A *Quasi*-Systematic Review on Software Visualization Approaches for Software Reuse

First round

Marcelo Schots, Renan Vasconcelos, Cláudia Werner

Systems Engineering and Computer Science Program (PESC)
Federal University of Rio de Janeiro – COPPE/UFRJ
schots@cos.ufrj.br, renanrv@cos.ufrj.br, werner@cos.ufrj.br

2014

Abstract

Reuse is present in the daily routine of software developers, yet mostly in an ad-hoc or pragmatic way. Reuse practices allow for reducing the time and effort spent on software development. However, organizations struggle in beginning and coping with a reuse program. A crucial concern for facilitating the acceptance/consciousness and adoption of reuse is how to provide appropriate reuse awareness.

Awareness mechanisms allow stakeholders to be percipient of what goes on in the development scenario, and can provide them the necessary information and support for performing their reuse-related tasks. One of the ways to increase awareness is by employing visualization resources and techniques. Software visualization has been exploited as a way to assist software development activities that involve human reasoning, helping people to deal with the large amount and variety of information by providing appropriate abstractions.

Although there are several works that aim to assist software engineering stakeholders in their day-to-day activities, little is known about the role of visualizations in supporting software reuse tasks. There are some software visualization approaches in the literature that are intended to support software reuse, but literature lacks a solid body of knowledge or a reference model of software visualizations targeted to reuse. Approaches are spread in the literature and their information is usually not clearly organized, classified and categorized. Consequently, stakeholders may not be able to properly find and choose reuse-oriented visualizations (i.e., based on their quality and concrete evidence on their actual effectiveness) for a given scenario.

This work presents a characterization study of visualizations that provide support for software reuse tasks, organized in terms of a task-oriented framework. Such framework was extended in order to capture more detailed information that may be useful for assessing the suitability of a particular visualization. Besides enabling a better organization of the findings, the use of the extended framework allows to identify aspects that lack more support, indicating opportunities for researchers on software reuse and software visualization. The results of the study were organized in a website (http://www.cos.ufrj.br/~schots/survis_reuse/), in order to allow a better exploration of the findings, as well as establish correlations between the visualization dimensions.

Summary

1. Introduction 8
1.1. Context and Problem Definition 8
1.2. Organization 10
2. Planning 10
2.1. Goal 10
2.2. Object of Study 10
2.3. Expected Results 11
2.4. Methodology 11
2.5. Research Questions 11
2.6. Search String Definition and Source Selection 12
2.6.1. Research Question Structuring 12
2.6.2. Definition of Control Publications 13
2.6.3. Source Selection 13
2.6.4. Search String Calibration 17
2.7. Procedure for Studies Selection 21
2.7.1. Inclusion Criteria (IC) 21
2.7.2. Exclusion Criteria (EC) 22
2.8. Data Extraction 22
2.8.1. Task – *why* is the visualization needed? 23
2.8.2. Audience – *who* will use the visualization? 24
2.8.3. Target – *what* is the data source to represent? 24
2.8.4. Representation – *how* to represent the data? 24
2.8.5. Medium – *where* to present the visualization? 25
2.8.6. Requirements – *which* resources are required by or used in the visualization? 25
2.8.7. Evidence – are the proposed visualizations *worthwhile*? 25
2.8.8. Data Extraction Form 26
3. Execution 28
3.1. Execution Data from the Manual Search 28
3.2. Execution Data from the Search Engine 29
3.3. Study Selection 31
4. Analysis 37

4.1. Visualization approaches supporting software reuse 38
4.2. Task – *why* is the visualization needed? 41
4.2.1. Pioneer works 41
4.2.2. Other works 42
4.2.3. Software engineering activities 45
4.2.4. Discussion 50
4.3. Audience – *who* will use the visualization? 51
4.3.1. Discussion 53
4.4. Target – *what* is the data source to represent? 53
4.4.1. Discussion 57
4.5. Representation – *how* to represent the data? 58
4.5.1. Discussion 64
4.6. Medium – *where* to present the visualization? 65
4.6.1. Discussion 66
4.7. Requirements – *which* resources are required by or used in the visualization? 66
4.7.1. Discussion 67
4.8. Evidence – are the proposed visualizations *worthwhile*? 67
4.8.1. Discussion 69
5. Final Remarks 69
5.1. Limitations 70
5.2. Open Questions and Future Works 71
Acknowledgements 72
References 72
Appendix A – Consensus information 77
Appendix B – Publication data extracted on the selection process 81
Appendix C – Visualization strategies and techniques 210

Figures

Figure 1. Reuse-based development [Kim & Stohr 1998] 8

Figure 2. Distribution of search results by subject area 20

Figure 3. Systematic review selection process 21

Figure 4. Dimensions of software visualizations (extended from [Maletic et al. 2002]) 23

Figure 5. Scopus data from study selection stages, by researcher 30

Figure 6. Word cloud indicating the 600 most frequent terms before the title reading stage 31

Figure 7. Word cloud indicating the 600 most frequent terms after the title reading stage (before the abstract reading) 32

Figure 8. Word cloud indicating the 600 most frequent terms after the abstract reading stage (before the full reading) 33

Figure 9. Word cloud indicating the 600 most frequent terms after the full reading stage (selected publications) 34

Figure 10. Distribution of selected publications per year 39

Figure 11. Distribution of selected publications per author 40

Figure 12. Approaches and supported stakeholders (*SQ2*) 51

Figure 13. Visualized items/data by approach (*SQ3*) 54

Figure 14. Evaluation scenarios (A = Academic; C = Commercial/Industrial; OS = Open Source) (*TQ7.1*) 68

Tables

Table 1. Analysis of title, abstract and keywords of control publications (occurrences of terms) 15
Table 2. Distribution of the search terms 16
Table 3. Search string definition 17
Table 4. Second round of the search string definition (filters in italic) 18
Table 5. Results from second attempt 19
Table 6. Search string in Portuguese 20
Table 7. Data extraction form 27
Table 8. Study selection data (manual search) 29
Table 9. Study selection data (search engines) 30
Table 10. Selected publications (sorted alphabetically) 35
Table 11. Visualization approaches (sorted by year) (*PQ*) 38
Table 12. Software engineering activities addressed by visualizations (*TQ1.1*) 46
Table 13. Reuse-related tasks addressed by visualizations (*TQ1.2*) 47
Table 14 . Mapping between the steps in [Kim & Stohr 1998] and the identified tasks 49
Table 15. Approaches and supported stakeholders (*SQ2*) 52
Table 16. Visualized items/data by approach (*SQ3*) 54
Table 17. Visualization metaphors employed by the approaches (*SQ4*) 58
Table 18. Data-to-visualization mapping (*TQ4.1*) 61
Table 19. Visualization strategies and techniques employed (*TQ4.2*) 62
Table 20. Publications accepted by both researchers 77
Table 21. Publications accepted only by the first researcher 78
Table 22. Publications accepted only by the second researcher 80
Table 23. Publications manually included (agreed by both researchers) 80
Table 24. Software Landscape [Mancoridis199374] 81
Table 25. Software Information Base (SIB) [Constantopoulos19951] 84
Table 26. Program Explorer [Lange1995342] 87
Table 27. Dotplot Patterns [Helfman199631] 90
Table 28. Alonso & Frakes's approach [Alonso1998483] 93
Table 29. Dy-re (Dynamic reuse) [Biddle199992] 96
Table 30. Dyno [Biddle199992 / Marshall2001 / Marshall2001103] 101
Table 31. Nested Software Self-Organising Map (NSSOM) [Ye2000266] 107

Table 32. Framework Interaction for REuse (Fire) [Marshall2001103].. 112
Table 33. Visualization Architecture for REuse (VARE) [Marshall2001103 / Anslow2004] ... 115
Table 34. Mittermeir et al.'s approach [Mittermeir200195].. 120
Table 35. Charters et al.'s approach [Charters2002765] 124
Table 36. Test Driver + SpyApp + Transformer [Marshall200381]... 127
Table 37. Spider [Anslow2004 / Marshall200435]... 131
Table 38. Claims Exploration of Relationships Visualization (CERVi) [Wahid2004414] 135
Table 39. TRAceability Pattern Environment (TRAPEd) [Kelleher200550]............................. 138
Table 40. Visualisation of Execution Traces (VET) [McGavin2006153] 140
Table 41. Growing Hierarchical Self-Organizing Map (GHSOM) [Tangsripairoj2006283] 145
Table 42. Washizaki et al.'s approach [Washizaki20061222].. 148
Table 43. DigitalAssets Discoverer [Gonçalves2007872 / Oliveira2007461]............................ 153
Table 44. Gilligan [Holmes2007100] 157
Table 45. Stollberg & Kerrigan's approach [Stollberg2007236].. 161
Table 46. BARRIO [Dietrich200891]... 165
Table 47. MUDRIK [Ali200950] 168
Table 48. Damaševičius's approach [Damaeviius2009507]... 173
Table 49. Ontology-Driven Visualization (ODV) [DeBoer200951] 178
Table 50. NFRs and Design Rationale (NDR) Ontology / Toeska/Review tool [López20091198] 181
Table 51. AMPLE Traceability Framework (ATF) [Anquetil2010427] 184
Table 52. Interface Descriptions Management System (IDMS) [Areeprayolkij2010208] 187
Table 53. FEATUREVISU [Apel2011421]... 189
Table 54. Variant Analysis [Duszynski2011303 / Duszynski201237] 193
Table 55. API-Dependence Visualization [Bauer2012435] 197
Table 56. FeatureCommander [Feigenspan20121] 200
Table 57. FlowTracker [Yazdanshenas2012143] 204
Table 58. Visualization strategies and techniques by publication/approach (*TQ4.2*)................ 210

1. Introduction

1.1. Context and Problem Definition

Software reuse is present in day-to-day software development and has been a promising paradigm in software engineering [Benedicenti et al. 1996], since it can be fully integrated and supported in software development processes, improving the life cycle by reducing time and effort needed to develop software systems. By reusing assets from past projects (i.e., that have been already tested and deployed) it is possible to provide more reliable applications and decrease maintenance efforts, since their quality is expected to reflect their previous experiences of use [Benedicenti et al. 1996] [Morisio et al. 2002].

[Kim & Stohr 1998] represent the software reuse process by dividing reuse activities into two groups: (i) *producing activities*, which involve the identification, classification and cataloging of software resources, and (ii) *consuming activities*, which involve the retrieval, understanding, modification, and integration of those resources into the software product. These groups of activities can also be classified as *development for reuse* (i.e., build generic assets, such as components, templates etc., in such a way that they can be reused in similar contexts) and *development with reuse* (i.e., the use existing assets to build [parts of the] software), respectively [Moore & Bailin 1991]. Figure 1 illustrates these activities in terms of a model of reuse-based development [Kim & Stohr 1998].

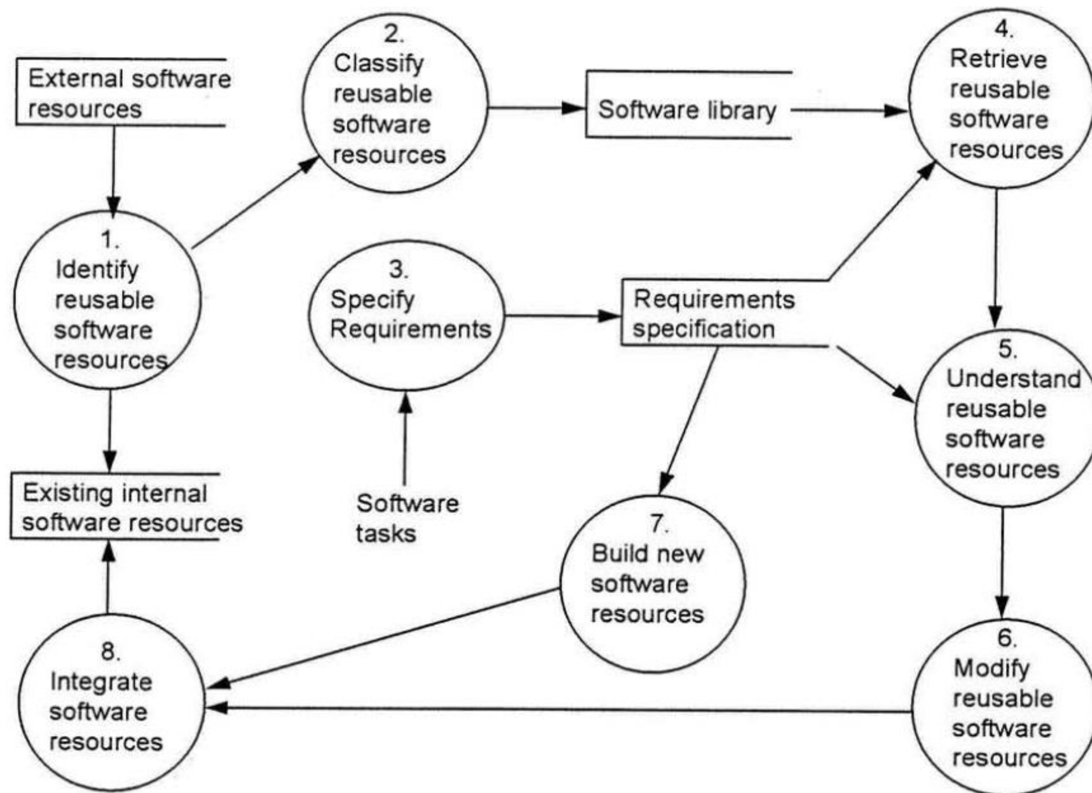


Figure 1. Reuse-based development [Kim & Stohr 1998]

According to this figure, the first step (step 1) involves analyzing existing software resources (that are developed internally or externally) in order to identify potentially reusable

artifacts (that may require some adjustments to this end), which must be then classified and cataloged (step 2) in a software library. These two steps have to be performed at the beginning of a reuse program and whenever a new software resource is acquired/developed [Kim & Stohr 1998]. Specifying requirements for the new system (step 3) has to be performed regardless of whether the software resource is to be developed from scratch or not.

Retrieving appropriate reusable software resources from the software library (step 4) is only necessary in a software reuse scenario [Kim & Stohr 1998]. After that, the next step (step 5) is to understand and assess the functionality of the selected resources in order to use or modify them. Modifying software resources (step 6) is necessary when the retrieved software resources do not exactly match the requirements specification, while building new software resources (step 7) is necessary when there are no similar software resources in the software library for some of the requirements. Finally, the last step (step 8) is the integration of both new and reusable software resources into the target software system [Kim & Stohr 1998].

Achieving effective software reuse is a difficult problem in itself, one that requires proper support in a number of facets, such as managerial aspects [Griss et al. 1994], the aid of tools [Marshall et al. 2003], and adequate mechanisms for retrieval of reusable assets[1] [Braga et al. 2006], among others. For instance, the lack of tools and techniques for effectively supporting software reuse was pointed out by Kim & Stohr (1998) and recently reaffirmed in a study with Brazilian software organizations implementing reuse practices [Schots & Werner 2013].

A crucial concern for introducing a software reuse program in an organization is the envisioning of non-technical aspects. Attempts to introduce a software reuse program may fail because of human issues, such as: (i) lack of management commitment, (ii) lack of understanding, (iii) lack of engagement of team members, (iv) absence of incentives, and (v) cognitive overload [Kim & Stohr 1998]. In order to achieve the acceptance/consciousness and successful adoption of software reuse, it is important to take into account how to better provide appropriate reuse awareness. Awareness mechanisms allow stakeholders to be percipient of what goes on in the development scenario [Hattori 2010] [Schots et al. 2012], and can provide them with the necessary information and support for performing their reuse-related tasks.

To reuse a software asset, stakeholders need to understand what it does, how it works, and how it can be reused; however, this is difficult in practice [Marshall 2001; Marshall et al. 2003]. If software engineers cannot understand assets, they will not be able to reuse them [Frakes & Fox 1996; Alonso & Frakes 2000]. In contrast, a proper understanding can help developers to decide whether and how the asset can be reused [Marshall 2001; Marshall et al. 2003]. An adequate awareness support for implementing reuse can also facilitate the acceptance/consciousness, adoption and institutionalization of a software reuse program.

One of the ways to increase awareness is by employing visualization resources and techniques. Software visualization has been exploited as a way to assist software development activities that involve human reasoning, helping people to deal with the large amount and variety of information by providing appropriate abstractions [Lanza & Marinescu 2006] [Diehl 2007]. The understanding of software is a complex activity that requires specific resources that facilitate

[1] Any item that is built for use in multiple contexts (such as software design, specification, source code, documentation, test cases, manuals, procedures etc.) can be considered as a reusable asset.

their development process [Diehl 2007]. In the software reuse scenario, visualization resources can be used for increasing awareness and comprehension of reuse elements and their surroundings.

It is known that, in general, every visualization system supports understanding of one or more aspects of a software system, and this understanding process will in turn support a particular engineering activity or task [Maletic et al. 2002], such as requirements engineering, software design, or coding. It is believed that most of these software engineering tasks can also be visually supported by software reuse. Mukherjea & Foley (1996) state that visualization is particularly important for allowing people to use perceptual reasoning (rather than cognitive reasoning) in task-solving. However, it is desirable to have an explicit description of the tasks being supported, in addition to the usual understanding goal, so that these visualizations can be more easily identified by potential users with corresponding information needs.

Although there are several works that aim to assist software engineering stakeholders in their day-to-day activities, little is known on the role of visualizations in supporting software reuse tasks. This is the object of investigation of the study described in this work. There are some software visualization approaches in the literature that are intended to support software reuse, but literature lacks of a solid body of knowledge or a reference model of software visualizations targeted to reuse. Consequently, stakeholders may not be able to properly choose reuse-oriented visualizations (i.e., based on their quality and concrete evidence on their actual effectiveness) for a given scenario.

1.2. Organization

This text is organized as follows: Section 2 and 3 describe the study planning and execution, respectively, Section 4 presents an analysis of the findings along with a brief discussion, and Section 5 contains the final remarks.

2. Planning

2.1. Goal

This study aims at characterizing and identifying visualization approaches that can be used for supporting software reuse, regardless of the focus of support. The study goals are described in the Goal-Question-Metric (GQM) format [Basili et al. 1994] as follows:

Analyze tools and approaches described in publications

For the purpose of characterizing

With respect to visualizations for supporting software reuse

Under the point of view of the researchers

In the context of software development tasks and organizational tasks

2.2. Object of Study

The objects of this study are the publications that present visualizations supporting software reuse.

2.3. Expected Results

The expected results are (i) the identification of visualizations that can be used for supporting software reuse, as well as their features and limitations, and (ii) the establishment of a solid body of knowledge on visualizations for software reuse.

2.4. Methodology

Since this study aims mainly at characterizing the state-of-the-art, it is performed by means of a *Quasi*-Systematic Review [Travassos et al. 2008]. This kind of study is also known as Systematic Mapping Study, i.e., a study that aims to identify and categorize the research in a fairly broad topic area [Kitchenham et al. 2009]. However, since this study must explore the same rigor and formalism for the methodological phases of protocol preparation and running (except for the fact that no meta-analysis in principle can be applied), the *quasi*-systematic review denomination seems to be more appropriate [Travassos et al. 2008].

2.5. Research Questions

The research questions are decomposed into primary (*PQ*), secondary (*SQ*) and tertiary (*TQ*) questions. They map to the data extraction information, shown in Section 2.8, and are partially inspired in the work by Maletic et al. (2002).

- *PQ*: Which visualization approaches have been proposed to support software reuse?
  - *SQ1*: How do visualizations support software reuse?
    - *TQ1.1*: Which software engineering activities are addressed by the visualizations?
    - *TQ1.2*: Which reuse-related tasks are supported by these visualizations?
  - *SQ2*: To which stakeholders are these visualizations intended/targeted?
  - *SQ3*: Which items/data are visually represented?
    - *TQ3.1*: Where do these items/data come from?
    - *TQ3.2*: How are these items/data collected?
  - *SQ4*: Which visualization metaphors are used?
    - *TQ4.1*: How are data mapped to the visualizations?
    - *TQ4.2*: Which visualization strategies and techniques are employed?
  - *SQ5*: Where are the visualizations displayed?
    - *TQ5.1*: Which resources can be used for interacting with the visualizations?
  - *SQ6*: Which hardware/software resources are needed to deploy and execute the visualization tools?
    - *TQ6.1*: Which programming languages, APIs and frameworks are used?

- *SQ7*: Which methods are used for assessing the quality[2] of the visualizations (if any)?
  - *TQ7.1*: In which scenarios are the visualizations employed (if any)?
  - *TQ7.2*: Which aspects of the visualizations are evaluated (if any)?
  - *TQ7.3*: What are the results/outcomes of the conducted evaluations (if any)?

2.6. Search String Definition and Source Selection

2.6.1. Research Question Structuring

The search string structure is based on the PICO approach [Pai et al. 2004], which separates the question into **P**opulation of interest, **I**ntervention or exposure being evaluated, **C**omparison intervention (if applicable) and **O**utcome. As this study aims mainly at characterizing the state-of-the-art, no comparison is carried out, i.e., it can be classified as a *quasi*-systematic review [Travassos et al. 2008].

Population (P)

Publications and works related to software reuse.

Keywords:

- software, system, program, asset, application, artifact
- reuse, reusability, reusable

Intervention (I)

Visualizations related to software reuse.

Keywords:

- visualization, visual, visualisation

Comparison (C)

Not applicable (N/A).

Outcome (O)

The outcome corresponds to the information to be extracted from the publications (as listed in the Data Extraction section). However, it was observed that the publications' titles and abstracts do not contain these key terms; due to this, such information cannot be identified on them. Thus, as well as Santa Isabel's [Santa Isabel 2011] and França's [França & Travassos 2011] works, the outcome is not included in the definition of the keywords and will be taken into account only during the information extraction stage. The main reason for this decision is because the outcome terms would constrain the comprehensiveness of publications if they were included in the search string – they only represent information that shall be extracted from the publications.

[2] Quality evaluation/assessment encompasses any quality attributes, such as effectiveness, efficacy, amongst others.

2.6.2. Definition of Control Publications

Some publications identified by means of informal reviews were used as a preliminary input for the search string definition, i.e., compounded an initial data set (i.e., a baseline) of control[3] for the definition of the search string. Such publications are:

- Biddle, R., Marshall, S., Miller-Williams, J., Tempero, E. (1999). "Reuse of debuggers for visualization of reuse". In: *Proceedings of the 5th Symposium on Software Reusability (SSR 1999)*, Los Angeles, USA, pp. 92-100, May.
- Alonso, O., Frakes, W. B. (2000). "Visualization of Reusable Software Assets". In: *Proceedings of the 6th International Conference on Software Reuse (ICSR 2000)*, pp. 251-265, Vienna, Austria, June.
- Marshall, S. (2001). "Using and Visualizing Reusable Code: Position Paper[4] for Software Visualization Workshop". In: *Workshop on Software Visualization, 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, Tampa, USA, October.
- Marshall, S. Jackson, K., Anslow, C., Biddle, R. (2003). "Aspects to visualising reusable components". In: *Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2003)*, Adelaide, Australia, pp. 81-88, February.
- Duszynski, S., Knodel, J. and Becker, M (2011). "Analyzing the source code of multiple software variants for reuse potential". In: *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, Limerick, Ireland, pp. 303-307, October.

These works allowed the extraction of some keywords that can be used in the search string of this work.

2.6.3. Source Selection

The sources must meet the following initial requirements:

- In case of search engines, the indexed publications must be available on the web and written in English or in Portuguese.
- In case of conference proceedings and journals not indexed by the search engines, they must be representative of the software reuse domain. Publications must be in English or in Portuguese.

The chosen search engine for carrying out the review is Scopus[5], due to its well-known stability, reliability, interoperability with different referencing systems, and high coverage – its database indexes most of the publications that are available in different digital libraries or other search engines (e.g., Compendex, IEEE, ACM Digital Library, Springer, Web of Science etc.) [Santa Isabel 2011]. Besides, it indexes relevant journals and proceedings from the main

---

[3] Control is the baseline or the initial data set which the researcher already possesses [Biolchini et al. 2005].

