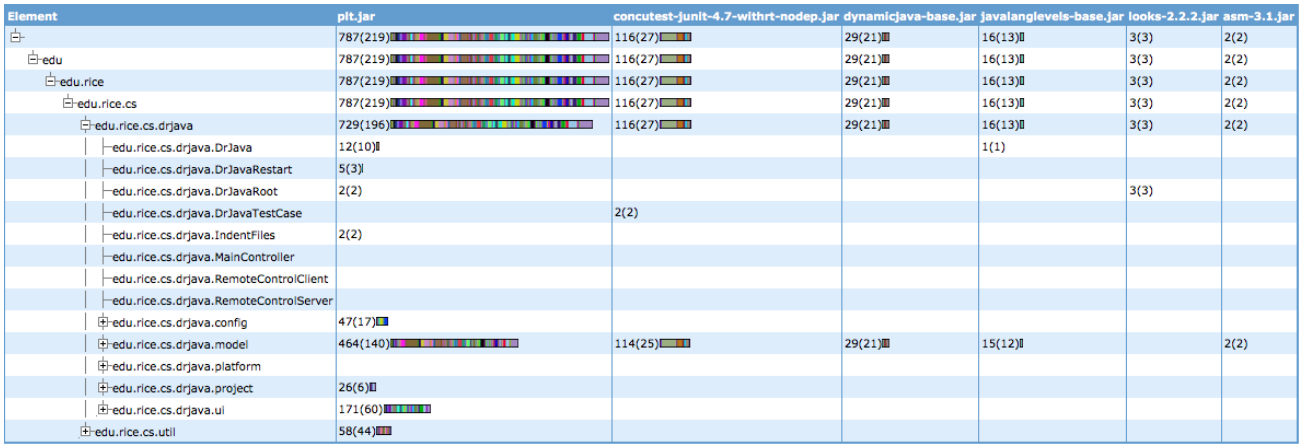

Figure 4. Dependency visualization for DrJava. This example shows the complete drill-down along the system hierarchy. Under the package edu.rice.cs.drjava, a list of classes are visible (represented by nodes which can not be expanded) with their dependencies to the library plt.

of distinct API calls. There are further static references to API elements, such as overridden types and methods as well as field references, which could have been included in the analysis. In preliminary evaluations, however, the share of these references was by several factors smaller than the one of API calls. Our notion of complexity is restricted to a syntactic level. Arguably, the complexity of the API dependencies is influenced by further factors, some of which are hard to quantify. An example is the intrinsic complexity of an API and its concepts. We also currently limit the analysis to direct static API references and did not take into account how the system transitively depends on the functionality of a library.

VIII. Future Work

To complete the drill-down on the visualization side, the next steps are to integrate the source code with the results in a way that dependencies are directly highlighted where they occur. Furthermore, we are planning to evaluate our approach more thoroughly. On one hand, we intend to perform quantitative studies in order to determine metric values and compare their occurrence over a large set of open source systems. On the other hand, we are planning a use case driven evaluation to assess the helpfulness and practicability of our approach for real-life maintenance scenarios.

We envision extending the presented approach to provide further support for API evolution scenarios. In addition to detecting the currently used version of an API, it seems useful to automatically determine whether newer versions are available. In that case, automated checks for breaking API changes could be executed to determine whether it is "safe" to switch to the new version or whether incompatibilities need to be addressed.

We are planning to employ our approach in the context of a wider spectrum of software quality analyses and risk assessment. For instance, the complexity of dependencies of a software system on external libraries could be a factor taken into account when evaluating the architecture of a system: if an external API is extensively used across different system packages, it might be necessary to revise parts of the system's architecture. Therefore, the complexity measure could be included into a quality assessment of a system's architecture. The dependency complexity could also serve as a risk indicator for a software project under development, entailing a risk classification.

IX. Conclusion

We presented an approach for the automated analysis and visualization of the API dependence for a software project. The approach allows for substantiated decisions in software maintenance scenarios, such as API migration and evolution. We introduced tool support and evaluated several use cases on open source software systems. Our findings indicate that the results of our analysis answer relevant questions in the identified usage scenarios. As the work presented in this paper is in early stages, a multitude of open research questions remain. Therefore, we outlined a number of interesting paths for future work.

Acknowledgements

We thank Florian Deissenboeck, Elmar Juergens, Birgit Penzenstadler, and Georg Wittenburg for their input and feedback on the paper.

References

[1] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[2] W. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23–30, 2002.

[3] S. Bohner and R. Arnold, "Software Change Impact Analysis." Wiley, 1996.

[4] B. G. Ryder and F. Tip, "Change Impact Analysis for Object-Oriented Programs," in *Proceedings of PASTE'01*, 2001.

[5] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *OOPSLA'05*, 2005.

[6] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC'11*, 2011.

[7] Y. Mileva, V. Dallmeier, and A. Zeller, "Mining API Popularity," in *Testing - Practice and Research Techniques*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6303, pp. 173–180.

[8] J. Quante, "Using Library Dependencies for Clustering," in *WSR'08*, 2008.

[9] S. Raemaekers, A. van Deursen, and J. Visser, "Exploring Risks in the Usage of Third-Party Libraries," Software Improvement Group, Tech. Rep., 2011.

[10] N. D. Gershon, "From perception to visualization," in *Scientific Visualization: Advances and Challenges*. Academic Press, 1994.

[11] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2002.

[12] F. P. Brooks Jr., "The computer scientist as toolsmith II," *Communications of the ACM*, vol. 39, no. 3, pp. 61–68, 1996.

[13] B. Price, R. Baecker, and I. Small, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, vol. 4, no. 3, pp. 211–266, 1993.

[14] T. Ball and S. G. Eick, "Software Visualization in the Large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, 1996.