[4] Although this publication is a position paper, it was decided to include it as control because it was considered as one of the pioneers on the topic.

[5] http://www.scopus.com/

software engineering conferences that comprise software reuse as a topic of interest. Examples of such conferences include:

- International Conference on Software Reuse (ICSR);
- International Conference on Software Maintenance (ICSM);
- European Conference on Software Maintenance and Reengineering (CSMR);
- International Conference on Information Reuse and Integration (IRI);
- International Conference on Software Engineering (ICSE);
- etc.

Thus, it was considered that the coverage/comprehensiveness provided by Scopus would be sufficient for the scope of this research.

Because Portuguese is the native language of the researchers involved in this study, it was decided that publications in Portuguese should be analyzed as well. The following conferences were identified as relevant for the purpose of this research:

- Brazilian Symposium on Software Engineering (SBES);
- Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS) and its predecessor Workshop on Component-Based Development (WDBC);
- Brazilian Symposium on Software Quality (SBQS).

Given that the Brazilian digital library (BDBComp[6]) does not index all the proceedings from any of these conferences (until the date of creation of this protocol), a manual search is required, following the same selection procedure (described in Section 2.7).

Before defining the search string, the title, abstract and keywords of each publication selected for control should be analyzed, aiming at identifying whether the most possibly common terms ("software", "reuse" and "visualization") would provide a good level of sensibility without the corresponding synonym terms. This is also used for verifying the adequacy of the search terms. The analysis is presented in Table 1.

[6] http://www.lbd.dcc.ufmg.br/bdbcomp/

Table 1. Analysis of title, abstract and keywords of control publications (occurrences of terms)

#1 – Title: Reuse of debuggers for visualization of reuse [Biddle et al. 1999]
Abstract: We have been investigating tool support for managing reuse of source code. One approach we have been exploring is the use of visualization in programming tools. A difficulty with this approach is that effective program visualization of ordinary programs requires substantial sophisticated low-level software. Our solution to this problem is to reuse debugging systems in an innovative way. We present our experience with this approach, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse.
Keywords: N/A
#2 – Title: Visualization of Reusable Software Assets [Alonso & Frakes 2000]
Abstract: This paper presents methods for helping users understand reusable software assets. We present a model and software architecture for visualizing reusable software assets. We described visualizations techniques based on design principles for helping the user understand and compare reusable components.
Keywords: Representation methods, Software assets, Information visualization, 3Cs, XML.
#3 – Title: Using and Visualizing Reusable Code [Marshall 2001]
Abstract: This paper describes a software visualization tool for helping a developer reuse existing Java code. The tool supports the creation and viewing of visual documentation of reusable code based on a developer's experience of using that code. The visual documentation, in essence software visualisations, can be used by the developer to understand what the code does, and how it does it. We have sought to create a tool that can create customizable software visualizations of Java code with minimal modifications to the code itself. This paper looks at both our first prototype, a stand alone Java application called Dyno, as well as at our second prototype called Vare. Vare expands on Dyno by working over a network and also acting as a code repository. We discuss the issues that have arisen so far in our development of these prototypes.
Keywords: Software Visualization, Test Driving, Code Repositories, Java
#4 – Title: Aspects to visualising reusable components [Marshall et al. 2003]
Abstract: We are interested in helping developers reuse software by providing visualisations of reusable code components. These visualisations will help determine if and how a given code component can be reused in the developer's new context. To provide these visualisations, we need both formatted information and tools. We need a format to describe the visualisations in. We need tools to create the visualisations. We need a format to describe information about the component and its runtime usage, and we need a tool to gather this information in the first place. In this paper, we discuss our two wish-lists for the required information formats. We set this against the background of software visualisation and code reuse research. Currently we are working with components from object oriented languages, specifically Java.
Keywords: Software Visualisation, Test Driving, Code Reuse

#5 – Title: Analyzing the Source Code of Multiple Software Variants for Reuse Potential [Duszynski et al. 2011]
Abstract: Software reuse approaches, such as software product lines, can help to achieve considerable effort and cost savings when developing families of software systems with a significant overlap in functionality. In practice, however, the need for strategic reuse often becomes apparent only after a number of product variants have already been delivered. Hence, a reuse approach has to be introduced afterwards. To plan for such a reuse introduction, it is crucial to have precise information about the distribution of commonality and variability in the source code of each system variant. However, this information is often not available because each variant has evolved independently over time and the source code does not exhibit explicit variation points. In this paper, we present Variant Analysis, a scalable reverse engineering technique that aims at delivering exactly this information. It supports simultaneous analysis of multiple source code variants and enables easy interpretation of the analysis results. We demonstrate the technique by applying it to a large industrial software system with four variants.
Keywords: Software reuse, product lines, reverse engineering, variant, visualization

The distribution of the search terms (in terms of titles, abstracts, and keywords) is summarized in Table 2.

Table 2. Distribution of the search terms

<table>
<tr><th>Search term</th><th># of occurrences of the search term</th><th># of publications which contain the search term</th></tr>
<tr><td>application</td><td>01 |</td><td>1 (#3)</td></tr>
<tr><td>assets</td><td>04 ||||</td><td>1 (#2)</td></tr>
<tr><td>program</td><td>01 |</td><td>1 (#1)</td></tr>
<tr><td>programs</td><td>01 |</td><td>1 (#1)</td></tr>
<tr><td>software</td><td>19 |||||||||||||||||||</td><td>5 (#1, #2, #3, #4, #5)</td></tr>
<tr><td>system</td><td>02 ||</td><td>1 (#5)</td></tr>
<tr><td>systems</td><td>03 |||</td><td>2 (#1, #5)</td></tr>
<tr><td>reusable</td><td>08 ||||||||</td><td>3 (#2, #3, #4)</td></tr>
<tr><td>reusability</td><td>01 |</td><td>1 (#1)</td></tr>
<tr><td>reuse</td><td>15 |||||||||||||||</td><td>4 (#1, #3, #4, #5)</td></tr>
<tr><td>reused</td><td>01 |</td><td>1 (#4)</td></tr>
<tr><td>visual</td><td>02 ||</td><td>1 (#3)</td></tr>
<tr><td>visualisation</td><td>02 ||</td><td>1 (#4)</td></tr>
<tr><td>visualisations</td><td>06 ||||||</td><td>2 (#3, #4)</td></tr>
<tr><td>visualising</td><td>01 |</td><td>1 (#4)</td></tr>
<tr><td>visualization</td><td>08 ||||||||</td><td>4 (#1, #2, #3, #5)</td></tr>
<tr><td>visualizations</td><td>02 ||</td><td>2 (#2, #3)</td></tr>
<tr><td>visualizing</td><td>02 ||</td><td>2 (#2, #3)</td></tr>
</table>

The third column (which indicates how many publications contain the corresponding search term) indicates that, in the *software* category, the term “software” is present in all of the publications selected for control, while the remaining terms only appear in one or two different publications. In the *reuse* category, both “reuse” and “reusable” terms are frequent, although none of them comprehends all of the analyzed publications. Finally, in the *visualization* category,

it is important to note that the only publication which is not covered by the term "visualization" (#4) contains the terms "visualisation" and "visualisations" – the "-ise" form is more frequently used in British English (UK) and its variations, while the "-ize" form is more common in American English (US) and its variations.

The number of occurrences of the search terms (depicted in the second column) suggests that "software", "reuse" and "visualization" are indeed the most common terms. However, two of the controls would not be retrieved by using only these main search terms ("software AND reuse AND visualization"), since only publications #1, #3 and #5 contain these three terms simultaneously. This indicates that the related terms (presented in the PICO definition) should be tested for obtaining a broader (and still relevant) range of publications. Additionally, although the population (**P**) is composed by publications and works related to software reuse, these terms do not always appear together, thus making word aggregations such as "software reuse" or "reusable software" too restrictive and hence inappropriate.

2.6.4. Search String Calibration

After performing some tests with the terms in the search engines, it was decided to suppress the terms "application" from the search string, as well as the term "artifact" (which was included previously as a correlated term). The reasons are twofold: first, based on the analysis on the occurrences of terms, they barely appeared in relevant results; moreover, the search results became very noisy with the inclusion of these terms, since they are overloaded in meaning, often used in situations not related to software development (e.g., archaeology artifacts, application for employment etc.).

Additionally, in spite of the frequency of other search terms depicted in Table 2 (e.g., "visualizing"), it was decided to keep only a subset of them that would return the control publications. Most of the obtained results have been already achieved by using the terms previously defined, which indicates that their usage could bring more noise than positive contributions to the research. Thus, it was decided to suppress them; such terms can be added later in a new round as a complement, if needed.

By using the structure of the PICO approach and taking into account the decision of suppressing the Outcome, the search string is the combination between the keywords presented in **P** (population) and **I** (intervention). Thus, the search string was defined as follows (Table 3):

Table 3. Search string definition

Search string	Scopus search string	# of results
((software OR system OR program OR asset) AND (reuse OR reusability OR reusable)) AND (visual OR visualization OR visualisation)	TITLE-ABS-KEY(((software OR system OR program OR asset) AND (reuse OR reusability OR reusable)) AND (visual OR visualization OR visualisation))	1204

After that, the following Document Types were removed from the search, since they are characterized as gray literature (i.e., technical reports, white papers, manuals and works in progress) [Lisboa et al. 2010][7]:

[7] There is a corresponding exclusion criterion presented in Section 2.7.2.

- “Conference Review” (40 entries);
- “Note” (3 entries);
- “Short Survey” (1 entry, unrelated);
- “Undefined” (1 entry, unrelated).

The subsequent version of the search string is presented in Table 4. The aforementioned filter is presented in italic.

Table 4. Second round of the search string definition (filters in italic)

Search string	Scopus search string	# of results
((software OR system OR program OR asset) AND (reuse OR reusability OR reusable)) AND (visual OR visualization OR visualisation)	TITLE-ABS-KEY(((software OR system OR program OR asset) AND (reuse OR reusability OR reusable)) AND (visual OR visualization OR visualisation)) *AND (EXCLUDE(DOCTYPE, “cr”) OR EXCLUDE(DOCTYPE, “no”) OR EXCLUDE(DOCTYPE, “sh”) OR EXCLUDE(DOCTYPE, “Undefined”))*	1159

As it can be seen, a large number of publications were obtained; however, it was decided not to constrain the search string, due to the exploratory nature of this study[8].

In spite of that, it was noticed that only 2 out of the 5 control publications were captured by this search string (publications #2 and #5). After investigating this issue, it was noticed that the 3 missing publications (#1, #3, #4) were not indexed by Scopus. Publication #3 was not found (by the time of this search) because it was neither indexed by Scopus nor any other academic search engine (it was found only by a Google Search). On the other hand, publications #1 and #4 are retrievable only from ACM Digital Library (DL) and Google Scholar. This led to reviewing the sources selection for the search.

Google Scholar has little support for advanced searching, and does not allow searching simultaneously on titles, abstracts and keywords, separately from the whole publication (only title filters are allowed). Because titles are not good representatives of the publication content and searching the full publication would bring more noise than relevant results, it was decided to keep Google Scholar out of the sources. ACM DL, in its turn, does not have any direct export functions, hampering to export the search results[9] and hence the identification of duplicates. Also, search results do not present the full abstract, so that publications should be exported one by one, which would require much effort for little benefit[10]. Other researchers have also pointed out problems with this search engine, e.g., the difficulties related to obtaining the same results when the search was repeated [Barcelos & Travassos 2006] and the lack of support for complex

[8] A similar situation was later identified in [Novais et al. 2013].

[9] See http://dl.acm.org/faq_dl.cfm for details (checked in November 30, 2013).

[10] The following search on ACM returns about 9117 results: ((software or system or program or asset) AND (reuse OR reusability OR reusable) AND (visual OR visualization OR visualisation)).

logical combinations [Brereton et al. 2007]. Due to these issues, ACM DL is not included in the search sources set.

Since ACM is the only digital library that contains the publications #1 and #4, it was decided to overcome this limitation by visiting the ACM Author Profile Page[11] of the respective authors and searching for the search string terms in the titles, abstracts and keywords of each listed publication. This decision was taken because the research described in these publications belongs to a specific research group, and contains a set of related works (in terms of goals and features).

Thus, the calibrated string presented in Table 4 was chosen as the final search string for the first round of this review. The search results are disposed in the Subject Areas in Table 5 and Figure 2 (it is important to note that there are entries located in more than one subject area):

Table 5. Results from second attempt

Subject Area	# of results
Computer Science	695
Engineering	491
Mathematics	173
Physics and Astronomy	77
Biochemistry, Genetics and Molecular Biology	69
Medicine	56
Social Sciences	40
Earth and Planetary Sciences	34
Environmental Science	32
Materials Science	29
Chemical Engineering	27
Decision Sciences	22
Agricultural and Biological Sciences	19
Business, Management and Accounting	13
Energy	13
Arts and Humanities	10
Health Professions	10
Psychology	6
Chemistry	5
Immunology and Microbiology	5
Neuroscience	5
Nursing	3
Pharmacology, Toxicology and Pharmaceutics	2
Dentistry	1
Veterinary	1
Undefined	3
Total	**1841**

[11] See http://www.acm.org/publications/acm-author-profile-page for details (checked in November 30, 2013).

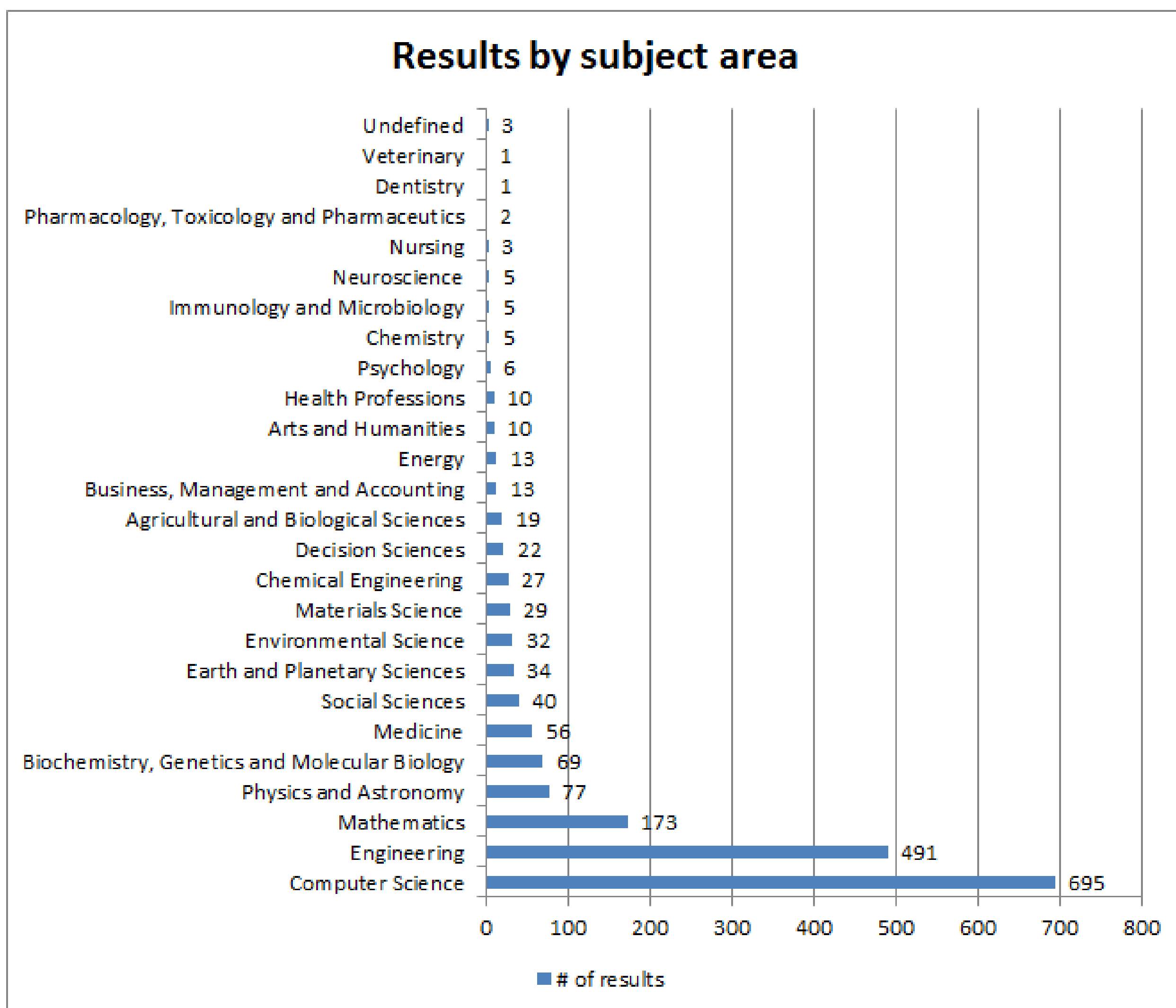


Figure 2. Distribution of search results by subject area

A Portuguese version of the search string (presented in Table 6) did not find any results in the search engine. Thus, only the manual search on the identified sources should be performed for this language.

Table 6. Search string in Portuguese

Search string	Scopus search string	# of results
((software OR sistema OR programa OR ativo) AND (reuso OR reúso OR reutilização OR reusabilidade OR reusável OR reutilizável)) AND (visual OR visualização)	TITLE-ABS-KEY(((software OR sistema OR programa OR ativo) AND (reuso OR reúso OR reutilização OR reusabilidade OR reusável OR reutilizável)) AND (visual OR visualização)) *AND (EXCLUDE(DOCTYPE, "cr") OR EXCLUDE(DOCTYPE, "no") OR EXCLUDE(DOCTYPE, "sh") OR EXCLUDE(DOCTYPE, "Undefined"))*	0

2.7. Procedure for Studies Selection

Due to the diversity of subject areas and for not constraining the systematic review search string, it was decided to make *title reading* as the first selection stage, before the usual *abstract reading*. Since the chosen search string is too broad, such stage can be useful for eliminating publications that are clearly targeted to other fields of study (e.g., publications in medicine or chemistry), thus reducing the amount of unnecessary abstract readings. It must be ensured, though, that a publication must be selected whenever there is not enough confidence for excluding it, so that it can be more carefully analyzed during the next stage. The process is depicted in Figure 3.

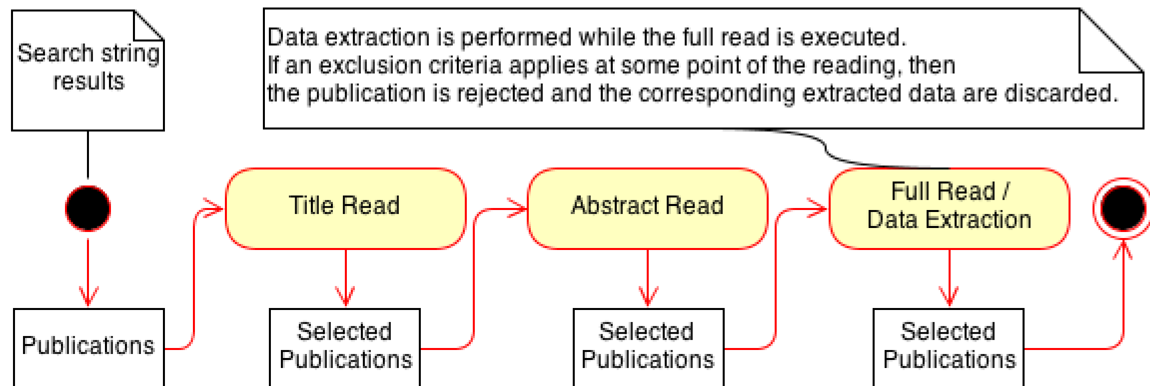


Figure 3. Systematic review selection process

The title (in the *title reading* stage) and the abstract (in the *abstract reading* stage) of each found publication is read and evaluated against the inclusion and exclusion criteria. Selected publications are then read entirely and once more evaluated against the inclusion and exclusion criteria (*full reading* stage), in order to remove false positives. In this stage, data extraction is performed (*data extraction* stage) on the remaining selected publications. By performing the *data extraction* at the same time of the *full reading*, it is expected to achieve a greater level of certainty regarding the pertinence of the publication to the scope of the study.

The process is executed by two researchers. Publications that are included in a given selection stage (i.e., by the publications' title and/or abstract reading) can be excluded in the full reading/data extraction stage. Since the publication is read entirely, leading to a better understanding of it, these stages allow for a more confident inclusion/exclusion decision in terms of the established criteria [França & Travassos 2011]. In case of conflict of opinions (between the two researchers) regarding a publication, a third researcher must execute the evaluation procedure for resolving it.

2.7.1. Inclusion Criteria (IC)

- IC1: The publication is fully available (on the web, for results from search engines) or has been provided by the authors.
- IC2: The publication is related to software reuse.
- IC3: The publication presents a visualization-based approach or tool for supporting software reuse.

2.7.2. Exclusion Criteria (EC)

- EC1: The publication is not available (on the web, for results from search engines), neither was provided by the authors.
- EC2: The publication is characterized as gray literature (i.e., technical reports, white papers, manuals and works in progress)[12] [Lisboa et al. 2010].
- EC3: Duplicate publication or self-plagiarism (when several reports of a study exist in different journals, the most complete version of the study must be included in the review [Kitchenham et al. 2009]).
- EC4: The publication is not about reuse, or tackles reuse in other areas not related to software development.
- EC5: The publication does not mention the use of visualization for supporting software reuse.

Regarding EC5, it was concluded (during some search tests) that visual programming approaches and languages do not belong to the scope of this study whenever the visualization resources are used only for representing language elements. A visual programming language lets users create programs by manipulating program elements graphically rather than by specifying them textually. All the visual elements implicitly improve reuse somehow, since they usually represent a certain piece of text/code in a higher abstraction level. The goal of this study is to identify approaches that are specifically targeted to support software reuse.

2.8. Data Extraction

The data extraction fields are identified with their corresponding research questions.

In addition to the usual publication metadata (which are listed in the end of this section), it is necessary to define a metadata field for identifying each visualization approach:

- Approach/tool name *(PQ)*

In order to identify an initial set of data to be extracted from the findings, this work uses the five dimensions of software visualization proposed by Maletic et al. (2002). The task-oriented framework proposed by these authors takes into account previous work on taxonomic descriptions for emphasizing general tasks of understanding and analysis during the development and maintenance of large-scale software systems [Maletic et al. 2002]. The framework dimensions reflect the why, who, where, what, and how of the software visualization.

Additionally, in order to complement the framework with information that is relevant to the visualization users and encompass other aspects related to the findings of this study, two additional, complementary dimensions that are not (or at least not directly) addressed in the original framework are proposed and used in this work: one related to the **requirements** of the visualization approaches (***which***) and other related to **evidence** on their use (***worthwhile***). These dimensions are depicted in Figure 4.

[12] This criterion is employed for ensuring the quality of the findings; however, when the number of search results is too low, it may not be taken into consideration.

Figure 4. Dimensions of software visualizations (extended from [Maletic et al. 2002])

Each dimension maps to a secondary question (*SQ*) shown in Section 2.5. Details on the dimensions and their corresponding information fields are described in the next subsections.

2.8.1. Task – *why* is the visualization needed?

A visualization system aims at supporting the understanding of one or more aspects of a software system, and this understanding process will in turn support a particular task [Maletic et al. 2002]. Thus, this dimension indicates what particular software engineering tasks are supported by the visualization [Maletic et al. 2002].

In order to understand why each of the visualizations is needed in the reuse scenario, it is important to identify which problems, motivations or issues leaded to the development of such approach. In this sense, the following fields are used in this work:

- Approach motivation/Assumptions *(SQ1)*
- Approach goals *(SQ1)*
- Visualizations' reuse-specific goals *(SQ1)*

Besides identifying which are the supported tasks, one important aspect that can be relevant is *when* – i.e., in which software engineering activities and stages of the development process – the visualization can be used. Particularly in this work, the purpose is to identify the software engineering activities (in which there are reuse opportunities) that are supported

somehow by the software visualizations. This can provide an overview of current tools' coverage, as well as identify opportunities for research and improvements.

Although this could be considered as an additional dimension (*when* it is visualized?), it was decided to keep it along with the *task* dimension (in which the supported tasks are identified), so that both can provide complementary information. Thus, this work resorts to the following fields:

- Software engineering activities addressed by the visualizations *(TQ1.1)*
- Reuse-related tasks supported by the visualizations *(TQ1.2)*

2.8.2. Audience – *who* will use the visualization?

The audience dimension defines the attributes of the users of the visualization system [Maletic et al. 2002]. Besides being oriented to distinct roles, different tools can also be tailored towards users with different skills (e.g., experienced versus beginner, developer versus manager etc.). An experienced developer may have different information needs other than a novice team member [Maletic et al. 2002].

In order to make visualizations effective in their goal, it is noteworthy to keep in mind that the representations and visual metaphors must be adapted to the stakeholders' perception abilities [Diehl 2007] [Schots et al. 2012], not the opposite (as it usually occurs [Diehl 2007]). The audience is represented in this work by the following field:

- Visualizations' audience (stakeholders who can benefit from the visualizations) *(SQ2)*

2.8.3. Target – *what* is the data source to represent?

The target of a software visualization system defines which (low level) aspects of the software are visualized, i.e., the work product, artifact, or part of the environment of the software system [Maletic et al. 2002]. Some examples include architecture, design, algorithm, source code, data, execution/trace information etc. Other types of target source data are measurements and metrics obtained from software, process information, and documentation; this type of information can support the software process and team management activities [Maletic et al. 2002]. Software development surroundings also provide several aspects that can be visualized [Schots & Werner 2012]. The aspects related to this dimension are described in this work by the following fields:

- Visualized items/data (what is visualized) *(SQ3)*
- Source of visualized items/data *(TQ3.1)*
- Collection procedure/method of visualized items/data *(TQ3.2)*

2.8.4. Representation – *how* to represent the data?

This dimension defines how the visualization is constructed based on the available information [Maletic et al. 2002]. According to [Few 2009], an aspect on which the effectiveness of information visualization hinges is the visualizations' ability to clearly and accurately represent information. The relationship between data values and visual parameters must be

univocal; otherwise, it may not be possible to distinguish one value's influence from another [Maletic et al. 2002]. In this work, the fields used for depicting this dimension are:

- Visualization metaphors used (how it is visualized) *(SQ4)*
- Data-to-visualization mapping (input/output) *(TQ4.1)*
- Visualization strategies and techniques *(TQ4.2)*

2.8.5. Medium – *where* to present the visualization?

The effectiveness of visualizations also relies on humans' ability to interact with them to figure out what the information means [Few 2009]. The medium is where the visualization is rendered, i.e., some display technology from which the user interacts and perceives the visualization [Maletic et al. 2002]. The medium dictates how interactions may occur; each one has different characteristics and in consequence is suited for different tasks [Maletic et al. 2002]. Some software visualization tools do not even exploit the graphics power of an average PC or laptop [Diehl 2007].

In this work, such dimension is comprised by the following fields:

- Device and/or environment used for displaying the visualizations (where it is visualized) *(SQ5)*
- Resources used for interacting with the visualizations *(TQ5.1)*

2.8.6. Requirements – *which* resources are required by or used in the visualization?

Although the original dimensions from Maletic et al.'s framework provide an organization of the goals and concepts implemented in the visualizations, it is not possible to distinguish what is needed to deploy and execute the tools. For instance, an important concern in any interactive visualization system is performance, in particular responsiveness to changes triggered by direct manipulation of images [Bavoil et al. 2005]. Certain visualizations may become costly, not only in terms of processing but also due to specific hardware and software solutions on which they depend. This cost can be expressed (to some extent) in terms of the visualizations' hardware and software requirements/dependencies, besides the programming languages, application programming interfaces (APIs) and frameworks reused for building it.

Visualization requirements can also provide indication on potentially conflicting configurations, e.g., if a given version of a framework used for building the visualization is not compatible (or has known behavior issues) with a certain hardware or software used by the organization. The prevention of conflicts can avoid unnecessary waste of time.

For capturing information of this new dimension, the following fields are used:

- Hardware and software requirements/dependencies *(SQ6)*
- Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*

2.8.7. Evidence – are the proposed visualizations *worthwhile*?

Many software visualization tools are continuously being developed by researchers and software development companies [Sensalire et al. 2009]. However, according to these authors,

many developers perform very limited or no evaluation at all of their tools. The lack of empirical studies is a shortcoming not only of software visualization research, but also of software engineering and computer science in general (in accordance with [Tichy 1998]). As a result, as stated by [Mulholland 1997], it may become unclear how effective such visualization tools are, either for students or professional programmers.

According to Diehl (2007), typical problems with evaluations of visualization techniques are the use of toy data sets and the generation of visual artifacts that suggest nonexistent relations; but, above all, many evaluations are biased because they have been done by the developers of the visualization. Moreover, there are some concerns on whether lessons in successive designs of software visualization tools, or whether the application of new technologies (e.g., 3D animation and the internet) has become the primary goal, rather than the true goal of making computer programs easier to understand [Mulholland 1997].

In order to determine if visualizations are worthwhile, i.e., effective in helping their target users, it is desirable to expose them to a proper evaluation [Sensalire et al. 2009]. This aspect is not emphasized in the original framework; quality attributes that illustrate the usefulness of the approaches (e.g., effectiveness) are put into the *Representation* dimension. This may "obfuscate" their importance and decrease their visibility. Thus, this dimension aims at characterizing what kinds of evaluation and assessment were carried out with the visualization (if any), as well as any indications (or identified limitations) for its use, providing insights of the visualizations' worthiness beforehand. This can be a valuable indication for people interested in making use of the visualization.

Thus, for describing this new dimension, the following fields are included in this work:

- Visualization evaluation methods *(SQ7)*
- Application scenarios of the visualizations *(TQ7.1)*
- Evaluated aspects *(TQ7.2)*
- Visualization evaluation results/outcomes *(TQ7.3)*

2.8.8. Data Extraction Form

The following information (presented in Table 7) shall be extracted and managed from each selected publication. The Google Spreadsheets tool[13] is used for supporting the data extraction process. Visualization dimensions are represented along with their corresponding questions (as depicted in Figure 4).

[13] http://spreadsheets.google.com/

Table 7. Data extraction form

	Field	Information to be extracted
Publication metadata	**Title**	*[Publication title]*
	Authors	*[List of authors separated by comma, e.g., "Singh, S., Cheung, L. K. Y." – "et al." must be avoided]*
	Publication date (year/month)	*[Year and month of publication, e.g., "September 2000"]*
	Publication type	*[Conference or Article (Journal)]*
	Source	*[Source of the publication, e.g., "Communications of the ACM" or "Proceedings of the International Conference on Software Engineering (ICSE 2007)"]*
	Volume and Edition (for journals)	*[Volume and edition, e.g., "v. 49, n. 10"]*
	Place (for conferences)	*[City and Country of event, e.g., "Washington, USA"]*
	Pages	*[Initial and final pages separated by hyphen, e.g., "pp. 184-191"]*
	Link (if applicable)	*[Link to the publication, preferably the Digital Object Identifier (DOI), e.g., "http://dx.doi.org/10.1109/ICSECOMPANION.2007.8"]*
	Abstract	*[Full abstract text]*
Visualization metadata	**Approach/tool name** *(PQ)*	*[Name of the approach/tool]*
	Screenshot	*[Screenshot of the approach/tool, if available]*
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	*[Problems, motivations or issues that leaded to the development of the approach]*
	Approach goals *(SQ1)*	*[Goals for which the approach was developed]*
	Visualizations' reuse-specific goals *(SQ1)*	*[Description of how the approach goals relate to software reuse, i.e., which goals support or are somehow related to reuse]*
	Software engineering activities addressed by the visualizations *(TQ1.1)*	*[Software engineering activities or development process stages that can be somehow supported by the visualizations (e.g., "requirements engineering", "software design", "software testing", "software maintenance" etc.), including the construction of reusable assets (development for reuse) or the reuse of these assets in a scenario (development with reuse)]*
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	*[Tasks supported by the visualizations, in a fine-grain level, e.g., "integrating reusable assets", "Searching and retrieving reusable assets" etc.]*
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	*[Software development stakeholders who can benefit from the visualizations, e.g., "programmers", "software designers", "end users" etc.]*
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	*[Items/data from the software development process that have a visual presentation; examples include source code entities (e.g., "classes and interfaces with their attributes and methods"), high-level artifacts (e.g., "UML diagrams"), metrics (e.g., "coupling", "number of commits" etc.), among others]*
	Source of visualized items/data *(TQ3.1)*	*[Sources from which the items/data are extracted, e.g., "version control system repository", "metrics base", "software tracing log file", "source folder" etc.]*
	Collection procedure/method of visualized items/data *(TQ3.2)*	*[Description on how the items/data are collected and/or aggregated by the approach, e.g., "parsing", "clustering algorithm" etc.]*

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	*[Visual metaphors used for describing the items/data, e.g., "squares and circles", "treemap", "graph" etc.]*
	Data-to-visualization mapping (input/output) *(TQ4.1)*	*[Description on how data are mapped to the visualizations, e.g., "classes are represented as circles and interfaces as triangles", "the color represents the complexity (the darker, the more complex)" etc.]*
	Visualization strategies and techniques *(TQ4.2)*	*[Strategies (e.g., "provide a global view while navigating into specific views") and techniques (e.g., "drill-down", "zoom", "clustering" etc.) used for displaying and interacting with the visualizations; strategies may use a given technique without mentioning it]*
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	*[Device used for displaying the visualizations, e.g., "Computer", "Smartphone", "Tablet", "Display wall" etc.]*
	Resources used for interacting with the visualizations *(TQ5.1)*	*[Resources that allow interacting with the visualizations, e.g., "mouse", "keyboard", "pen", "finger touch", "gestures" etc.]*
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	*[Hardware (e.g., "Quad-core processor", "Graphic card" etc.) and software (e.g., "Eclipse IDE", etc.) required for the approach]*
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	*[Programming languages, Application Programming Interfaces (APIs) and frameworks used for building the approach, e.g., "Java Reflection API", "Prefuse" etc.]*
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	*[Method applied for evaluating the approach, e.g., controlled experiment, observational study, case study etc.]*
	Application scenarios of the visualizations *(TQ7.1)*	*[Scenarios in which the approach was employed, e.g., "in an industrial setting", "in the context of an academic course" etc.]*
	Evaluated aspects *(TQ7.2)*	*[Evaluated approach aspects, e.g, performance, response time, usefulness, scalability etc.]*
	Visualization evaluation results/outcomes *(TQ 7.3)*	*[Evaluation findings and results]*

According to Kitchenham (2004), if the data require manipulation or assumptions and inferences to be made, an appropriate validation process should be specified. In this study, the following procedures take place:

- Data extracted by one researcher are reviewed by the other researcher;
- For publications retrieved from the search engine, an automatic search is performed in non-encrypted PDF files for cross-checking their content;
- Additionally, an external evaluator is responsible for supporting the validation process (i.e., checking whether the extracted data corresponds to the publication).

3. Execution

3.1. Execution Data from the Manual Search

The manual search (as mentioned in Section 2.6.3) was performed in the following Brazilian proceedings:

- Brazilian Symposium on Software Engineering (SBES): proceedings from 1987 (1st edition) to 2012 (26th edition) (including);
- Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS): proceedings from 2007 (1st edition) to 2012 (6th edition) (including);

- Workshop on Component-Based Development (WDBC): proceedings from 2002 (2nd edition)[14] to 2006 (6th edition) (including);
- Brazilian Symposium on Software Quality (SBQS): proceedings from 2002 (1st edition) to 2012 (11th edition) (including).

According to the selection procedures described in Section 2.7, the *abstract reading* stage should be executed after the *title reading* stage. However, due to the effort involved in handling and reading the printed proceedings, it was decided to perform the *title reading* stage and the *abstract reading* stage simultaneously in the manual search.

The search results are listed in Table 8.

Table 8. Study selection data (manual search)

	SBES	WDBC / SBCARS	SBQS
Title and abstract reading	556	158	315
Number of accepted publications	30	42	26
Number of rejected publications	526	116	289
Number of duplicated publications	0	0	0
Full reading	30	42	26
Number of accepted publications	0	0	0
Number of rejected publications	30	42	26
Number of duplicated publications	0	0	0

As it can be seen, from the 1030 analyzed publications, no one was selected. Most of the publications selected during the title/abstract reading (98) are related to software reuse; however, no publication mentions the use of visualization resources with the goal of supporting software reuse.

3.2. Execution Data from the Search Engine

The searches were performed on October 1st, 2012 at 3PM local time (UTC/GMT -3) in both the Scopus search engine and the selected ACM Author Profile Pages (mentioned in Section 2.6.4). Although no time constraint was set, it is believed from the search results that the publications were obtained in the range between 1980 and September 2012. However, publications that had not been indexed until the date of search may have been added to the digital libraries afterwards.

As described in Section 2.6.4, 1159 results were obtained from Scopus by performing the search with the chosen search string. The publications were exported from this search engine and formatted in tables. After that, the procedure for studies selection described in Section 2.7 took place.

The search performed on the ACM Author Profile Pages was conducted in a different way: all the publications listed in the pages of each key author identified from the control

[14] The first edition of WDBC has no proceedings; selected works evaluated by the program committee were invited for publication in a book: Gimenes, I. M. S., Huzita, E. H. M. (2005). *Component-Based Development: Concepts and Techniques* (in Portuguese), 1st ed., 304p., Editora Ciência Moderna.

publications (as discussed in Section 2.6.4) were manually exported and their title, authors and keywords were extracted with the support of regular expressions in a text editing tool (Notepad++[15]). After that, duplicates were semi-automatically identified and removed, resulting in 304 results. Then, a semi-automatic searched was performed using the search string terms, and 6 publications were returned.

Table 9 and Figure 5 summarize the study selection stages in terms of accepted, rejected and duplicated publications.

Table 9. Study selection data (search engines)

	Scopus		ACM
	1st researcher	**2nd researcher**	**Both researchers**
Title reading	1159	1159	6
Number of accepted publications	411	320	6
Number of rejected publications	740	831	0
Number of duplicated publications	8	8	0
Abstract reading	411	320	6
Number of accepted publications	77	45	6
Number of rejected publications	326	275	0
Number of duplicated publications	8	0	0
Full reading	77	45	6
Number of accepted publications	**29**	**19**	**5**
Number of rejected publications	47	26	0
Number of duplicated publications	1	0	1

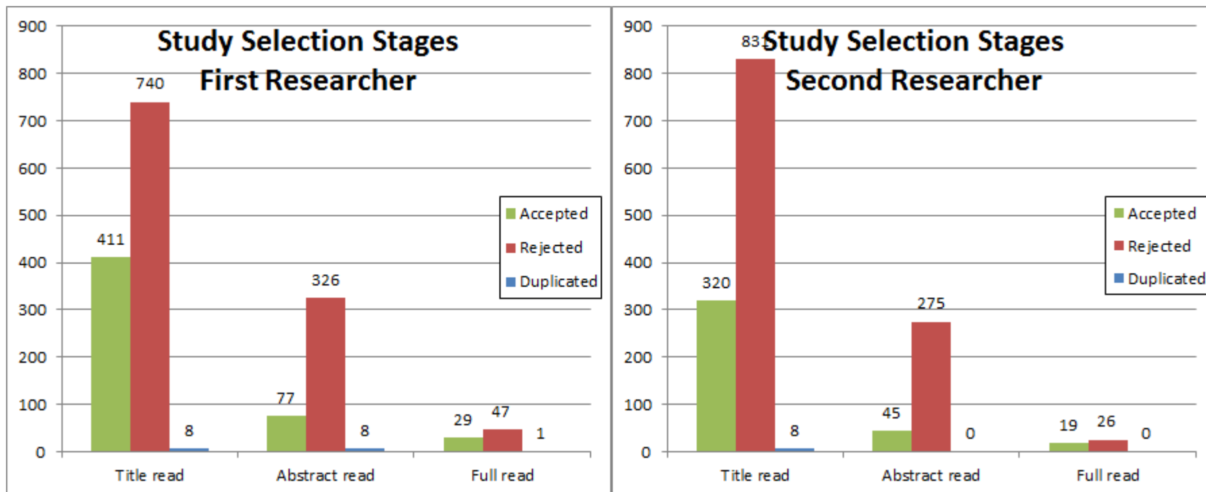


Figure 5. Scopus data from study selection stages, by researcher

Details on the execution of each stage are given in the next sections. For summarizing, whenever "X + Y" appears in the text, X refers to Scopus and Y to ACM results.

[15] http://notepad-plus-plus.org/

3.3. Study Selection

From the total number of results (1159 + 6), ten entries were first identified as duplicates (i.e., there was another corresponding entry among the results). After analyzing their abstracts, two of these entries were false positives (i.e., they were not characterized as duplicates). Thus, the 8 duplicate entries were removed. The duplicates removal leads to 1151 + 6 entries, which were effectively used as input for the title reading stage.

From this stage, a master student (hereinafter called *second researcher*) supported the study selection process, aiming to avoid selection bias and give more confidence in the results. He performed the selection steps in a different time from the main researcher's, due to availability issues; however, during the process, he did not know any of the results obtained by the main researcher (hereinafter called *first researcher*), i.e., he executed it in a "blind" fashion.

In order to provide an overview of the pre-execution stage, Figure 6 shows a word cloud[16] (or tag cloud) for the 600 most frequent terms in the publications' abstracts before the title reading stage. In the context of this work, the larger the term in the word cloud, the most frequently it is present in the abstracts. It can be noted that the main key terms used in the search string are displayed larger (as expected), but there are other (initially non-related) terms with similar frequency, e.g., "simulation" and "edu".

Figure 6. Word cloud indicating the 600 most frequent terms before the title reading stage

After the title reading stage, 411 + 6 publications were selected by the first researcher, based on the inclusion/exclusion criteria, and the remaining 740 + 0 were rejected. The second

[16] Generated by the Tagxedo tool – http://www.tagxedo.com/

researcher, in his turn, selected 320 + 6 publications and rejected 831 + 0. The large number of selected publications can be explained by the established procedure described in Section 2.7: publications were selected for the next stage whenever there was not enough confidence for excluding them. Figure 7 shows the word cloud after the title reading stage, i.e., before the abstract reading (next stage). A little decreasing of unrelated words can be noticed, although it is still not expressive.

Figure 7. Word cloud indicating the 600 most frequent terms after the title reading stage (before the abstract reading)

The abstract of the 411 + 6 selected publications was read and evaluated by the first researcher against the established criteria. This resulted in the rejection of 326 + 0 publications. From those, 6 were rejected due to the exclusion criterion EC2 (gray literature), 276 due to EC4 (i.e., they were not about reuse, or tackled reuse in other areas not related to software development), and 44 due to EC5 (i.e., although they were about software reuse, they did not mention the use of visualization for supporting software reuse). In addition to these 326 + 0 rejected publications, 8 + 0 publications were identified as duplicates (EC2) in this stage. 77 + 6 publications were then selected for the full reading stage.

The second researcher performed the same process stage for the 320 + 6 selected publications, and rejected 275 + 0 publications. The remaining 45 + 6 publications were likewise selected for the full reading stage.

Figure 8 shows the word cloud after the abstract reading stage, i.e., before the full reading stage. Words like “repository” and “understanding” start to become more frequent, since results became more related to the topic of this work.

Figure 8. Word cloud indicating the 600 most frequent terms after the abstract reading stage (before the full reading)

The full reading stage was executed along with the data extraction stage, i.e., as publications were selected, the corresponding data for answering the research questions (as specified in Section 2.8) were extracted.

Concerning the availability of the publications, 10 + 0 publications selected by the first researcher and 9 + 0 publications selected by the second researcher were not available on the web. The authors of such publications were then contacted by e-mail. Some of the e-mails returned because the addresses were no longer active or available; in these cases, a search for an alternative e-mail address was performed in Google. By using this strategy, 4 out of the 10 missing publications could be retrieved for the first researcher, and 0 out of 9 could be retrieved for the second researcher.

For the first researcher, the extraction was carried out with the 29 + 5 (out of the 77 + 6) publications selected in this stage. From the remaining publications, 6 + 0 were rejected because they were not available on the web, neither were provided by the authors (EC1), 24 + 0 because they were not about reuse or tackled reuse in other areas not related to software development (EC4), and 17 + 0 because even though they were about software reuse, they did not mention the use of visualization for supporting software reuse (EC5), amounting to 47 + 0 rejected publications. Additionally, 1 + 1 publications were identified as duplicates (EC3).

The second researcher, in his turn, selected 19 + 5 (out of the 45 + 6) publications in this stage and performed the data extraction. From the remaining publications, 9 + 0 were rejected because they were not available on the web, neither were provided by the authors (EC1), 10 + 0 because they were not about reuse or tackled reuse in other areas not related to software

development (EC4), and 7 + 0 because even though they were about software reuse, they did not mention the use of visualization for supporting software reuse (EC5), amounting to 26 + 0 rejected publications. Additionally, 1 + 1 publications were identified as duplicates (EC3).

Figure 9 shows the final word cloud after the full reading stage (i.e., extracted from the abstracts of the 29 + 5 selected publications). In this figure, key terms (such as "visualization") reasonably differ from other peripheral terms (e.g., "domain" or "directives"). It is noteworthy that some words (like "views") did not play a major role compared to their similar words (e.g., "visualization"). The sequence of pictures (from Figure 6 to Figure 9) shows how some key terms increased their proportional frequency and gained more relevance as unrelated publications were filtered out – which is an expected situation, as previously mentioned.

Figure 9. Word cloud indicating the 600 most frequent terms after the full reading stage (selected publications)

After the second researcher has finished the selection process, the consensus (for conflict resolution) took place. 14 + 5 publications were selected by both researchers, so they did not need to be reanalyzed. From the 15 publications selected only by the first researcher, 13 were included after discussion, and 2 were rejected. From the 5 publications selected only by the second researcher, 2 were included after discussion, and 3 were rejected (being 1 by a third researcher, since consensus had not been achieved). Details on the consensus stage can be found in Appendix A.

Beyond the control publication #3, another related publication [Anslow et al. 2004] was manually added. It was found based on the citations of the ACM key authors. It was agreed to include it in the consensus stage, along with the control publication already included.

The 36 selected publications are listed in Table 10.

Table 10. Selected publications (sorted alphabetically)

BibTeX ID	Publication
Ali200950	Ali, J. (2009). "Cognitive support through visualization and focus specification for understanding large class libraries". *Journal of Visual Languages and Computing*, v. 20, n. 1, pp. 50-59.
Alonso1998483	Alonso, O., Frakes, W. B. (2000). "Visualization of Reusable Software Assets". In: *Proceedings of the 6th International Conference on Software Reuse (ICSR 2000)*, pp. 251-265, Vienna, Austria, June.
Anquetil2010427	Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., Sousa, A. (2010). "A model-driven traceability framework for software product lines". *Software and Systems Modeling*, v. 9, n. 4, pp. 427-451.
Anslow2004	Anslow, C., Marshall, S., Noble, J., Biddle, R. (2004). "Software visualization tools for component reuse". In: *2nd Workshop on Method Engineering for Object-Oriented and Component-Based Development, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, October.
Apel2011421	Apel, S., Beyer, D. (2011). "Feature cohesion in software product lines: An exploratory study". In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, Honolulu, Hawaii, pp. 421-430, May.
Areeprayolkij2010208	Areeprayolkij, W., Limpiyakorn, Y., Gansawat, D. (2010). "IDMS: A system to verify component interface completeness and compatibility for product integration". *Communications in Computer and Information Science*, v. 117 CCIS, pp. 208-217.
Bauer2012435	Bauer, V., Heinemann, L. (2012). "Understanding API usage to support informed decision making in software maintenance". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, Szeged, Hungary, pp. 435-440, March.
Biddle199992	Biddle, R., Marshall, S., Miller-Williams, J., Tempero, E. (1999). "Reuse of debuggers for visualization of reuse". In: *Proceedings of the 5th Symposium on Software Reusability (SSR 1999)*, Los Angeles, USA, pp. 92-100, May.
Charters2002765	Charters, S. M., Knight, C., Thomas, N., Munro, M. (2002). "Visualisation for informed decision making; from code to components". In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, Ischia, Italy, pp. 765-772, July.
Constantopoulos19951	Constantopoulos, P., Jarke, M., Mylopoulos, J., Vassiliou, Y. (1995). "The software information base: A server for reuse". *The VLDB Journal*, v. 4, n. 1, pp. 1-43.
Damaeviius2009507	Damaševičius, R. (2009). "Analysis of components for generalization using multidimensional scaling". *Fundamenta Informaticae*, v. 91, n. 3-4, pp. 507-522.
DeBoer200951	De Boer, R. C., Lago, P., Telea, A., Van Vliet, H. (2009). "Ontology-driven visualization of architectural design decisions". In: *Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009)*, Cambridge, UK, pp. 51-60, September.
Dietrich200891	Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., Duchrow, M. (2008). "Cluster analysis of Java dependency graphs". In: *Proceedings of the 4th ACM Symposium on Software Visualization (SOFTVIS 2008)*, Ammersee, Germany, pp. 91-94, September.

BibTeX ID	Publication
Duszynski2011303	Duszynski, S., Knodel, J., Becker, M. (2011). "Analyzing the source code of multiple software variants for reuse potential". In: *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, Limerick, Ireland, pp. 303-307, October.
Duszynski201237	Duszynski, S., Becker, M. (2012). "Recovering variability information from the source code of similar software products". In: *Proceedings of the 3rd International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2012)*, Zürich, Switzerland, pp. 37-40, June.
Feigenspan20121	Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G. (2013). "Do background colors improve program comprehension in the #ifdef hell?". *Empirical Software Engineering*, v. 18, n. 4, pp. 699-745.[17]
Gonçalves2007872	Gonçalves, E. M., Oliveira, M. D. S., Bacili, K. R. (2007). "DigitalAssets discoverer: Automatic identification of reusable software components". In: *Proceedings of the 22nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, Montreal, Canada, pp. 872-873, October.
Helfman199631	Helfman, J. (1996). "Dotplot patterns: a literal look at pattern languages". *Theory and Practice of Object Systems*, v. 2, n. 1, pp. 31-41.
Holmes2007100	Holmes, R., Walker, R. J. (2007). "Task-specific source code dependency investigation". In: *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*, Banff, Canada, pp. 100-107, June.
Kelleher200550	Kelleher, J. (2005). "A reusable traceability framework using patterns". In: *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2005)*, Long Beach, USA, pp. 50-55, November.
Lange1995342	Lange, D. B., Nakamura, Y. (1995). "Interactive visualization of design patterns can help in framework understanding". In: *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)*, Austin, USA, pp. 342-357, October.
López20091198	López, C., Inostroza, P., Cysneiros, L. M., Astudillo, H. (2009). "Visualization and comparison of architecture rationale with semantic web technologies". *Journal of Systems and Software*, v. 82, n. 8, pp. 1198-1210.
Mancoridis199374	Mancoridis, S., Holt, R. C., Penny, D. A. (1993). "Conceptual framework for software development". In: *Proceedings of the 1993 ACM Computer Science Conference*, Indianapolis, USA, pp. 74-80, February.
Marshall2001	Marshall, S. (2001). "Using and Visualizing Reusable Code: Position Paper for Software Visualization Workshop". In: *Workshop on Software Visualization, 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, Tampa, USA, October.
Marshall2001103	Marshall, S., Jackson, K., McGavin, M., Duignan, M., Biddle, R., Tempero, E. (2001). "Visualising reusable software over the web". In: *Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2001)*, Sydney, Australia, pp. 103-111, December.

[17] Although it is cited as 2013, it was retrieved from the search engine in 2012 when it was accepted for publication.

BibTeX ID	Publication
Marshall200381	Marshall, S., Jackson, K., Anslow, C., Biddle, R. (2003). “Aspects to visualising reusable components”. In: *Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2003)*, Adelaide, Australia, pp. 81-88, February.
Marshall200435	Marshall, S., Biddle, R., Noble, J. (2004). “Using software visualisation to enhance online component markets”. In: *Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2004)*, Christchurch, New Zealand, pp. 35-41, January.
McGavin2006153	McGavin, M., Wright, T., Marshall, S. (2006). “Visualisations of execution traces (VET): an interactive plugin-based visualisation tool”. In: *Proceedings of the 7th Australasian User Interface Conference (AUIC 2006)*, Hobart, Australia, pp. 153-160, January.
Mittermeir200195	Mittermeir, R. T., Bollin, A., Pozewaunig, H., Rauner-Reithmayer, D. (2001). “Goal-driven combination of software comprehension approaches for component based development”. In: *Proceedings of the 2001 Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, pp. 95-102, May.
Oliveira2007461	Oliveira, M., Gonçalves, E. M., Bacili, K. R. (2007)., “Automatic Identification of reusable Software development assets: Methodology and tool”. In: *Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI 2007)*, Las Vegas, USA, pp. 461-466, August.
Stollberg2007236	Stollberg, M., Kerrigan, M. (2007). “Goal-based visualization and browsing for semantic Web services”. *Lecture Notes in Computer Science*, v. 4832 LNCS, pp. 236-247.
Tangsripairoj2006283	Tangsripairoj, S., Samadzadeh, M. H. (2006). “Organizing and visualizing software repositories using the growing hierarchical self-organizing map”. *Journal of Information Science and Engineering*, v. 22, n. 2, pp. 283-295.
Wahid2004414	Wahid, S., Smith, J. L., Berry, B., Chewar, C. M., McCrickard, D. S. (2004). “Visualization of design knowledge component relationships to facilitate reuse”. In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IRI 2004)*, Las Vegas, USA, pp. 414-419, November.
Washizaki20061222	Washizaki, H., Takano, S., Fukazawa Y. (2006). “A system for visualizing binary component-based program structure with component functional size”. *WSEAS Transactions on Information Science and Applications*, v. 3, n. 7, pp. 1222-1230.
Yazdanshenas2012143	Yazdanshenas, A. R., Moonen, L. (2012). “Tracking and visualizing information flow in component-based systems”. In: *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC 2012)*, Passau, Germany, pp. 143-152, June.
Ye2000266	Ye, H., Lo, B. W. N. (2000). “A visualised software library: Nested self-organising maps for retrieving and browsing reusable software assets”. *Neural Computing and Applications*, v. 9, n. 4, pp. 266-279.

4. Analysis

The analysis is described in terms of the dimensions presented in Section 2.8 and driven by the corresponding research questions. Each item result is complemented by a brief discussion on the findings. It is important to mention that, in most of the dimensions, a single approach may fit into more than one category, depending on the broadness of its support/features. The detailed description of each approach individually is presented in Appendix B.

The results were also organized in a website (http://www.cos.ufrj.br/~schots/survis_reuse/), in order to allow a better exploration of the findings, as well as establish correlations between the visualization dimensions.

4.1. Visualization approaches supporting software reuse

Regarding the primary question (*PQ: Which visualization approaches have been proposed to support software reuse?*), the identified approaches and tools are listed in Table 11.

Table 11. Visualization approaches (sorted by year) (*PQ*)

Approach/tool name	Publications	Year
Software Landscape	Mancoridis199374	1993
Software Information Base (SIB)	Constantopoulos19951	1995
Program Explorer	Lange1995342	1995
Dotplot Patterns	Helfman199631	1996
N/A	Alonso1998483	1998
Dy-re (Dynamic reuse)	Biddle199992	1999
Dyno	Biddle199992 / Marshall2001 / Marshall2001103	1999/ 2001
Nested Software Self-Organising Map (NSSOM)	Ye2000266	2000
Framework Interaction for REuse (Fire)	Marshall2001103	2001
Visualization Architecture for REuse (VARE), which includes Abstraction Tool (AT), XML Data Storage Environment (XDSE) and Blur	Marshall2001103 / Anslow2004	2004
N/A	Mittermeir200195	2001
N/A	Charters2002765	2002
Test Driver + SpyApp + Transformer[18]	Marshall200381	2003
Spider	Anslow2004 / Marshall200435	2004/ 2004
Claims Exploration of Relationships Visualization (CERVi)	Wahid2004414	2004
TRAceability Pattern Environment (TRAPEd)	Kelleher200550	2005
Visualisation of Execution Traces (VET)	McGavin2006153	2006
Growing Hierarchical Self-Organizing Map (GHSOM)	Tangsripairoj2006283	2006
N/A	Washizaki20061222	2006
DigitalAssets Discoverer	Gonçalves2007872 / Oliveira2007461	2007/ 2007
Gilligan	Holmes2007100	2007
N/A	Stollberg2007236	2007
BARRIO	Dietrich200891	2008
MUDRIK	Ali200950	2009
N/A	Damaeviius2009507	2009
Ontology-Driven Visualization (ODV)	DeBoer200951	2009
NFRs and Design Rationale (NDR) Ontology / Toeska/Review tool	López20091198	2009
AMPLE Traceability Framework (ATF)	Anquetil2010427	2010
Interface Descriptions Management System (IDMS)	Areeprayolkij2010208	2010

[18] Test Driver and Transformer are also mentioned in VARE and Spider as modules of these approaches. According to [Marshall2001103], as opposed to Dyno and Fire, they are not described as a standalone application.

Approach/tool name	Publications	Year
FEATUREVISU	Apel2011421	2011
Variant Analysis	Duszynski2011303 / Duszynski201237	2011/ 2012
API-Dependence Visualization[19]	Bauer2012435	2012
FeatureCommander	Feigenspan20121	2012
FlowTracker	Yazdanshenas2012143	2012

It can be noticed that, in some cases, the same approach is described in more than one publication. Additionally, a single publication may also describe more than one approach. Thus, for associating publications to the approaches, the following criteria are used:

- If a single publication contains all the information of a given approach that is present in other publication(s), the approach is associated (in the analysis) only to that single publication, and, in this case, the other publications are only listed in the results if they present some other approach (otherwise they would be considered as duplicates);
- If a publication contemplates most (but not all) of the information about an approach and there is another publication that presents complementary information (i.e., that is not present in the former), both are included (this is the case of [Anslow2004] and [Marshall200435] regarding Spider, as well as [Duszynski2011303] and [Duszynski201237] regarding Variant Analysis, among others).

Figure 10 shows the distribution of publications per year. It is interesting to note that the use of visualization resources for supporting reuse has been receiving frequent (yet not increasing) attention from the research community over the years, which can be an indicator of its relevance. On the other hand, the topic has not yet been thoroughly explored, since there are at most 4 publications per year (from the ones identified in this study).

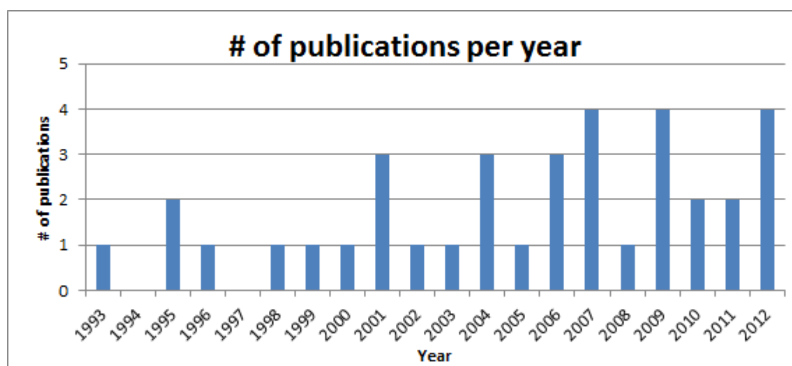


Figure 10. Distribution of selected publications per year

By analyzing the number of publications per author (Figure 11), it can be noticed that several researchers gave some contribution to this research field. Moreover, a research group from the Victoria University of Wellington (New Zealand) is responsible for the largest amount of publications (7), given that, from the 13 authors that published two or more publications in the field, 7 were (or are) from this University. However, according to the findings, the last publication from this research group in the field of this study was in 2006 (see Table 10).

[19] This is not an official name, but one of the screenshots refers to the tool by using this name.

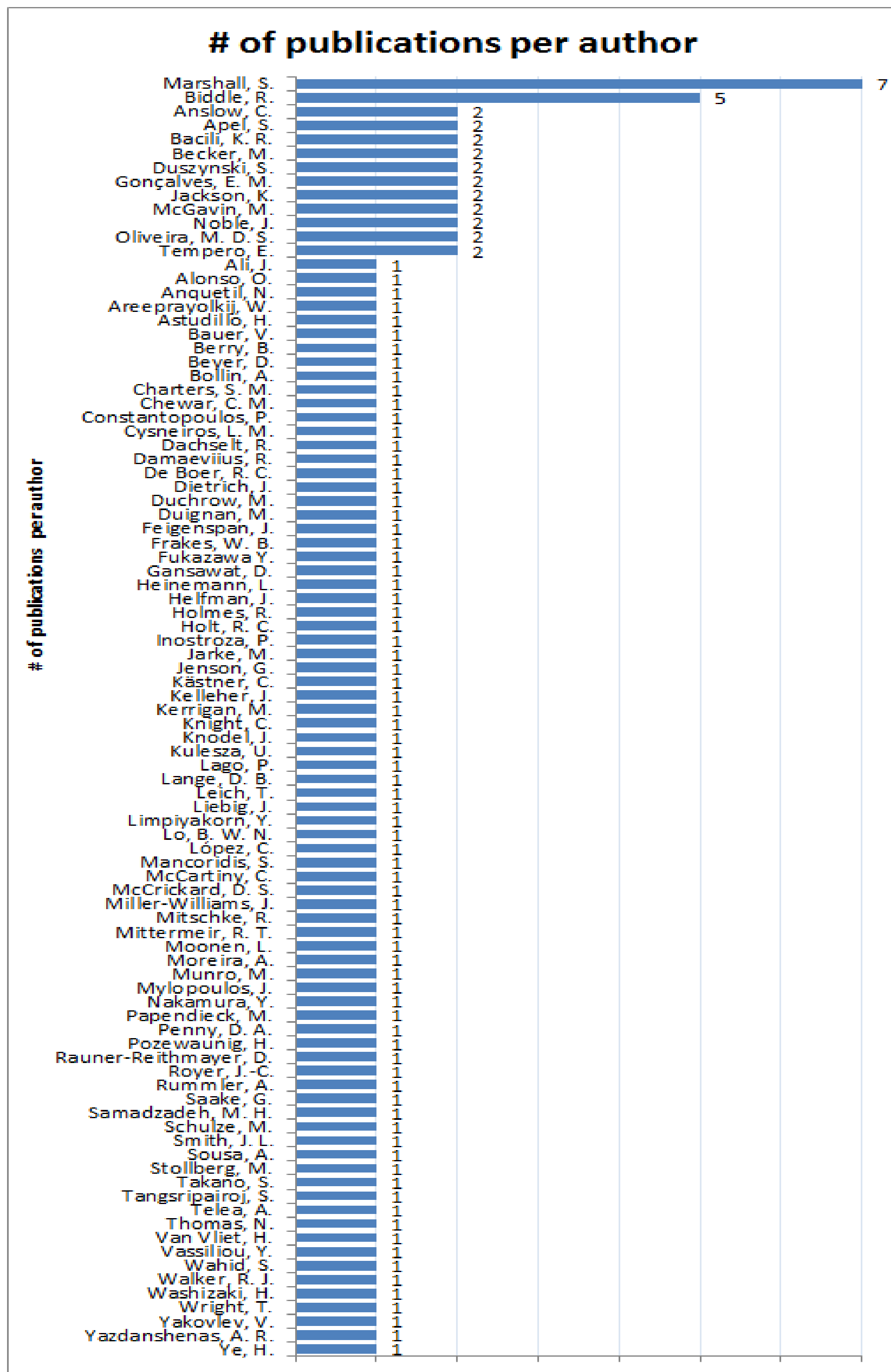


Figure 11. Distribution of selected publications per author

The analysis of the identified approaches is described in the next sections, and the extracted data is presented in Appendix B.

4.2. Task – *why* is the visualization needed?

The first analyzed dimension relates to the reuse goals that led to the proposal of each visualization-based approach. Thus, the findings presented hereafter in this section describe how visualizations aim to support software reuse (*SQ1*). The motivation and assumptions related to each work are presented in Appendix B.

4.2.1. Pioneer works

The earliest work identified aims to (1) make software development information more accessible (by collecting as much of it as possible in one place and by providing uniform visual access to it at the appropriate level of granularity), and (2) provide an intuitive "big picture" understanding for navigating through the software space, thus managing the complexities of large-scale software development for developing and organizing software [Mancoridis199374]. It supports reuse by allowing "to explore, understand, and use the products of software development, including analyses, designs, specifications, and implementations" [Mancoridis199374].

A similar goal is defined in [Constantopoulos19951]: store and manage information about requirements, designs, and implementations of software, also offering facilities for locating and selecting software components, thus broadening and supporting the communication channel between developer and reuser. This work explicitly mentions the reuser role. Through the employed visualization, the work aims to offer valuable assistance to software artifact understanding efforts through the representation and organization of software descriptions, in order to find the software artifacts faster than the time it takes to develop them [Constantopoulos19951].

[Lange1995342] presents a more "programming-oriented" approach: it aims to "make the process of O-O understanding more empirical and realistic by connecting program execution to the understanding of objects and interactions, and the static program information source code to the understanding of classes and their relationships", aiming to demonstrate how patterns can serve as guides in program exploration and thus make the process of understanding more efficient. The work provides class- and object-centered views of the structure and behavior of large C++ systems with information accurate enough to enable programmers to reuse and maintain undocumented parts of these systems [Lange1995342]. It can be seen that both static and dynamic information is taken into account by the approach.

[Biddle199992] also uses static and dynamic information for helping programmers in both programming for and with reuse. In programming for reuse, one of the proposed approaches relates to understanding the structure of the software being developed by the programmers themselves, by the dynamic displaying of the internal structure of the software under development. It aims to make it easy to detect patterns of usage and patterns of dependence within a program – these patterns may help the programmer to determine how best to articulate the structure of a program using components that will be useful and independent for later reuse in other contexts [Biddle199992].

The other proposed approach (related to programming with reuse) is also described in other two publications [Marshall2001] [Marshall2001103], and aims at understanding some dynamic aspects involved in code reuse (e.g., the correct usage and functionality of a component they are considering reusing), in order to better identify potentially reusable components within the structure. Software visualizations (dynamic documentation) are created from executing code (getting all the necessary runtime information) with minimal modifications to the code itself (i.e., "test-drive" by exploring the behavior of a reusable Java component interactively), for providing the developer with a deeper understanding of what the component does, and how it does it, thus helping to decide if and how the component can be reused, making code reuse more appealing [Biddle199992 / Marshall2001 / Marshall2001103].

4.2.2. Other works

Although there are many works in which visualization is the core element (e.g., [Ali200950] and [McGavin2006153]), some of them use it as a final stage of a more complex process for presenting results of an analysis (e.g., [Gonçalves2007872 / Oliveira2007461] and [DeBoer200951]). The next subsections categorize approaches according to their most relevant goals.

- *Identifying reusable assets*

Starting from the premise that pattern languages promote reuse, [Helfman199631] proposes the identification of patterns in software at many different levels of abstraction, ranging in abstraction from the syntax of programming languages to the organizational uniformity of large, multi-component systems [Helfman199631]. [Gonçalves2007872 / Oliveira2007461] aims to provide automatic identification of software components in order to help companies in their reuse and SOA initiatives, bringing to light what companies have already developed by applying reuse indicators with mechanisms to identify artifacts that can be considered as reusable assets. Visualizations are used to evaluate the candidates to become components, helping to inspect a group of applications, configure and trigger the identification mechanisms, tune and reapply them in the analysis process [Gonçalves2007872 / Oliveira2007461].

[Dietrich200891] detects and visualizes clusters in dependency graphs, producing a list of refactorings that can be used to transform programs into a more modular structure, one that is easier to customize and maintain. The work aims to assist software engineers to redraw component boundaries in software, in order to improve the level of reuse and maintainability [Dietrich200891]. [Damaeviius2009507] analyzes software components in a multidimensional feature space, partitioning an initial set of components into groups of similar source code components that can be further used as candidates for generalization (generalization is mainly used for developing reusable software components and reuse libraries). The multidimensional software component feature space is visualized for identifying clusters of similar components as candidates for generalization: the more there are similarities between the generalized components, the better generalization can be achieved, which ultimately allows for better component reuse, library scaling and maintenance [Damaeviius2009507].

[Duszynski2011303 / Duszynski201237] recovers and visualizes information about commonalities and differences that exist in the source code of multiple similar software systems (delivering quantitative information about similarity across system variants) for identifying system parts suitable for transformation into reusable assets and planning necessary

implementation steps (i.e., supporting the reuse potential assessment and the migration to systematic software reuse), besides providing an overview of commonality distribution in the whole analyzed system family, allowing for detailed goal-driven refinement of the analysis results. Visualization is employed to deliver precise quantitative information about the similarity across the analyzed system variants through an abstracted result presentation, in order to assess reuse potential [Duszynski2011303 / Duszynski201237].

- *Organizing software repositories*

Other works aim to support the structuring of software repositories and the retrieval of reusable assets from them. [Ye2000266] aims at making software libraries self-structuring, helping users to predict desired components by providing an intelligible search space for retrieving software assets, giving a whole picture of the library at a relatively general level for finding some interests in certain subareas. [Wahid2004414] aims to browse a repository through visualization by exploiting relationships between units of knowledge (claims), allowing to find the most appropriate reusable knowledge based on design conditions [Wahid2004414]. [Tangsripairoj2006283] organizes and visualizes a collection of reusable software components stored in a software repository aiming to obtain a better insight into the structure of the repository and increase understanding of the relationships among components.

- *Searching and retrieving reusable assets*

[Charters2002765] aims at increasing the understanding of a given code and aid any future development and maintenance, by providing a mechanism in which informed decisions can be made. The work provides an easily navigable environment with a shallow learning curve for non-expert users, allowing them to select components based on multiple attributes and find the ones that could possibly be used in the development of their system [Charters2002765]. [Stollberg2007236] allows to browse and understand available Web services on the level of the problems that can be solved by them, in terms of the structure and the available resources in a domain. Its goal is to aid clients in the goal instance formulation process and allow them to better understand the available resources [Stollberg2007236].

[DeBoer200951] supports the auditors in effectively reusing their know-how and assist the core aspects of their decision making process, namely trade-off analysis, impact analysis, and if-then scenarios. The approach allows to perform a trade-off analysis for determining which quality criteria to include in an audit, select and prioritize the quality attributes to be used in such audit and support the auditor in deciding which quality criteria to use [DeBoer200951].

- *Understanding components and libraries*

[Marshall2001103] aims to support the visualization of framework interactions, which helps to understand how the frameworks are used and aids the identification of the critical interactions between framework and user objects [Marshall2001103]. By storing and retrieving program traces, [Marshall2001103 / Anslow2004] aims to help to understand what a component does, how it works, and whether or not it can be reused in a new program [Marshall2001103 / Anslow2004].

[Anslow2004 / Marshall200435] also provides software visualizations of a component's behavior, complementing other existing documentation. It aims at browsing web-based software repositories to explore existing reusable code components and frameworks by creating visual documentation, thus helping consumers in evaluating a candidate component (by giving them an

insight into the existing behavior as well as possible means of extending that behavior) and producers in advertising their components [Anslow2004 / Marshall200435].

With a broader, general goal, [Marshall200381] aims to help determining if and how a given code component can be reused in the developer's new context, guide a developer's decision as to whether a component is reusable in the developer's current context, and help foster understanding in the developers as to how they could save time and effort through the process of reusing old code in new contexts [Marshall200381].

[Washizaki20061222] aims to help programmers in gaining understanding of a binary component-based program and the overall functional size, as well as whether the break-down and allocation of functionality within the program is appropriate. The long-term goal is to support maintenance activities, i.e., the execution of maintenance tasks (such as fixing bugs or adding extensions) efficiently [Washizaki20061222].

[Ali200950] aims to support the understanding of a potentially large class library (i.e., existing object-oriented systems/class libraries) in a relatively short span of time, allowing programmers to find useful information in the library by helping them understand what is important and relevant, easing to locate and understand appropriate objects [Ali200950].

[McGavin2006153] helps programmers in managing the complexity of execution traces in order to understand code behavior, resulting in more effective software reuse. Dynamic execution of software is visualized, allowing users to interact with and understand the execution traces [McGavin2006153]. [Yazdanshenas2012143] tracks and visualizes information flow in a component-based system at various levels of abstraction, providing source-based evidence that signals from the system's sensors (inputs) trigger the appropriate actuators (outputs). It also aims to support software certification, improving the comprehensibility of such information flow [Yazdanshenas2012143].

Some works focus on understanding software dependencies. [Holmes2007100] aims to help developers to view and navigate through structural dependency information, aimed specifically at pragmatic reuse tasks, and allow developers to record their decisions as they investigate individual dependencies. [Bauer2012435], in turn, analyzes the dependencies of software projects on external APIs, enabling quick insight into how external libraries are used by a project and how complex the dependencies are, besides aiding in decision making regarding library migration scenarios and determining the degree of dependence to its included libraries. Visualizations are used in these works, respectively, to reduce the cognitive effort required while investigating the structural dependencies for a source code fragment (allowing to quickly identify and triage both direct and indirect structural dependencies) [Holmes2007100] and to gain a quick overview of the library dependencies, understanding to which extent a package is dependent on APIs and also how the dependencies of a certain API span over the system architecture [Bauer2012435].

- *Selecting/Integrating components*

[Alonso1998483] presents an architecture and an example application for helping users to understand and compare reusable components and integrate them into applications [Alonso1998483]. [Mittermeir200195] aims at establishing confidence whether a given reusable component satisfies the needs of the intended reuse situation by identifying whether the hidden state of an object (class) satisfies the properties a reuser is expecting from the piece of code at

hand. For supporting the comprehension task, it proposes the combination of software comprehension techniques and technologies [Mittermeir200195].

[López20091198] describes SIGs through an ontology, and represent them as named graphs, enabling their view-based exploration and comparison of decisions and rationales. It facilitates reuse of rationale by allowing architects to understand rationale of previous decisions and/or projects, supporting, for example, the selection between reuse candidates by identifying domain constraints or contexts that are more similar to the problem at hand [López20091198]. [Areeprayolkij2010208] aims to facilitate verifying and reviewing component interfaces for completeness and compatibility and help clustering the components for ordering the sequences of integration plan [Areeprayolkij2010208].

- *Managing traceability*

[Kelleher200550] provides a standardized mechanism for the visualization and communication of reusable traceability practices, while [Anquetil2010427] helps to solve complex traceability problems in feature-oriented software development, by allowing the definition of hierarchical artifact and link types (as well as constraints between these types) in order to observe both the structure of the feature model and the evolution of the realization of the features, also allowing to compare refinement sets of different versions.

- *Developing feature-oriented software and software product lines*

[Apel2011421] visually relates the structural elements of a product line to its features, by visually exploring the structure of product lines, especially regarding feature cohesion, and exploring the reasons for a particular clustering, for example, to get insights into why a feature is not cohesive and how to change that [Apel2011421]. [Feigenspan20121], in turn, allows a programmer to identify feature code at first sight and distinguish code of different features, thus helping to distinguish feature code from base code [Feigenspan20121].

4.2.3. Software engineering activities

Table 12 presents the summarization of the software engineering activities that are addressed by the visualizations (*TQ1.1*).

Table 12. Software engineering activities addressed by visualizations (*TQ1.1*)

Activity	# of approaches	Approaches
Software development with reuse	15	[Constantopoulos19951] [Alonso1998483] [Biddle199992 / Marshall2001 / Marshall2001103] [Ye2000266] [Marshall2001103] [Marshall2001103 / Anslow2004] [Mittermeir200195] [Charters2002765] [Marshall200381] [Anslow2004 / Marshall200435] [Tangsripairoj2006283] [Gonçalves2007872 / Oliveira2007461] [Holmes2007100] [Areeprayolkij2010208] [Yazdanshenas2012143]
Software development for reuse	2	[Biddle199992] [Damaeviius2009507]
Software maintenance	9	[Helfman199631] [Marshall2001103 / Anslow2004] [Marshall200381] [Washizaki20061222] [Dietrich200891] [Damaeviius2009507] [Duszynski2011303 / Duszynski201237] [Bauer2012435] [Feigenspan20121]
Software product line engineering	4	[Anquetil2010427] [Apel2011421] [Duszynski2011303 / Duszynski201237] [Feigenspan20121]
Software design	4	[Lange1995342] [Helfman199631] [Wahid2004414] [López20091198]
Programming / Coding	3	[Lange1995342] [McGavin2006153] [Ali200950]
Analysis / Specification / Requirements engineering	3	[Wahid2004414] [Kelleher200550] [Stollberg2007236]

Activity	# of approaches	Approaches
Quality assurance / Testing / Debugging / Profiling[20]	2	[McGavin2006153] [DeBoer200951]
Software development in general	1	[Mancoridis199374]

Results show different areas of activities grouped into 9 main categories. In a general way, a broad range of software engineering activities is somehow encompassed by the approaches (e.g., maintenance, design etc.). However, it can be noticed that most of the approaches aim at supporting development with reuse (i.e., developing software from reusable assets), while only 2 relates to development for reuse.

In terms of software product line engineering (which can be seen as an advanced step towards systematic reuse), one can note that only more recent approaches (from the ones identified in this study) mention providing some visual support for supporting reuse somehow. Besides the product line development itself, one approach [Duszynski2011303 / Duszynski201237] also prepares for an extractive introduction of the product line paradigm.

A fine-grain analysis on reuse-related tasks that are supported by these visualizations (*TQ1.2*) is presented in Table 13.

Table 13. Reuse-related tasks addressed by visualizations (*TQ1.2*)

Reuse task	# of approaches	Approaches
Understanding assets' structure / asset information / repository	16	[Mancoridis199374] [Constantopoulos19951] [Lange1995342] [Alonso1998483] [Ye2000266] [Mittermeir200195] [Tangsripairoj2006283] [Washizaki20061222] [Holmes2007100] [Stollberg2007236] [Ali200950] [López20091198] [Anquetil2010427] [Apel2011421] [Bauer2012435] [Feigenspan20121]

[20] Software profiling is a method to dynamically analyze software measures in order to optimize the program execution.

Reuse task	# of approaches	Approaches
Understanding assets' behavior	6	[Lange1995342] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [Marshall2001103 / Anslow2004] [McGavin2006153] [Yazdanshenas2012143]
Understanding assets' evolution	1	[Anquetil2010427]
Integrating reusable assets	10	[Lange1995342] [Alonso1998483] [Marshall2001103 / Anslow2004] [Mittermeir200195] [Marshall200381] [Anslow2004 / Marshall200435] [Kelleher200550] [DeBoer200951] [López20091198] [Areeprayolkij2010208] [Yazdanshenas2012143]
Searching and retrieving reusable assets	7	[Constantopoulos19951] [Ye2000266] [Charters2002765] [Wahid2004414] [Stollberg2007236] [Ali200950] [DeBoer200951]
Discovering and evaluating[21] potentially reusable assets	5	[Helfman199631] [Biddle199992] [Gonçalves2007872 / Oliveira2007461] [Damaeviius2009507] [Duszynski2011303 / Duszynski201237]
Restructuring assets for reuse	2	[Dietrich200891] [Apel2011421]

For easing the analysis, a mapping between the steps presented in the reuse-based software development model presented in Figure 1 (proposed by [Kim & Stohr 1998]) and the tasks identified in this study was established. Such mapping is presented in Table 14. Some steps do not have a correspondence (e.g., "specify requirements" and "build new software resources") because they are related to software development in general [Kim & Stohr 1998]. Steps in italic

[21] Evaluating a potentially reusable asset relates to assessing its adequacy to be considered as a reuse asset and to be included in the repository of reusable assets. This can be done in terms of its measured reusability and other applicable organizational factors.

are not reuse-specific (relates to conventional software development), thus they do not contain a corresponding task in this study.

Table 14 . Mapping between the steps in [Kim & Stohr 1998] and the identified tasks

<table>
<tr><th colspan="2">Reuse steps [Kim & Stohr 1998]</th><th>Identified tasks (TQ 1.2)</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">Identify reusable software resources</td><td>Discovering and evaluating potentially reusable assets</td></tr>
<tr><td>Restructuring assets for reuse</td></tr>
<tr><td>2</td><td>Classify reusable software resources</td><td>Discovering and evaluating potentially reusable assets</td></tr>
<tr><td>3</td><td>Specify requirements</td><td>N/A</td></tr>
<tr><td>4</td><td>Retrieve reusable software resources</td><td>Searching and retrieving reusable assets</td></tr>
<tr><td rowspan="3">5</td><td rowspan="3">Understand reusable software resources</td><td>Understanding assets’ structure / asset information / repository</td></tr>
<tr><td>Understanding assets’ behavior</td></tr>
<tr><td>Understanding assets’ evolution</td></tr>
<tr><td>6</td><td>Modify reusable software resources</td><td>Integrating reusable assets</td></tr>
<tr><td>7</td><td>Build new software resources</td><td>N/A</td></tr>
<tr><td>8</td><td>Integrate software resources</td><td>Integrating reusable assets</td></tr>
</table>

Since most of the identified approaches that aim at identifying reusable resources also provide some means to classify them, these steps are mapped to the same task. This also applies to the steps of modifying and integrating reusable resources. Due to the focus of this study, the step of understanding reusable assets was decomposed in understanding the structure, behavior and evolution (in accordance with [Diehl 2007]), so that these characteristics can be analyzed separately.

Some of the approaches aim at supporting the understanding of the structure of an asset or a software repository in order to facilitate reuse. Examples on the structure of an asset include (i) source code views, which may be class- and object-centered [Lange1995342] [Ali200950] or focused on structural dependencies [Holmes2007100], and (ii) software components [Alonso1998483] and their information (e.g., functional size [Washizaki20061222] or library dependencies [Bauer2012435]). Feature models’ structure [Anquetil2010427] is also targeted for a better understanding on product-line engineering, as regards to measures (such as feature cohesion [Apel2011421]) or for distinguishing feature code from base code [Feigenspan20121]. Other kinds of structural analysis of assets include web services [Stollberg2007236], rationale of previous decisions and/or projects [López20091198].

Repository structure comprehension approaches [Ye2000266] [Tangsripairoj2006283] aim to provide a whole picture of the library for finding some interests in certain subareas, and obtain a better insight into the structure of a software repository and, in some cases, increase the understanding of the relationships among software components [Ye2000266] [Tangsripairoj2006283].

Regarding asset’s behavior, in general, the approaches provide ways to better understand how a chosen asset may work. With this goal, [Lange1995342], [Biddle199992 / Marshall2001 / Marshall2001103], [Marshall2001103 / Anslow2004] and [McGavin2006153] support the

comprehension of code behavior. With a broader scope, [Marshall2001103] aims to help understanding how frameworks are used.

Only one approach deals with understanding evolution [Anquetil2010427], with respect to the realization of features, allowing to compare the refinement sets of different versions.

There are approaches oriented to supporting on determining whether and how an asset can be reused in the developer's current context, i.e., establishing confidence whether a given reusable component satisfies the needs of the intended reuse situation), e.g., [Mittermeir200195] and [Marshall200381]. Such goal is also used by [Anslow2004 / Marshall200435] in order to evaluate candidate components to be chosen. Some of these approaches aim at identifying domain constraints or contexts that are more similar to the problem at hand (e.g., [López20091198]). Particularly, in [Marshall200381], the authors deal with the cost of understanding reusable components, but they state that other costs (such as the time to search for potential candidate components for reuse) should also be addressed [Marshall200381].

There are approaches aiming to allow finding reusable assets faster than the time it takes to develop them. For instance, [Ye2000266] proposes mechanisms to support the retrieval of reusable assets in a software library, and [DeBoer200951] aims at supporting an auditor in deciding which quality criteria to include in an audit, through selection and prioritization of quality attributes. Other approaches are geared to discovering and evaluating assets that may be potentially reusable. For example, [Gonçalves2007872 / Oliveira2007461] evaluates the candidates to become components, inspecting a group of applications, while [Duszynski2011303 / Duszynski201237] delivers quantitative information about the similarity across analyzed system variants, in order to assess reuse potential.

Finally, 2 approaches support the identification of actions that may improve assets' reusability (in terms of increasing their chances to be reused) by restructuring them. [Dietrich200891] assists to redraw component boundaries in software, in order to improve the level of reuse and maintainability. [Apel2011421] allows exploiting the reasons for a particular clustering to get insights into why a feature is not cohesive and how to change that.

4.2.4. Discussion

Regarding the approach goals and motivations (*SQ1*), ever since the first identified works were published, there was already a concern on supporting reuse of a variety of artifacts [Mancoridis199374] [Constantopoulos19951]. None of the identified visualization approaches aims to support the understanding of the dynamics of software reuse in terms of software projects, assets, users, and the relationship between these core elements (i.e., not only understanding assets in terms of their properties and metadata, but also how they are being used and maintained, in order to increase reuse confidence). In other words, their goals are mostly artifact-oriented.

Although approaches somehow encompass many software engineering activities (*TQ1.1*), only a few of them present integration among different activities. Another concern about the approaches is the lack of integration with development environments that can provide interfaces to other activities.

There is support for a variety of reuse tasks (*TQ1.2*), and understanding assets is by far the most supported one. This is indeed expected, since understanding is a likely benefit in

employing visualizations. Nevertheless, research on using visualization for restructuring assets for reuse is still much underexplored, since the identified works are very specific, with limited customizability. Other tasks miss evidence on their support.

In terms of the different aspects that can be the focus of comprehension, most approaches support the understanding of an asset's structure, and some help understanding their behavior. Evolution information about reusable assets are a particular absence of existing works – the only related work deals with comparing refinement sets of different versions of feature models, and it is based on a trace repository; no other evolution aspects are taken into account by any approach. Moreover, repository-related information is only focused on structural characteristics, i.e., usage data related to reuse repositories are not handled by the approaches.

Finally, approaches in general do not provide a modular architecture (regarding data sources and visualization abstractions) for supporting different stages of the data extraction and analysis process (such as [Anslow2004]). Such modularity might offer more flexibility to them, allowing to add new functionalities and, consequently, providing more support for software engineering activities and tasks.

4.3. Audience – *who* will use the visualization?

Figure 12 and Table 15 present the stakeholders to which each visualization approach is intended/targeted (*SQ2*). Among the approaches that aim to support programmers, not all of them mention explicitly the "reuser" role. Thus, whenever a publication seems to differ between common programmers and "reusers", it was decided not to merge these roles into a single role representation, as some approaches may have broader goals not related to reuse, and such goals may be indeed targeted to different roles. Moreover, reuse possibilities go beyond the programmer role.

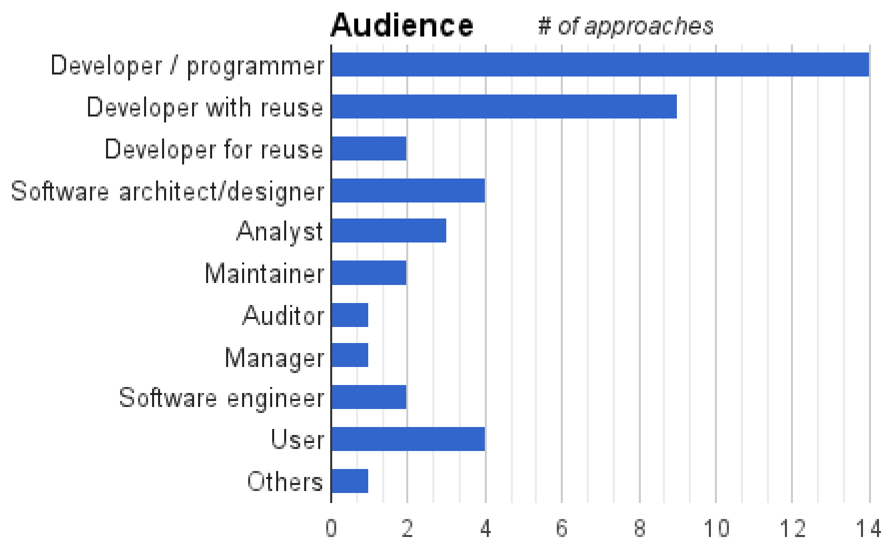


Figure 12. Approaches and supported stakeholders (*SQ2*)

Table 15. Approaches and supported stakeholders (*SQ2*)

Visualizations' audience	# of approaches	Approaches
Developer / programmer	14	[Mancoridis199374] [Constantopoulos19951] [Lange1995342] [Marshall2001103] [Marshall2001103 / Anslow2004] [Mittermeir200195] [McGavin2006153] [Tangsripairoj2006283] [Washizaki20061222] [Ali200950] [Damaeviius2009507] [Anquetil2010427] [Apel2011421] [Yazdanshenas2012143]
Developer with reuse (consumer)	9	[Constantopoulos19951] [Biddle199992 / Marshall2001 / Marshall2001103] [Mittermeir200195] [Charters2002765] [Marshall200381] [Anslow2004 / Marshall200435] [Holmes2007100] [Stollberg2007236] [Feigenspan20121]
Developer for reuse (producer)	2	[Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103]
Software architect/designer	4	[Mancoridis199374] [Helfman199631] [Wahid2004414] [López20091198]
Analyst	3	[Mancoridis199374] [Kelleher200550] [Gonçalves2007872 / Oliveira2007461]
Maintainer	2	[Bauer2012435] [Feigenspan20121]
Auditor	1	[DeBoer200951]
Manager	1	[Mancoridis199374]
Software engineer	2	[Mittermeir200195] [Dietrich200891]

Visualizations' audience	# of approaches	Approaches
User	4	[Alonso1998483] [Ye2000266] [Areeprayolkij2010208] [Duszynski2011303 / Duszynski201237]
Others / non-related to software development	1	[Yazdanshenas2012143]

From the results, it can be noticed that programmers are the most supported stakeholders by far. 14 approaches mention programmers in general, while other 9 support specifically developers with reuse (also referred to as "reusers" or "system integrators"). Only 2 approaches mention support for developers for reuse: one focuses on programmers [Biddle199992] and the other is geared to component writers who wish to create visual documentation of their own components [Biddle199992 / Marshall2001 / Marshall2001103]. Although in [Damaeviius2009507] development for reuse is also supported (as can be seen in Table 13), such role is not explicitly specified.

Other supported roles include software architects/designers [Mancoridis199374] [Helfman199631] [Wahid2004414] [López20091198], analysts [Mancoridis199374] [Kelleher200550] [Gonçalves2007872 / Oliveira2007461], maintainers [Bauer2012435] [Feigenspan20121], auditors [DeBoer200951] and managers [Mancoridis199374]. The latter states that the visualization is targeted to developers (the presented views refer to software development technical details), but points out some actions that could eventually be performed by managers. However, it only presents low-level technical details on a software project.

Some approaches mention a more general role, such as software engineer [Mittermeir200195] [Dietrich200891] or user [Alonso1998483] [Ye2000266] [Areeprayolkij2010208] [Duszynski2011303 / Duszynski201237]. One approach aims to support a role non-related to software development ([Yazdanshenas2012143], geared to safety domain experts).

4.3.1. Discussion

Although there is a reasonable variety of stakeholder support (*SQ2*), only a few works support more than one stakeholder simultaneously. This would not be a major problem if different approaches could communicate with each other (but this is not the case, as discussed in Section 4.7.1). Thus, the lack of a multi-stakeholder approach hampers the evaluation of how well organization's goals related to reuse are being accomplished, under the perspectives of each reuse stakeholder.

Particularly, because the one approach that mentions some support for managers only presents technical details on a software project, it does not seem feasible for the reality of project management. Managers need more high-level details that can be useful for decision making, so that they can promote actions not only to stimulate reuse, but especially to mitigate potential barriers to reuse in their organizations.

4.4. Target – *what* is the data source to represent?

Figure 13 and Table 16 summarize which items/data are visually represented (*SQ3*).

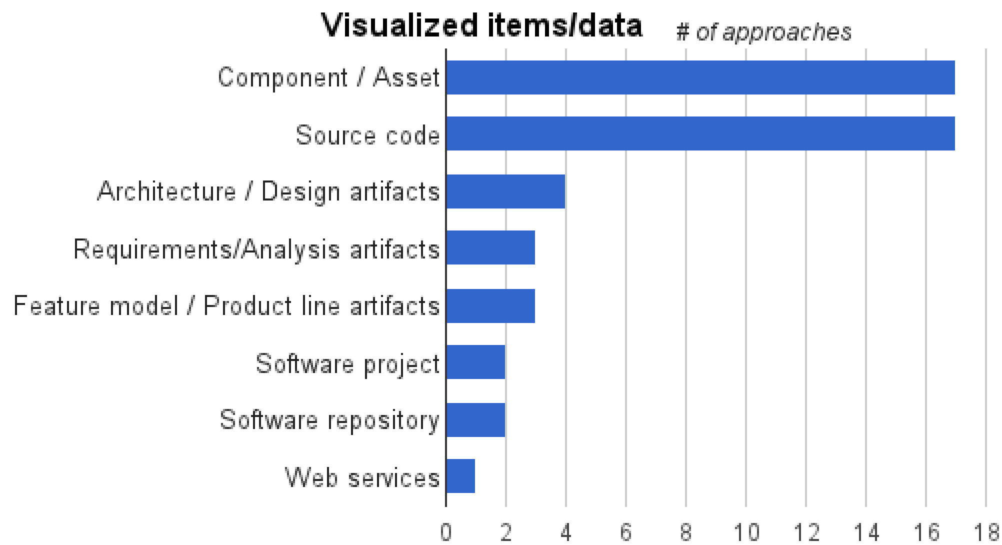


Figure 13. Visualized items/data by approach (*SQ3*)

Table 16. Visualized items/data by approach (*SQ3*)

Visualized items/data	# of approaches	Approaches
Component / Asset and related information	17	[Mancoridis199374] [Alonso1998483] [Ye2000266] [Marshall2001103] [Marshall2001103 / Anslow2004] [Mittermeir200195] [Charters2002765] [Marshall200381] [Anslow2004 / Marshall200435] [Tangsripairoj2006283] [Washizaki20061222] [Gonçalves2007872 / Oliveira2007461] [Ali200950] [Damaeviius2009507] [Areeprayolkij2010208] [Bauer2012435] [Yazdanshenas2012143]

Visualized items/data	# of approaches	Approaches
Source code and related information	17	[Mancoridis199374] [Constantopoulos19951] [Lange1995342] [Helfman199631] [Alonso1998483] [Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Mittermeir200195] [Charters2002765] [McGavin2006153] [Gonçalves2007872 / Oliveira2007461] [Holmes2007100] [Dietrich200891] [Ali200950] [Duszynski2011303 / Duszynski201237] [Feigenspan20121] [Yazdanshenas2012143]
Architecture / Design artifacts[22] and related information	4	[Biddle199992 / Marshall2001 / Marshall2001103] [Wahid2004414] [DeBoer200951] [López20091198]
Requirements/Analysis artifacts and related information	3	[Mittermeir200195] [Kelleher200550] [Stollberg2007236]
Feature model / Product line artifacts and related information	3	[Anquetil2010427] [Apel2011421] [Feigenspan20121]
Software project and related information	2	[Mancoridis199374] [Bauer2012435]
Software repository and related information	2	[Ye2000266] [Charters2002765]
Web services and related information	1	[Stollberg2007236]

There is a variety of approaches that visually represent reusable software assets – also referred to as components or libraries[23], usually related to implementation artifacts (but not always [Tangsripairoj2006283]) –, or even components suggested as reusable [Gonçalves2007872 / Oliveira2007461]. Some approaches allow for binary components

[22] Includes architectural/design knowledge.

[23] The term “library” is used in the publications with different meanings: they can be related to a component (e.g., [Bauer2012435]) or a repository (e.g., [Ye2000266]).

provided in byte-code format (e.g., [Washizaki20061222]). Related information on these components/assets include static and dynamic information present in a component (e.g., [Marshall200381]), such as dependency relationships between components [Washizaki20061222] [Gonçalves2007872 / Oliveira2007461], semantic relationships among software components [Ye2000266], their functional size (an indication of the amount of functionality provided by them) [Washizaki20061222], and so on.

Some approaches provide detailed, fine-grained information on the represented components and libraries. In [Ali200950], for instance, object-oriented information is presented, such as referential relationships among classes of the library, parameter(s) or method(s) returned value, inheritance tree of all the classes, amongst others. In [Yazdanshenas2012143], in turn, the focus is on information regarding the intercomponent and intra-component information flow (dependencies between a component's input and output ports, dependencies between system-level inputs and outputs, and so on).

Source code may be visualized in terms of its lines of code [Helfman199631] [Yazdanshenas2012143] as well as its related information, which can be either static [Ali200950] or dynamic [Biddle199992 / Marshall2001 / Marshall2001103], e.g., method calls between objects and how objects interact [McGavin2006153]. Dependencies are also illustrated, as structural relationships between classes [Gonçalves2007872 / Oliveira2007461] or semantic relationships between objects [Constantopoulos19951]. Clusters of source code and byte code can be created based on their dependencies [Dietrich200891].

Another aspect is the analysis of commonalities and variabilities in the source code of multiple software systems [Duszynski2011303 / Duszynski201237], in order to separate feature code from base code or to analyze the percentage of each feature in the source code file [Feigenspan20121].

Approaches that visually represent architecture and design elements include UML diagrams [Biddle199992 / Marshall2001 / Marshall2001103], reusable design knowledge claims and their relationships [Wahid2004414], and quality attributes of interest (e.g., quality criteria relevant to a given audit, including their hierarchy and relations [DeBoer200951] or architecture rationale represented by softgoal interdependencies [López20091198]). Requirements/analysis and related information include specifications [Mittermeir200195], goal templates [Stollberg2007236] and information related to requirements and traceability items [Kelleher200550], including relationships between them and requirement attributes, such as status and priority.

Some approaches visually represent feature models [Feigenspan20121], dependency relations between elements of features [Apel2011421] and trace information [Anquetil2010427], including the name and type of each link/artifact, time links related to a product, and versions of artifacts which have evolved. Software repositories (also referred to as "software component repositories" or "software libraries") are only represented statically [Ye2000266] [Charters2002765]. As stated in Section 4.2.4, no usage data information related to reuse repositories could be found in the analyzed publications. Software project representations include their hierarchical composition [Bauer2012435] or their sub-systems, projects, and libraries [Mancoridis199374]. Web service information includes available Web services and their usability in a problem domain with respect to the goals that can be solved by them [Stollberg2007236].

The identified items that are visually represented reveal a variety of selected sources (*TQ3.1*). However, a common category of source is related to a specific type of repository, such as a software library or a simple database to store data about components. One example of this kind of source is presented in [Helfman199631], in which the code is used together with a database including information about moments and causes of creations and changes in C code.

Beyond the source code, other artifacts are used as data source. In [Mittermeir200195], the approach is based on test logs. Binary files serve other mechanisms as well, such as Java byte-code in [Marshall200381], [Anslow2004 / Marshall200435], [Washizaki20061222], [Dietrich200891] and [Ali200950] approaches. Software behavior also provides sources for exploration. [McGavin2006153]'s approach uses information stored in an execution trace format as a XML file, while [Marshall2001103 / Anslow2004] uses events during runtime to extract data.

Despite the different sources and approach goals, it is noteworthy that most of the approaches directly use the source code as a means to obtain the data. [Helfman199631], [Mittermeir200195], [Marshall2001103 / Anslow2004], [Gonçalves2007872 / Oliveira2007461], [Dietrich200891], [Damaeviius2009507], [Bauer2012435], [Feigenspan20121] and [Yazdanshenas2012143] are examples of approaches that base their processing on the code itself.

In order to gather data and visually represent them, approaches use different collection procedures (*TQ3.2*). For instance, [Washizaki20061222] extracts data from Java binary files by applying byte-code analysis and uses Java reflection for obtaining the dependency relationships in the components. Similarly, using JAR or binary (.class) files as input, the approach presented in [Ali200950] loads all the classes, interfaces and packages stored in the library, and the system analyzes the loaded entities and collects detailed information about them.

Combining source code and byte-code, [Dietrich200891] proposes a separation of a graph in clusters using the betweenness measure for edges, which indicates the shortest paths between all pairs of nodes in the graph passing through that edge. In some cases, when the data source is a software repository or database, the collection is made by user queries through a search system [Alonso1998483] [Wahid2004414].

4.4.1. Discussion

The vast majority of the visualized items and data (*SQ3*) are source code artifacts (object-oriented entities, such as classes and relationships, or software components). In spite of this imbalance, there are many different kinds of artifacts (from different software development stages) that can be visualized.

There are few approaches for visualizing software repositories with the intention to promote reuse (providing relevant reuse data), and no repository information or metadata are visually represented as a means of awareness.

One can observe that the data sources (*TQ3.1*) are usually the source code of a program and databases. Only a few approaches combine information from different sources (e.g., [Kelleher200550]), and some are compatible with a limited set of data types. Although several kinds of information may be used for supporting reuse, some common data sources are not explored by any of the works (e.g., version control system repositories, issue trackers etc.).

Moreover, although many assets have additional related data available online, such data are usually underexplored or overlooked.

Since each visualization technique may have some constraints, each collection procedure (*TQ3.2*) must deal with this issue and make the proper arrangements. For instance, in [Kelleher200550] some format conversions are mentioned in order to make the data ready to be represented by the intended visualization. During the data collection procedure, the source may still require some transformations to have the data set in the correct format to be used by different representations. Some authors also defend the use of intermediary formats for storing the collected data [Alonso1998483] [Anslow2004 / Marshall200435] in order to make them reusable in different visualizations.

4.5. Representation – *how* to represent the data?

Particularly for this dimension, it must be emphasized that all the considerations are based solely in the information presented in the selected publications. Some interpretations were made only for categorizing purposes.

In order to make the data representation more intelligible, some visual metaphors are adopted in these works trying to look for more appropriate ways to communicate the results. Table 17 shows which visualization metaphors are used for representing the items/data (*SQ4*).

Table 17. Visualization metaphors employed by the approaches (*SQ4*)

Visualization metaphors	# of approaches	Approaches
Network / Graph	13	[Mancoridis199374] [Constantopoulos19951] [Lange1995342] [Mittermeir200195] [Wahid2004414] [Gonçalves2007872 / Oliveira2007461] [Stollberg2007236] [Dietrich200891] [López20091198] [Anquetil2010427] [Areeprayolkij2010208] [Apel2011421] [Yazdanshenas2012143]
Hierarchy	10	[Alonso1998483] [Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Ye2000266] [Tangsripairoj2006283] [Holmes2007100] [Ali200950] [DeBoer200951] [Anquetil2010427] [Bauer2012435]

Visualization metaphors	# of approaches	Approaches
Diagrams	6	[Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [Marshall2001103 / Anslow2004] [McGavin2006153] [Yazdanshenas2012143]
Matrix / Matrix-like	5	[Helfman19963] [McGavin2006153] [DeBoer200951] [Bauer2012435] [Yazdanshenas2012143]
Geometric forms	5	[Washizaki20061222] [Holmes2007100] [Duszynski2011303 / Duszynski201237] [Bauer2012435] [Feigenspan20121]
Map	4	[Ye2000266] [Kelleher200550] [Tangsripairoj2006283] [Damaeviius2009507]
Real world metaphor	2	[Mancoridis199374] [Charters2002765]
Others	2	[Biddle199992 / Marshall2001 / Marshall2001103] [McGavin2006153]
N/A	2	[Marshall200381] [Anslow2004 / Marshall200435]

The selected approaches show a major adoption of the network/graph and hierarchical metaphors for representing the data. Among the hierarchy metaphors used, tree representations are the most common ones (e.g., hyperbolic tree [Alonso1998483], sideways tree [Biddle199992], tree lists and tables [Holmes2007100] [Bauer2012435] and interactive cone tree [Ali200950]).

It is also noteworthy to highlight the fact that the diagram metaphor chosen by some approaches make use of UML diagrams [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [Marshall2001103 / Anslow2004]. Additionally, some real world metaphors were used by 2 approaches [Mancoridis199374] [Charters2002765], presenting the results with cityscape, world, country, city, districts, streets, buildings etc.

Regarding the way the data are mapped to the visualizations (*TQ4.1*), the following mapping categories were identified:

- *Colors*: Different colors indicate different items or different properties of an item;

- *Nodes and edges/arrows*: Edges/arrows are used for indicating whether there is a relationship between the items described as nodes – in some cases, different edge/arrow formats may also indicate the kind of relationship;
- *Position*: Positioning of elements provides information regarding a given measure: for instance, it may indicate a particular value associated to an item (e.g., when related to Cartesian axes) or whether a given property applies to an item or not (e.g., when items are depicted close to each other);
- *Size/dimension*: The length, width and/or height of an item may be proportional to the measure of one or more of its properties;
- *Brightness/contrast*: Similar to the color mapping, but in this case, there is an underlying scale whose ends usually relate to a given amount of a measure;
- *Ordering*: This can be seen as a particular case of position, in which items are disposed in ascending or descending order of a given measure or property;
- *Icon*: Different visual representations usually indicate different items or item properties;
- *Containment*: The fact that an item is contained inside another may indicate some kind of relationship between them (e.g., hierarchical);
- *Density*: The more items are close to each other, the more a given property applies to them;
- *General/customizable*: In this case, the user can customize the mapping between data/items and visual representations;
- *N/A*: No mapping was identified during the analysis.

Table 18 shows a mapping between approaches and these graphical representation resources. More detailed information for each approach can be found in Appendix B.

Table 18. Data-to-visualization mapping (*TQ4.1*)

	COLOR	NODES AND EDGES/ARROWS	POSITION	SIZE/DIMENSION	BRIGHTNESS/CONTRAST	ORDERING	ICON	CONTAINMENT	DENSITY	GENERAL/CUSTOMIZABLE	N/A	TOTAL
[Mancoridis199374]		X			X			X	X			4
[Constantopoulos19951]	X	X										2
[Lange1995342]		X	X	X		X						4
[Helfman199631]			X		X							2
[Alonso1998483]										X		1
[Biddle199992]	X	X	X	X								4
[Biddle199992 / Marshall2001 / Marshall2001103]										X		1
[Ye2000266]			X					X				2
[Marshall2001103]											X	1
[Marshall2001103 / Anslow2004]										X		1
[Mittermeir200195]	X	X										2
[Charters2002765]	X		X	X				X				4
[Marshall200381]											X	1
[Anslow2004 / Marshall200435]											X	1
[Wahid2004414]	X											1
[Kelleher200550]		X										1
[McGavin2006153]	X	X	X		X							4
[Tangsripairoj2006283]			X	X								2
[Washizaki20061222]	X	X		X								3
[Gonçalves2007872 / Oliveira2007461]											X	1
[Holmes2007100]	X				X		X					3
[Stollberg2007236]		X	X									2
[Dietrich200891]		X					X					2
[Ali200950]	X		X	X		X	X					5
[Damaeviius2009507]			X									1
[DeBoer200951]	X											1
[López20091198]							X					1
[Anquetil2010427]	X	X	X									3
[Areeprayolkij2010208]		X										1
[Apel2011421]	X	X	X	X								4
[Duszynski2011303 / Duszynski201237]	X			X								2
[Bauer2012435]	X		X	X		X						4
[Feigenspan20121]	X			X		X						3
[Yazdanshenas2012143]	X											1
TOTAL	16	13	13	10	4	4	4	3	1	3	4	

Colors are the most frequent visual resource to represent variations of data. Nodes and arrows usually map to concrete artifacts and their relations, respectively. Position (along an axis) and dimensions (e.g., height) represent a given aspect of software. In [Apel2011421], a feature has a higher cohesion than coupling if its elements are close to each other; [Bauer2012435] uses the width of the colored bars to indicate the total number of API calls.

Other identified attributes include ordering ([Lange1995342] [Ali200950] [Bauer2012435] [Feigenspan20121]) and density ([Mancoridis199374]). There are also approaches that use different icons to distinguish software elements and their properties ([Holmes2007100] [Dietrich200891] [Ali200950] [López20091198]). Some approaches offer a custom mapping between visual elements and data ([Alonso1998483] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103 / Anslow2004]).

Finally, as regards to the visualization strategies and techniques employed (*TQ4.2*), Table 19 shows a mapping between approaches and such strategies. More detailed information can be found in Appendix C, and individual information of the approaches is presented in Appendix B.

Table 19. Visualization strategies and techniques employed (*TQ4.2*)

	Selection	Browsing / Navigation	Browsing / Querying	Details on demand / Drill-down	Details on demand / Labeling	Details on demand / Labeling / Tooltip	Clustering	Filtering	Filtering / Highlighting/Mitigation	Filtering / Tuning/Tweaking	Filtering / Inclusion/Removal	Filtering / Collapse/Expand	Overview+detail	Layout	Layout / 3D	Zooming / Geometric	Zooming / Semantic	Panning	Panning / Drag-and-drop	Hierarchical visualizations	Animation	Sorting	Rotate	Presentation	Presentation / Simultaneous	Overlap / Flipping	Overlap / Transparency	Linking	Focus+context	N/A	TOTAL
[Mancoridis199374]		X		X	X				X		X		X																		6
[Constantopoulos19951]	X	X	X		X			X																							5
[Lange1995342]	X	X										X													X						4
[Helfman199631]							X						X									X									3
[Alonso1998483]	X	X		X					X										X							X			X		7
[Biddle199992]								X		X		X		X							X	X			X		X				8
[Biddle199992 / Marshall2001 / Marshall2001103]	X							X		X											X				X			X			6
[Ye2000266]	X	X	X	X	X		X		X				X																		8
[Marshall2001103]										X	X								X		X				X						5
[Marshall2001103 / Anslow2004]		X							X	X		X				X															5
[Mittermeir200195]				X					X					X							X			X							5
[Charters2002765]	X	X		X			X						X	X																	6
[Marshall200381]																														X	1
[Anslow2004 / Marshall200435]											X																				1
[Wahid2004414]	X	X		X				X																							4
[Kelleher200550]			X				X	X														X									4
[McGavin2006153]		X		X						X	X		X				X								X	X					8
[Tangsripairoj2006283]							X						X							X											3
[Washizaki20061222]															X	X							X								3
[Gonçalves2007872 / Oliveira2007461]	X	X					X		X	X	X																				6
[Holmes2007100]	X	X		X	X			X				X													X				X		8
[Stollberg2007236]	X	X		X	X						X		X	X		X			X	X			X								11
[Dietrich200891]	X				X		X		X		X			X		X		X	X		X										10
[Ali200950]	X							X	X				X	X	X	X							X	X		X		X			11
[Damaeviius2009507]							X																								1
[DeBoer200951]	X					X			X										X						X			X			6
[López20091198]					X				X																						2
[Anquetil2010427]	X	X		X	X				X					X																	6
[Areeprayolkij2010208]							X																								1
[Apel2011421]						X	X			X				X		X			X												6
[Duszynski2011303 / Duszynski201237]							X																								1
[Bauer2012435]				X								X																			2
[Feigenspan20121]	X			X					X	X		X	X							X		X			X						9
[Yazdanshenas2012143]	X	X		X	X		X		X			X								X				X							9
TOTAL	16	14	3	13	9	2	12	7	13	8	7	7	9	8	2	6	1	1	6	4	5	4	3	3	8	3	1	3	2	1	

Looking for ways to represent the data in an intuitively composition, a large variety of visualization techniques are employed[24]. From the table results, it is possible to reinforce the importance of techniques related to selection, browsing/navigation, drill-down, clustering and filtering by highlighting/mitigation, the most frequently used strategies among the results.

From the works that allow panning (e.g., [Dietrich200891]), some provide support for drag and drop (e.g., [Alonso1998483], [Marshall2001103] and [Stollberg2007236]). Regarding zoom interactions, the most frequent use is the geometric zooming (i.e., enlarging objects while zooming in and shrinking them while zooming out [Buering et al. 2006]). Examples include [Marshall2001103 / Anslow2004], [Washizaki20061222] and [Apel2011421]. In contrast, semantic zooming – which shows different visual representations to information items according to the available space based on zooming interactions [Buering et al. 2006] – was only found in [McGavin2006153].

Focus + context techniques integrates focus and context into a single display where all parts are concurrently visible and the focus is displayed seamlessly within its surrounding context [Cockburn et al. 2008] (e.g., [Alonso1998483] and [Holmes2007100]). The overview + detail technique, in turn, is characterized by the simultaneous display of both an overview and detailed view of an information space, each in a distinct presentation space [Cockburn et al. 2008] (e.g., [Helfman199631], [Ye2000266] and [Charters2002765]). According to Table 19, the latter technique is more frequently than the former. Browsing interactions can be performed through navigation (e.g., [Wahid2004414] and [Anquetil2010427]) or querying (e.g., [Constantopoulos19951] and [Ye2000266]). Regarding presentation techniques, some works present different, multiple views simultaneously (e.g., [Lange1995342]).

Filtering is applied by collapsing/expanding a set of elements (e.g., [Holmes2007100] and [Bauer2012435]), by including/removing an element from the view (e.g., [Mancoridis199374] and [Gonçalves2007872 / Oliveira2007461]), by highlighting items of interest (e.g., [Ye2000266], [López20091198] and [Yazdanshenas2012143]) or by tuning/tweaking (through parameters customization) (e.g., [Marshall2001103] and [Feigenspan20121]). Some approaches offer more than one kind of filtering (e.g., [Biddle199992]).

Drill-down techniques can be applied in order to reveal details on demand, as in [Alonso1998483], in which the user can select a visual item for revealing its source code. An alternative way for exploration of details on demand is through labeling (e.g., [Dietrich200891]). Labeling can also be presented as tooltips (e.g., [Apel2011421]).

Among overlap-based techniques, flipping (e.g., [Alonso1998483]) and transparency (e.g., [Biddle199992]) are used. Other visualization techniques employed include rotating (e.g., [Washizaki20061222]), sorting (e.g., [Helfman199631] and [Kelleher200550]) and linking (e.g., [DeBoer200951]), among others.

Some cognitive-driven decisions also lead to some visualization strategies. In order to prevent matches between frequent tokens from saturating the plot, [Helfman199631]'s approach resorts to reconstruction methods and inverse-frequency weighting for displaying matches from more than one pair of tokens in a single pixel. In [Charters2002765], monuments are placed in

[24] A detailed explanation of these visualization strategies and techniques is available in [Vasconcelos et al. 2014]

the center and at each corner of Component City aiding users in their construction of a cognitive map. [Holmes2007100]'s tool allows to quickly traverse through a series of dependencies without getting lost; moreover, the nodes retain their parents so their origin can be easily seen.

A strategy that also supports cognitive interactions is to keep track of user's actions and showing to them as a history. In [Constantopoulos19951], a history list contains the names of the objects selected as current during a session in chronological order. In [Ye2000266], the system records the process of users' navigation, and some marks are drawn on the accessed maps to trace the users' action so that they can easily identify where they are and which region in the map has been explored.

4.5.1. Discussion

Regarding the representation of data (*SQ4*), as expected, different abstractions are used for representing different data. Although several types of abstractions are used, publications lack a discussion on how/why a given metaphor was chosen and, more importantly, whether it is effective or not in its purpose. The mapping between data and visualizations (*TQ4.1*) is barely described in most of the publications, so the reader/user has to "guess" it, which can be risky and lead to wrong interpretations of data.

Moreover, although several visualization strategies and techniques are used (*TQ4.2*), only a few approaches make a comprehensive use of them. This does not mean that every possible technique should be employed, but some approaches may require more interaction facilities for allowing an effective understanding of data.

There is a lack of mechanisms that offer flexibility to software stakeholders in customizing their visualizations, so one can focus on relevant data and information to improve the understanding of their activities (as stated in [Silva et al. 2012]). Although this can be seen as a downside, on the other hand, letting the user decide which visualization to use may not be adequate, as he/she may not know which metaphors better fit the structure to be visualized.

In [Marshall2001], for instance, the user must map visualizations to data, although the amount of required mapping information that the user needs to supply was intended to be minimized. The author recognizes that this can be a problem, as the purpose is to understand the component, and "a developer may not know enough (...) which methods should map to which sequence". However, there is no tool support for this task. According to the author, "it is bordering on the impossible for a tool to be able to automatically create mappings from one arbitrary name to another arbitrary name, so it is necessary for the developer to say which method in the component maps to which sequence", and this can be a one-to-one or a many-to-one mapping [Marshall2001]. Nevertheless, there should be at least some kind of support for filtering inappropriate visualizations according to underlying restrictions associated with the data.

The flexibility may also be compromised due to some approach restrictions. In [Marshall200381], the collected information for creating visualizations as a complement for documentation is mostly based on the developers' experiences of using the components, and creating visualizations "does require some prior knowledge of the component and its important features and uses (i.e., knowing what to focus on in the visualization)" [Marshall200381]. This means that "any configuration is better left to experienced users who wish to create

visualizations of that component for other developers". Nevertheless, there is no support for either the developer or other users in creating, choosing or selecting visualizations.

As stated in [Silva et al. 2012], some works try to generate flexible visualizations, but they usually require expertise knowledge (e.g., programming skills for configuring/mapping views and data) for stakeholders to operate them. A list of works in this regard can be found in [Silva et al. 2012]. When an approach makes an assumption of a particular technical knowledge for creating the visualizations, it may potentially inhibit some stakeholders to use it.

Another issue that must be taken into account is that, in comparative visualizations, elements must be compared using the same visualization [Alonso & Frakes 2000] – otherwise, in addition to the data-to-visualization mapping, another mental mapping between visualizations would be required.

Enhancing awareness and understanding of software information and the software itself requires the identification of adequate abstractions according to the comprehension needs [Schots et al. 2012]. The choice of the visualization abstractions and techniques for representing the data, as well as the interaction techniques to be employed, heavily depends on contextual information, e.g., the nature of data, the visualization constraints, and the task to be supported (e.g., selecting the most suitable assets from a set of reusable assets).

Regardless of the number of occurrences for each of the strategies, it is unwise to affirm that certain techniques are more important than others. Visualization strategies and techniques must be chosen according to the visualization goals. Moreover, the available data must meet the representation constraints associated to the employed visualizations.

4.6. Medium – *where* to present the visualization?

During the analysis, it was noticed that some publications (12) do not specify information regarding the medium (i.e., the device and/or the environment) where the visualizations are displayed (*SQ5*). In such cases, the assumed information is based on the analysis of publications' contents.

Only a single approach [Charters2002765] uses a virtual reality environment (VRML-enabled browsers or standalone viewers) for displaying information (not specifying the physical medium). The other 33 approaches present visual information in a computer screen. From these, 10 contain (in their corresponding publications) information associated with the environment: 6 of them use a web environment [Alonso1998483] [Marshall2001103] [Marshall2001103 / Anslow2004] [Anslow2004 / Marshall200435] [López20091198] [Bauer2012435] and 4 employ the Eclipse IDE [Holmes2007100] [Dietrich200891] [Anquetil2010427] or an extension of it [Stollberg2007236] (the latter depends on an IDE implemented in the Eclipse framework, called Web Service Modeling Toolkit (WSMT)). The remaining 23 approaches are standalone tools (e.g., [Ye2000266]) or characterize their own environment (e.g., [Constantopoulos19951]), but only around half of them (11) makes such information explicit (that is, the aforementioned 12 approaches that do not specify information regarding the medium are among these 23).

Regarding the resources which can be used for interacting with the visualizations (*TQ5.1*), only 2 approaches ([Ye2000266] and [Holmes2007100]) explicitly state the use of both keyboard and mouse, while other 13 approaches mention mouse or mouse interactions. Among the remaining approaches, based on other details presented, it is assumed that 20 of them provide mouse support and 3 support keyboard as interaction resources, 2 of these supporting both

devices [Constantopoulos19951] [Kelleher200550]. Although [Charters2002765] runs in a virtual reality environment, no interaction resource is specified – it seems to be mouse-based, though; thus, it is included in the set of 20 approaches.

4.6.1. Discussion

Regarding the medium for displaying information (*SQ5*), it was noticed that only a few approaches (explicitly) mention that they work in (or are integrated with) a web environment. This was somehow surprising, since computers are usually equipped with web browsers, not requiring any additional installation procedure. Some recent web-based visualization frameworks may help changing this scenario.

Publications also lack more detailed information regarding the compatibility of the approaches with different media. For instance, even among more recent approaches, none mentions or focuses on mobile devices as an alternative to execute and interact with the visualizations. Moreover, in spite of the existence of web-based approaches, one cannot state (based solely on the publications) that they are multiplatform, i.e., whether they work in other devices or not, since some devices such as smartphones and tablets contain displaying and interaction constraints that must be accounted for when designing visualizations.

Regarding the resources used for interacting with the visualizations (*TQ5.1*), it is not surprising that mouse and keyboard are the main interaction resources, as current information visualization systems still largely focus on these peripherals for interacting with data [Lee et al. 2012].

In spite of that, there has been a constantly growing interest for incorporating more natural forms of interaction such as touch, speech, gestures, handwriting, and vision. However, these new forms of interaction need to “follow the basic rules of interaction design, which means well-defined modes of expression, a clear conceptual model of the way they interact with the system, their consequences, and means of navigating unintended consequences” [Norman 2010].

According to the same author, because gesturing is a natural, automatic behavior, systems also have to “be tuned to avoid false responses to movements that were not intended to be system inputs”. Thus, as an interaction technique, gestures “need time to be better developed”, so that interaction designers can “understand how best to deploy them” and, as a consequence, standard conventions can be established [Norman 2010].

4.7. Requirements – *which* resources are required by or used in the visualization?

In order to deploy and execute the systems and their visualizations (*SQ6*), 4 approaches cite the Eclipse IDE as a requirement [Holmes2007100] [Dietrich200891] [Anquetil2010427] [Duszynski2011303 / Duszynski201237], and other 2 approaches require Jun for Java [Washizaki20061222] [Ali200950]; the latter also requires OpenGL. In terms of hardware, no special requirements are mentioned in the publications – at most, one approach states that a color monitor should be preferably used [Constantopoulos19951].

The selected approaches use a variety of programming languages, APIs and frameworks (*TQ6.1*). Java is the most adopted language, explicitly mentioned by 16 approaches. For building the visualizations, the Prefuse toolkit is used by 3 approaches ([Gonçalves2007872 / Oliveira2007461] [Dietrich200891] [Anquetil2010427]). Other visualization frameworks include

the Tk graphics library (Tcl/Tk) [Biddle199992] [McGavin2006153], JPowerGraph [Stollberg2007236], Grappa [Areeprayolkij2010208] and JUNG [Dietrich200891].

4.7.1. Discussion

Software and hardware requirements (*SQ6*) are not well discussed in the publications, which hampers the proper evaluation of the feasibility of the approaches to particular contexts. The same occurs with information about programming languages, APIs and frameworks (*TQ6.1*). Such information, if properly discussed, helps to evaluate how up-to-date a tool is, as well as to identify any potential integration constraint. It can be noticed that some of the technologies used by the approaches are already in disuse.

A particular concern is the lack of integration with development environments: this hampers integration with other tools (and among the tools themselves), and may require additional efforts from stakeholders to cope with reuse tasks. IDEs can provide an enormous amount of information about the developer and his/her system, and using the IDE as the source of information is the closest way to understand the developer's intentions [Robbes & Lanza 2006].

4.8. Evidence – are the proposed visualizations *worthwhile*?

In the analysis of the evidence dimension, only evaluations informed on the selected publications were analyzed (i.e., no other sources of information were observed). Moreover, example applications made by the authors themselves without quantitative measurements (comparatively) and a minimal evaluation goal, as well as comparative evaluations made by them only mentioning differences in terms of features, were not taken into account.

Regarding the methods used for assessing the quality of the visualizations (*SQ7*), almost half of the approaches (16 out of the 34) do not describe any kind of evaluation of their use, not even performed by the authors themselves. In some cases, at most, a simple example of use is shown. Among the remaining 18 approaches, one publication presents a semi-controlled experiment [Holmes2007100] and another one presents three experiments [Feigenspan20121]. The other 16 are evaluated in terms of their use in practice, and 5 mention external evaluators/subjects ([Constantopoulos19951] [Lange1995342] [Wahid2004414] [DeBoer200951] [Yazdanshenas2012143]), sometimes applying user tests followed by a survey [Wahid2004414] or exploratory studies followed by a structured interview, guided by a questionnaire [Yazdanshenas2012143].

Only a few of the other 11 evaluations in practice specify that they were performed by the authors themselves (e.g., [Dietrich200891] and [Duszynski2011303 / Duszynski201237]), but it is assumed that they all may fit into this setting, since no external evaluators/subjects is mentioned. Evaluation methods include usability evaluations [Biddle199992 / Marshall2001 / Marshall2001103], tests executing the tool [Gonçalves2007872 / Oliveira2007461], exploratory studies [Apel2011421] and experiments that may be classified as benchmarking analyses [Ye2000266] [Tangsripairoj2006283], due to their comparative nature. Moreover, although 3 approaches mention a case study [Damaeviius2009507] [López20091198] [Areeprayolkij2010208], they do not present some expected information to be categorized this way (such as the definition of a research question, a planning of the data collection procedures and the inferences from the collected data) [Runeson & Höst 2009].

Regarding the scenarios in which the visualizations are employed (*TQ7.1*), from the 18 approaches (described in the publications) that present some kind of evaluation, 6 are performed in an academic context, 7 use open source data, and 9 present studies involving commercial projects. Some approaches are evaluated in more than one scenario. One approach does not specify the evaluation scenario. Figure 14 summarizes such information.

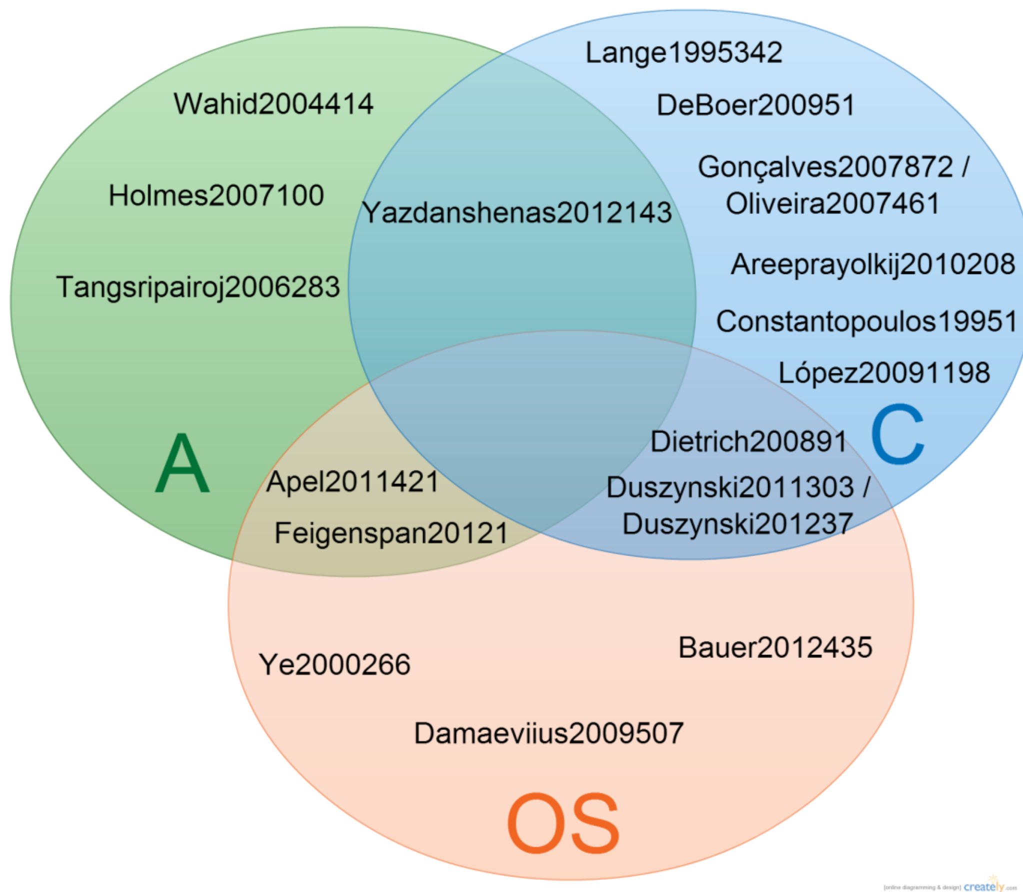


Figure 14. Evaluation scenarios (A = Academic; C = Commercial/Industrial; OS = Open Source) (*TQ7.1*)

Concerning the evaluated aspects of the approaches (*TQ7.2*), some examples include usability [Biddle199992 / Marshall2001 / Marshall2001103] [Yazdanshenas2012143], opinion of subjects [Wahid2004414] [Feigenspan20121], performance and response times [Duszynski2011303 / Duszynski201237] [Feigenspan20121], correctness [Areeprayolkij2010208] [Feigenspan20121], usefulness [Areeprayolkij2010208], effectiveness [Ye2000266] [Holmes2007100] [Yazdanshenas2012143] and scalability [Dietrich200891] [Duszynski2011303 / Duszynski201237].

Finally, regarding the results/outcomes of the conducted evaluations (*TQ7.3*), among the publications that present some evaluation of the proposed approaches, most of them show benefits on the use of visualizations. Some publications also present limitations that were found from the evaluation results (e.g., [Wahid2004414], [Damaeviius2009507], [López20091198], [Feigenspan20121] and [Yazdanshenas2012143]). Particularly, [Feigenspan20121] and

[Yazdanshenas2012143] present a detailed description of the findings, which may support and give insights for future works.

4.8.1. Discussion

In this dimension, it was observed that the majority of the works does not present a proper evaluation on their use (*SQ7*): some of them do not present any at all. This can be partially explained by the lack of demand for evidence in publications (a scenario that has been changing in the last years). In many cases, the evaluation is done by the authors themselves, which is subjective and may bring some bias. The absence of proper evaluations may raise questions as regards to meeting the purpose to which the approaches were proposed. This can be seen as a major downside.

Because quantitative evaluations that involve humans can be very time-consuming, at least qualitative evaluations should be performed during the design of visualization tools or posthoc [Diehl 2007]. For instance, two criteria that can be used for evaluating the mapping of data to visual metaphors are expressiveness and effectiveness [MacKinlay 1986] [Maletic et al. 2002]. The former refers to the capability of the metaphor to visually represent all the information to be visualized (e.g., if the number of visual parameters available in the metaphor for displaying information is equal to or greater than the number of data values to be visualized). The latter relates to the efficacy of the metaphor as a means of representing the information, and can be further distinguished in effectiveness regarding the information passing as visually perceived, regarding aesthetic concerns etc.

Moreover, in general, the reported data about the evaluations lack more useful details, so one cannot understand in which scenarios they were conducted (*TQ7.1*), which aspects were evaluated and why (*TQ7.2*), how the analysis was made and which strengths and opportunities for improvements were identified (*TQ7.3*). It must be emphasized that the experimental rigor must be correlated with the relevance of the findings, in order to avoid wrong conclusions. Some recent works present a proper experimental soundness that helps to understand the identified limitations, so that other researches aiming to support reuse can use their evaluation report as a basis.

An interesting finding is that there is a balance between the evaluation scenarios (*TQ7.1*), since not only academic projects are used, but open source projects are also taken into account (which allows verification of results), as well as commercial (thus strengthening the interaction with industry). Still, the field lacks studies on whether the perceptive and cognitive abilities of the stakeholders in carrying out software reuse tasks are properly stimulated [Schots et al. 2012]. Particularly, since industry stakeholders can directly benefit from the results of such studies, experiments in industry are recommended for strengthening interaction with academia.

5. Final Remarks

Enhancing awareness and understanding of both software information and software itself requires the identification of adequate abstractions according to the comprehension needs. In this sense, software visualizations have been increasingly supporting software engineers in performing their day-to-day activities. However, no work to date has provided a comprehensive set of research questions for providing evidence on how visualization approaches have been supporting software reuse. Moreover, the presented study also contributes with a broad and

concrete use of the task-oriented taxonomy framework proposed in [Maletic et al. 2002]. The results found in this review can be used as a starting point for future research directions that can be addressed by the software engineering community when choosing, instantiating or developing visualization-based approaches for supporting software reuse.

Besides, the presented information can also be used as a body of knowledge to support the decision making regarding the choice of visualization approaches, as well as to conduct other secondary studies on software visualization applied to another field of interest (e.g., software maintenance). This study can also be seen as a summarized catalog of the approaches (also available in a website[25] for a better exploration of the findings), whose further details can be obtained from the corresponding original publications. The extended framework not only allows organizing the findings of the study in terms of visualization dimensions, but also highlights aspects that lack support, and may indicate research opportunities on software reuse and software visualization.

Finally, the lack of data in publications for answering the research questions in terms of the software visualization dimensions may serve as a motivation to researchers for describing and categorizing their approaches.

5.1. Limitations

Some limitations of this study include:

- (i) the chosen search string, which may have not captured some relevant related work;
- (ii) the scope of analysis, which was based solely on the content of the retrieved publications (i.e., no other source was taken into account);
- (iii) the lack of variety of search engines used, which may not be representative, and
- (iv) the publication selection and the data analysis, which were made from the viewpoint of the researchers, and may be biased.

For each threat identified, some actions were performed to minimize its damage on the study results.

With respect to (i), it is known that software engineering has several terminology problems [França & Travassos 2011]; thus, for establishing the search string, an ad-hoc literature review was performed to identify the most common terminology in software visualization and software reuse (as presented in Section 2.6.1). Moreover, a recent study [Novais et al. 2013] has used a similar set of visualization-related keywords, which brings more confidence to the chosen strategy.

Regarding (ii), although it is known that publications may not have mentioned some of the visualization features in the corresponding approaches, hands-on analyses made by the researchers would make the research results too biased, since a full understanding of the tool capabilities would be required. Moreover, many of the tools are not available online. So, it was decided to accept this limitation.

[25] http://www.cos.ufrj.br/~schots/survis_reuse/

With respect to (iii), although the search was only performed in Scopus and in manual search on Brazilian events, there is evidence that Scopus has the better coverage publications on software engineering area. After analyzing the trade-off between the effort to perform the search in other bases and the Scopus effectiveness, the risk was assumed.

Finally, regarding (iv), it is worth mentioning that the main researcher (responsible for conducting the study) has practical experience in software reuse (as a developer and by implementing reuse programs in Brazilian software organizations) [Schots & Werner 2013] and academic experience in software visualization, with publications in the field (e.g., [Schots & Werner 2013] and [Schots et al. 2002], among others). This may alleviate the risk of rejecting a relevant publication. Even so, relevant data may not have been observed.

It was a very intense process of reading, analysis and data extraction (due to the large number of research questions), and some information may be mismatched due to the error-proneness nature of the manual work. During the extraction process, the studies were classified based on the judgment of the researchers, which means that some studies may have been classified incorrectly. The second researcher and the data validation procedure (mentioned in Section 2.8.8) were defined for mitigating this risk. Indeed, problems were identified and fixed due to this support. Additionally, if any problem with this study is identified afterwards, an erratum may be included in the technical report of the next review round.

5.2. Open Questions and Future Works

This study has provided evidence for answering the following question: *What are the characteristics and limitations of the visualization approaches that have been proposed to support software reuse?*

Some questions that could not be answered (or were raised) by means of this *quasi*-systematic-review are listed as follows:

- *Which aspects (comprising stakeholders' needs, reuse tasks and reuse-related data) should be taken into account for a visualization-based approach to support software reuse?*
- *Are reuse-oriented visualizations feasible in helping stakeholders to be aware of the reuse scenario and performing reuse tasks more accurately, increasing their efficiency and efficacy?*
- *Can the use of proper visualization resources assist stakeholders in carrying out their software reuse tasks, facilitating the institutionalization of a reuse program in software development organizations?*

Regarding the first question, Marshall et al. (2003) present a wish list of what they want to see in software visualizations of reusable components, as well as a wish list for the characteristics of the intermediary format that would carry collected information about the components. A drawback is that they are not based on or supported by any study or any kind of evaluation that demonstrates the adequacy, completeness and/or relevance of the wish list items. Moreover, other aspects of reuse management (present in other approaches) are not taken into account.

In this sense, future works include studies with worldwide software reuse researchers and organizations that implemented reuse processes, in order to identify and/or validate what

information should be taken into account for supporting reuse tasks. Some of the findings and perceptions obtained from the current study can be used for validation. This might also provide directions on what information should be visualized by what kinds of reuse stakeholders.

Based on the findings of this study, it was noticed that literature lacks a visualization approach that supports analyzing/monitoring the reuse scenario in an organization, providing and correlating information from different sources regarding reusable assets, users (producers and consumers of these assets) and projects in which the assets were reused. Evolution information, for instance, allows assessing an asset's stability and frequency of updates (i.e., how active the development community is). Projects' history is also useful for identifying new assets candidate to reuse, as well as for analyzing projects in which there have been reuse attempts and, from those, which were successful and why. "Social" information (e.g., who developed/reused which assets with/from which developers) has proven to be relevant as well – as shown in [Schots & Werner 2013] –, but this is also not well explored. An approach that encompasses this set of information can be provided not only for a better, effective reuse management, but also for supporting decision making on reuse activities.

We also intend to perform a new round of this review in the future (include potential term variations that were not explored in the search string of the first round), for enriching the captured information with novel approaches.

Acknowledgements

The authors would like to thank:

- the researchers Eric Wong, Robertas Damaševičius, Mariagrazia Fugini and Hironori Washizaki for gently providing by e-mail their publications that were not available on the web,
- professor Guilherme Horta Travassos, for his contributions in the research protocol of this study,
- the anonymous reviewers of the submitted papers related to the study results[26], for their suggestions on improving the description of the taxonomy framework,
- the Ph.D. students Chessman Corrêa and Natália Schots, for kindly supporting the conflict resolution procedure (during the selection process) and the data verification and aggregation (during the analysis), respectively, and
- Maria Helena Oliveira, for partially supporting the transcription and copy of data from publications in PDF.

References

Alonso, O., Frakes, W. B. (2000). "Visualization of Reusable Software Assets". In: *Proceedings of the 6th International Conference on Software Reuse (ICSR 2000)*, pp. 251-265, Vienna, Austria, June.

[26] The reviewers were provided with an early release of this study.

Barcelos, R. F., Travassos, G. H. (2006). "Evaluation Approaches for Software Architectural Documents: a Systematic Review". In: *Ibero-American Conference on Software Engineering (CIbSE 2006)*, La Plata, Argentina, pp. 433-446, April.

Basili, V., Caldiera, G., Rombach, H. (1994). "Goal Question Metric Paradigm", *Encyclopedia of Software Engineering*, v. 1, edited by John J. Marciniak, John Wiley & Sons, pp. 528-532.

Bavoil, L., Callahan, S. P., Crossno, P. J., Freire, J., Scheidegger, C. E., Silva, C. T.,. Vo, H. T. (2005). "Vistrails: Enabling interactive multiple-view visualizations". In: *IEEE Visualization* (*IEEE VIS 2005)*, Minneapolis, USA, pp. 135-142, October.

Benedicenti, L., Succi, G., Valerio, A., Vernazza, T. (1996). "Monitoring the efficiency of a reuse program". *SIGAPP Applied Computing Review*, v. 4, n. 2, pp. 8-14, September.

Biolchini, J., Mian, P. G., Natali, A. C. C., Travassos, G. H. (2005). "Systematic review in software engineering". Technical Report ES 679/05, COPPE/UFRJ.

Braga, R. M. M., Werner, C. M. L., Mattoso, M. (2006). "Odyssey-Search: A Multi-Agent System for Component Information Search and Retrieval". *Journal of Systems and Software*, v. 79, n. 2, pp. 204-215, February.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., Khalil, M. (2007). "Lessons from applying the systematic literature review process within the software engineering domain". *Journal of Systems and Software*, v. 80, n. 4, pp. 571-583, April.

Buering, T., Gerken, J., Reiterer, H. (2006). "User interaction with scatterplots on small screens – a comparative evaluation of geometric-semantic zoom and fisheye distortion". *IEEE Transactions on Visualization and Computer Graphics*, v. 12, n. 5, pp. 829-836.

Cockburn, A., McKenzie, B. (2000). "An evaluation of cone trees". In: McDonald, S., Waern, Y., Cockton, G. (eds.), *People and Computers XIV – Usability or Else! – Proceedings of HCI 2000*, pp. 425-436, Springer London.

Diehl, S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, 1st ed., Springer.

Few, S. (2009). *Now You See it: Simple Visualization Techniques for Quantitative Analysis*, 1st ed., Analytics Press.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Frakes, W., Fox, C. (1996). "Quality Improvement Using a Software Reuse Failure Modes Model". *IEEE Transactions on Software Engineering*, v. 22, n. 4, April.

França, B. B. N., Travassos, G. H. (2011). "Quasi-Systematic Review – Simulation Studies in Software Engineering". *Technical Report*. Available at http://lens-ese.cos.ufrj.br/rsl_sbs/RSL_SBS.pdf. Accessed on June 2013.

Griss, M. L., Favaro, J., Walton, P. (1994). "Managerial and organizational issues – Starting and Running a Software Reuse Program". In: Schaefer, W., Prieto-Diaz, R., Matsumoto, M. (eds.), *Software Reusability*, pp. 51-78, Ellis Horwood Ltd..

Hattori, L. (2010). "Enhancing collaboration of multi-developer projects with synchronous changes". In: *32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, Cape Town, South Africa, pp. 377-380, May.

Kim, Y., Stohr, E. A. (1998). "Software Reuse: Survey and Research Directions". *Journal of Management Information Systems,* v. 14, n. 4, pp. 113-147, March.

Kitchenham, B. (2004). "Procedures for Performing Systematic Reviews", Keele University Technical Report TR/SE-0401. Available at http://www.scm.keele.ac.uk/ease/sreview.doc.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., Linkman, S. (2009). "Systematic literature reviews in software engineering – A systematic literature review", *Information and Software Technology*, v. 51, n. 1, pp. 7-15, January.

Lanza, M., Marinescu, R. (2006). Object-Oriented Metrics in Practice. Springer-Verlag Berlin Heidelberg New York, 1st edition.

Lee, B., Isenberg, P., Riche, N. H., Carpendale, S. (2012). "Beyond Mouse and Keyboard: Expanding Design Considerations for Information Visualization Interactions". *IEEE Transactions on Visualization and Computer Graphics*, v. 18, n. 12, pp. 2689-2698, December.

Lisboa, L. B., Garcia, V. C., Lucrédio, D., Almeida, E. S., Meira, S. R. L., Fortes, R. P. M. (2010). "A Systematic Review of Domain Analysis Tools". *Information and Software Technology*, v. 52, n. 1, pp. 1-13, January.

MacKinlay, J. D. (1986). "Automating the design of graphical presentation of relational information". *ACM Transaction on Graphics*, v. 5, n. 2, pp. 110-141, April.

Maletic, J. I., Marcus, A., Collard, M. L. (2002). "A task oriented view of software visualization". In: *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis* (VISSOFT 2002), Paris, France, pp. 32-40, June.

Marshall, S. (2001). "Using and Visualizing Reusable Code: Position Paper for Software Visualization Workshop". In: *Workshop on Software Visualization, 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, Tampa, USA, October.

Marshall, S., Jackson, K., Anslow, C., Biddle, R. (2003). "Aspects to visualising reusable components". In: *Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2003)*, Adelaide, Australia, pp. 81-88, February.

Moore, J. M., Bailin, S. C. (1991). "Domain Analysis: Framework for reuse". In: Prieto-Díaz, R., Arango, G. (eds), *Domain Analysis and Software System Modeling*, pp. 179-202, IEEE Computer Society Press, Los Alamitos, USA.

Morisio, M., Ezran, M., Tully, C. (2002). "Success and failure factors in software reuse". *IEEE Transactions on Software Engineering*, v. 28, n. 4, pp. 340-357.

Mukherjea, S., Foley, J. (1996). "Requirements and Architecture of an Information Visualization Tool". In: *Database Issues for Data Visualization*, Springer Berlin/Heidelberg, pp. 57-75.

Mulholland, P. (1997). “Using a fine-grained comparative evaluation technique to understand and design software visualization tools”. In: *7th Workshop on Empirical Studies of Programmers*, Alexandria, USA, pp. 91-108, October.

Norman, D. A. (2010). “Natural User Interfaces Are Not Natural”. *Interactions*, v. 17, n. 3, pp. 6-10, May.

Novais, R. L., Torres, A., Mendes, T. S., Mendonça, M., Zazworka, N. (2013). “Software Evolution Visualization: A Systematic Mapping Study”. *Information and Software Technology*, v. 55, n. 11, pp. 1860-1883.

Pai, M., McCulloch, M., Gorman, J. D., Pai, N., Enanoria, W., Kennedy, G., Tharyan, P., Colford, J. M. (2004). “Systematic Reviews and Meta-Analyses: An Illustrated, Step-by-Step Guide”, *The National Medical Journal of India*, v. 17, n. 2, pp. 89-95.

Robbes, R., Lanza, M. (2006). “Change-Based Software Evolution”. In: *Proceedings of the 2nd International ERCIM Workshop on Software Evolution (EVOL 2006)*, Lille, France, pp. 159-164, April.

Runeson, P., Höst, M. (2009). “Guidelines for Conducting and Reporting Case Study Research in Software Engineering”. *Empirical Software. Engineering*, v. 14, n. 2, pp. 131-164, April.

Santa Isabel, S. L. (2011). “Selection of Testing Approaches for Web Applications” (in Portuguese). M.Sc. Thesis, COPPE/UFRJ, July.

Schots, M., Werner, C. (2012). ““Exploiting the Intangible: An Overview of Software Visualization and its Applications” [Explorando o Intangível: Um Panorama da Visualização de Software e suas Aplicações], Tutorial. In: *3rd Brazilian Conference on Software: Theory and Practice (CBSoft 2012)*, Natal, Brazil, September.

Schots, M., Werner, C., Mendonça, M. (2012). “Awareness and Comprehension in Software/Systems Engineering Practice and Education: Trends and Research Directions”. In: *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES 2012)*, Natal, Brazil, pp. 186-190, September.

Schots, M., Werner, C. (2013). “Characterizing the Implementation of MR-MPS-SW Reuse Processes: Preliminary Results” [Caracterizando a Implementação de Processos de Reutilização do MR-MPS-SW: Resultados Preliminares]. *Proceedings of the IX Annual Workshop of MPS (WAMPS 2013)*, Campinas, Brazil, pp. 44-53, October.

Sensalire, M., Ogao, P., Telea, A. (2009). “Evaluation of software visualization tools: Lessons learned”. In: *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*, Edmonton, Canada, pp. 19-26, September.

Silva, M., Schots, M., Werner, C. (2012). “Supporting Software Maintenance Activities through a Software Visualization Product Line Infrastructure”. In: *IX Workshop on Modern Software Maintenance (WMSWM 2012)*, Fortaleza, Brazil, pp. 1-8, June.

Tichy, W. F. (1998). “Should computer scientists experiment more?”, *IEEE Computer*, v. 31, n. 5, pp. 32-40, May.

Travassos, G. H., Santos, P. S. M., Mian, P. G., Dias Neto, A. C., Biolchini, J. (2008). “An Environment to Support Large Scale Experimentation in Software Engineering”, In:

Proceedings of the 13th International Conference on Engineering of Complex Computer Systems (ICECCS 2008), Belfast, Northern Ireland, pp. 193-202, April.

Vasconcelos, R., Schots, M., Werner, C. (2014). "An Information Visualization Feature Model for Supporting the Selection of Software Visualizations". In: *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), Early Research Achievements Track*, Hyderabad, India, pp. 261-264, June.

Appendix A – Consensus information

The first and the second researcher met personally for discussing the divergent results from the selection process and achieve the consensus. This appendix presents the following details of this stage:

- List of publications selected by both researchers (Table 20);
- List of publications selected only by the first researcher (Table 21);
- List of publications selected only by the second researcher (Table 22).
- List of publications included manually (Table 23).

Table 20. Publications accepted by both researchers

BibTeX ID	Title	Year
Ali200950	Cognitive support through visualization and focus specification for understanding large class libraries	2009
Alonso1998483	Visualization of Reusable Software Assets	1998
Anquetil2010427	A model-driven traceability framework for software product lines	2010
Areeprayolkij2010208	IDMS: A system to verify component interface completeness and compatibility for product integration	2010
Biddle199992	Reuse of debuggers for visualization of reuse	1999
Charters2002765	Visualisation for informed decision making; from code to components	2002
Damaeviius2009507	Analysis of components for generalization using multidimensional scaling	2009
DeBoer200951	Ontology-driven visualization of architectural design decisions	2009
Dietrich200891	Cluster analysis of Java dependency graphs	2008
Duszynski201237	Recovering variability information from the source code of similar software products	2012
Gonçalves2007872	DigitalAssets discoverer: Automatic identification of reusable software components	2007
Lange1995342	Interactive visualization of design patterns can help in framework understanding	1995
López20091198	Visualization and comparison of architecture rationale with semantic web technologies	2009
Marshall2001103	Visualising reusable software over the web	2001
Marshall200381	Aspects to visualising reusable components	2003
Marshall200435	Using software visualisation to enhance online component markets	2004
McGavin2006153	Visualisations of execution traces (VET): an interactive plugin-based visualisation tool	2006
Oliveira2007461	Automatic Identification of reusable Software development assets: Methodology and tool	2007
Stollberg2007236	Goal-based visualization and browsing for semantic Web services	2007

Table 21. Publications accepted only by the first researcher

BibTeX ID	Title	Year	Previous status[27]	Decision	Comments
Apel2011421	Feature cohesion in software product lines: An exploratory study	2011	Rejected (abstract)	Accepted	N/A
Bauer2012435	Understanding API usage to support informed decision making in software maintenance	2012	Rejected (title)	Accepted	N/A
Constantopoulos 19951	The software information base: A server for reuse	1995	Rejected (abstract)	Accepted	N/A
DiFelice199128 7	An interaction environment supporting the retrievability of reusable software components	1991	Rejected (full)	Rejected	Decision (by consensus): The visual part of the developed tool is a Graphical User Interface (GUI), thus it does not present a visualization approach for supporting software reuse.
Duszynski20113 03	Analyzing the source code of multiple software variants for reuse potential	2011	Rejected (abstract)	Accepted	N/A
Feigenspan2012 1	Do background colors improve program comprehension in the #ifdef hell?	2012	Rejected (title)	Accepted	N/A
Holmes2007100	Task-specific source code dependency investigation	2007	Rejected (full)	Accepted	Decision (by consensus): After the new read, it was agreed that the visualization approach supports software reuse tasks in usual software development paradigms.
Mancoridis1993 74	Conceptual framework for software development	2007	Rejected (title)	Accepted	N/A
Mittermeir20019 5	Goal-driven combination of software comprehension approaches for component based development	2001	Rejected (abstract)	Accepted	Decision (by consensus): After re-analyzing the publication (a full read), it was observed that it is indeed related to software reuse.

[27] This column represents the classification given by the second researcher before the consensus.

BibTeX ID	Title	Year	Previous status[27]	Decision	Comments
Tangsripairoj2006283	Organizing and visualizing software repositories using the growing hierarchical self-organizing map	2006	Rejected (full)	Accepted	Decision (by consensus): Although the authors do not present a visualization tool, they provide information (i.e., give enough details) on how the visualization of the growing hierarchical self-organized map can be visually presented.
Wahid2004414	Visualization of design knowledge component relationships to facilitate reuse	2006	Rejected (abstract)	Accepted	N/A
Washizaki20061222	A system for visualizing binary component-based program structure with component functional size	2006	Rejected (abstract)	Accepted	N/A
Washizaki2007284	A framework for measuring and evaluating program source code quality	2007	Rejected (abstract)	Rejected	Decision (by consensus): Although the authors point out a "visualization tool", such tool is a HTML table that presents the aggregated results from the measurement process. Thus, it was decided to reject the publication, since it does not present a visualization approach for supporting software reuse.
Yazdanshenas201212143	Tracking and visualizing information flow in component-based systems	2012	Rejected (full)	Accepted	Decision (by consensus): Although not clearly stated, the visualization supports software reuse.
Ye2000266	A visualised software library: Nested self-organising maps for retrieving and browsing reusable software assets	2000	Rejected (abstract)	Accepted	Decision (by consensus): After re-analyzing the publication (a full read), it was observed that it indeed supports software reuse by visualization.

Table 22. Publications accepted only by the second researcher

BibTeX ID	Title	Year	Previous status[28]	Decision	Comments
Kelleher200550	A reusable traceability framework using patterns	2005	Rejected (abstract)	Accepted	N/A
Kang1998175	Using design abstractions to visualize, quantify, and restructure software	1998	Rejected (full)	Rejected	Decision (by consensus): It was noticed that the publication focuses on reengineering software systems, but its impact on software reuse is not clearly stated.
Helfman199631	Dotplot patterns: a literal look at pattern languages	1996	Rejected (title)	Accepted	N/A
Kazman199694	Tool support for architecture analysis and design	1996	Rejected (full)	Rejected	Decision (**after analysis of a third researcher**): The visualization support presented in the publication does not support software reuse.
Sefika1996389	Architecture-oriented visualization	1996	Rejected (abstract)	Rejected	Decision (by consensus): Reuse is mentioned in the introduction (high-level architectures support reuse), but the publication does not present a visualization for supporting reuse.

Table 23. Publications manually included (agreed by both researchers)

BibTeX ID	Title	Year
Anslow2004	Software visualization tools for component reuse	2004
Marshall2001	Using and Visualizing Reusable Code[29]	2001

[28] This column represents the classification given by the first researcher before the consensus.

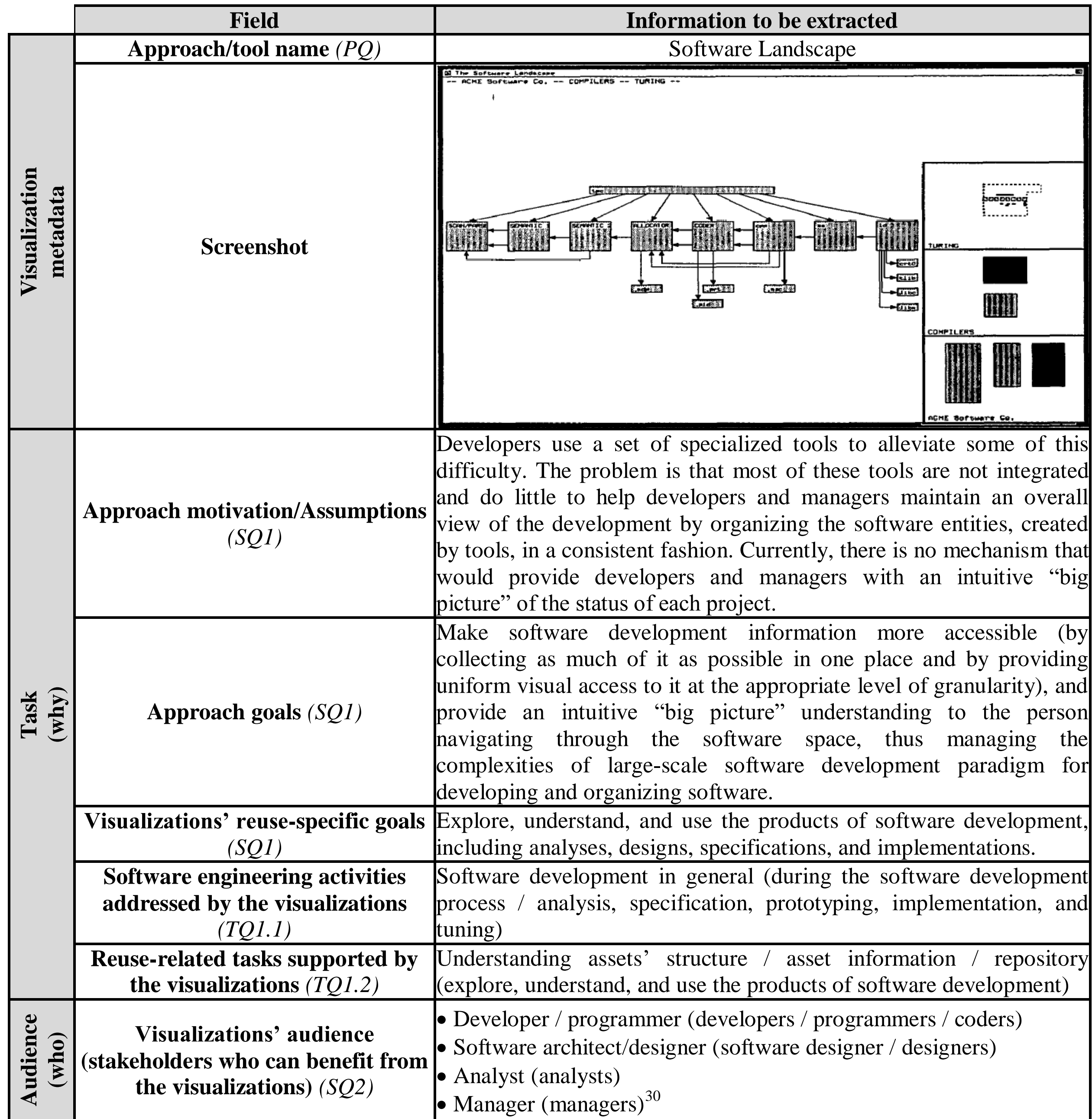
[29] Already included as control publication.

Appendix B – Publication data extracted on the selection process

The following tables (from Table 24 to Table 57) present the data extracted on the selection process for each approach listed in Table 10.

Table 24. Software Landscape [Mancoridis199374]

	Field	Information to be extracted
Publication metadata	Title	Conceptual framework for software development
	Authors	Mancoridis, S., Holt, R. C., Penny, D. A.
	Publication date (year/month)	February, 1993
	Publication type	Conference
	Source	Proceedings of the 1993 ACM Computer Science Conference
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Indianapolis, USA
	Pages	pp. 74-80
	Link (if applicable)	http://dx.doi.org/10.1145/170791.170806
	Abstract	Large scale software development is an intrinsically difficult task. Developers use a set of specialized tools to alleviate some of this difficulty. The problem is that most of these tools are not integrated and do little to help developers and managers maintain an overall view of the development by organizing the software entities, created by tools, in a consistent fashion. Our solution, called the Software Landscape, provides developers with a conceptual framework of integrated tools while providing a metaphor for managing the complexities of large-scale software development. The Software Landscape is a metaphor of a country-side viewed from above in which each major entity, such as a software project, appears as a large plot of land, and each minor entity, such as a source C module, is contained within a plot. Plots can be libraries of reusable software as well as ongoing developments. A Software Landscape can be used as a mechanism that allows the developer to navigate around the entities created during the software development process, much the way a flight simulator allows one to 'fly' and optionally to dive down to entities of interest. During this flight, the SDE 'pilot' chooses appropriate views of entities and controls their level of visible detail. This model is constructive, allowing the developer to manipulate, as well as view, the entities of the Landscape.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Software Landscape
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Developers use a set of specialized tools to alleviate some of this difficulty. The problem is that most of these tools are not integrated and do little to help developers and managers maintain an overall view of the development by organizing the software entities, created by tools, in a consistent fashion. Currently, there is no mechanism that would provide developers and managers with an intuitive “big picture” of the status of each project.
	Approach goals *(SQ1)*	Make software development information more accessible (by collecting as much of it as possible in one place and by providing uniform visual access to it at the appropriate level of granularity), and provide an intuitive “big picture” understanding to the person navigating through the software space, thus managing the complexities of large-scale software development paradigm for developing and organizing software.
	Visualizations’ reuse-specific goals *(SQ1)*	Explore, understand, and use the products of software development, including analyses, designs, specifications, and implementations.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development in general (during the software development process / analysis, specification, prototyping, implementation, and tuning)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets’ structure / asset information / repository (explore, understand, and use the products of software development)
Audience (who)	**Visualizations’ audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer / programmer (developers / programmers / coders) • Software architect/designer (software designer / designers) • Analyst (analysts) • Manager (managers)[30]

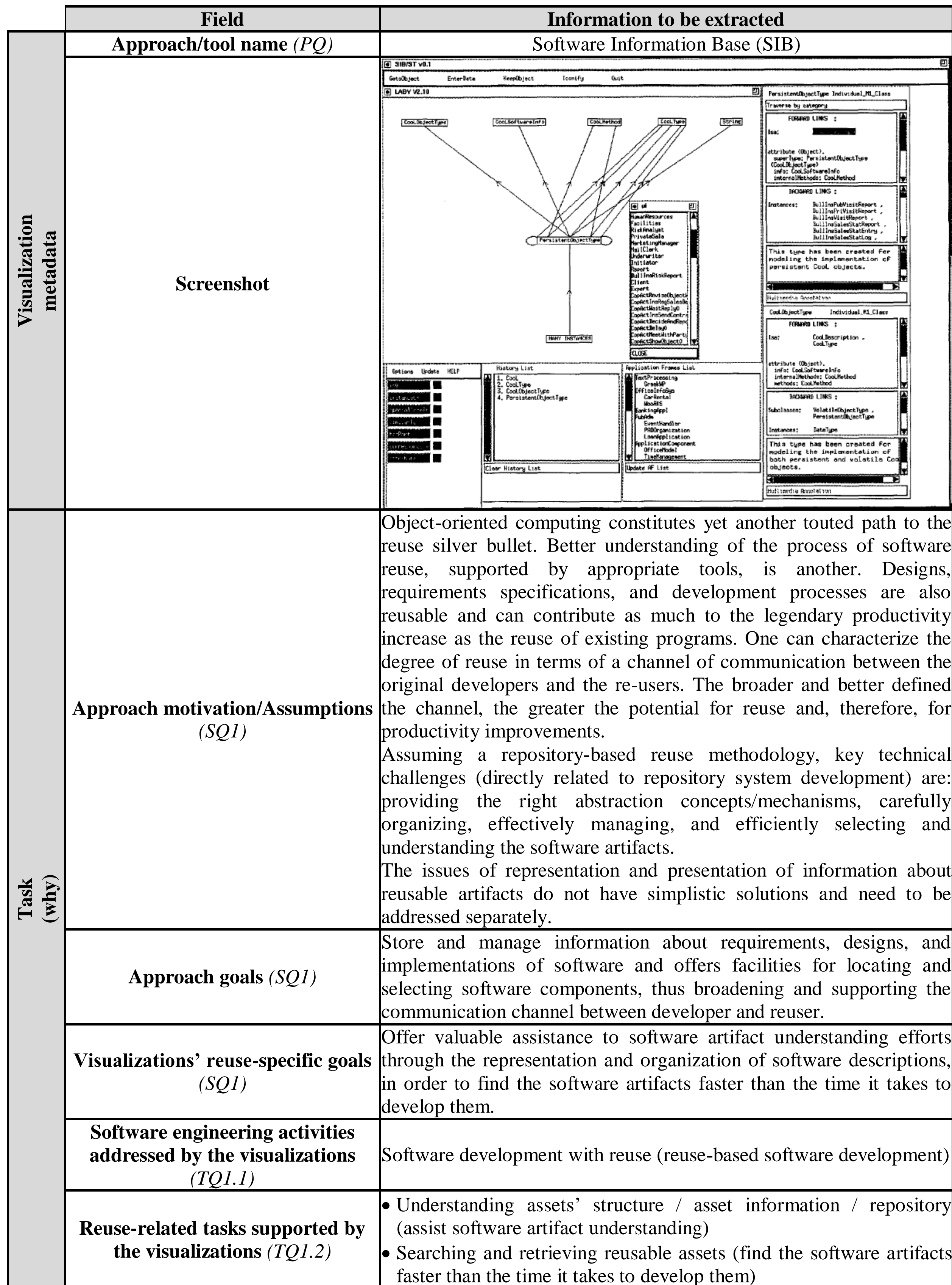
[30] The authors state that the visualization is targeted to developers, but points out actions that could eventually be performed by managers [Mancoridis199374].

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (libraries of reusable software) • Source code and related information (source module, programming language constructs such as modules and classes) • Software project and related information (software project / sub-systems, projects, and libraries)
	Source of visualized items/data *(TQ3.1)*	The entities and relations of the Software Landscape are stored in the SDE repository.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The automatic checking of syntax rules against a design is done by generating a Prolog database. The database is checked for correctness by a Prolog program that encodes definitions and rules.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Network / Graph (box-and-arrow diagrams) • Real world metaphor (country-side viewed from above)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each major entity, such as a software project, appears as a large plot of land, and each minor entity, such as a source C module, is contained within a plot. Plots can be libraries of reusable software as well as ongoing developments. Each of these projects appears as a box (a plot). Shaded boxes represent software entities that contain subsystems. A Landscape cluttered with entities indicates insufficient structuring, and one cluttered with arrows indicates excessive coupling. Entities are represented as boxes. The import relation is shown by arrows, and the contain relation by drawing one entity inside the other. The inherit relation is shown by arrows labeled inherit. The export relation is shown by having the exported entity protrude from its exporting entity.
	Visualization strategies and techniques *(TQ4.2)*	• Browsing / Navigation (navigate around the entities / navigate into a particular entity) • Details on demand / Drill-down (navigate around the entities created during the software development process, much the way a flight simulator allows one to "fly" and optionally to dive down to entities of interest) • Details on demand / Labeling (arrows labeled inherit) • Filtering / Highlighting/Mitigation (controls their level of visible detail) • Filtering / Inclusion/Removal (hide entities and relations that are of no relevance to their current task / allows users to expose or hide the sub-structure of each entity) • Overview + detail (navigate around the entities created during the software development process, much the way a flight simulator allows one to "fly" and optionally to dive down to entities of interest)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (on a large display screen / a language-centered programming environment)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: A prolog database is necessary. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	It is based in a Prolog program that encodes the domain definitions and rules.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 25. Software Information Base (SIB) [Constantopoulos19951]

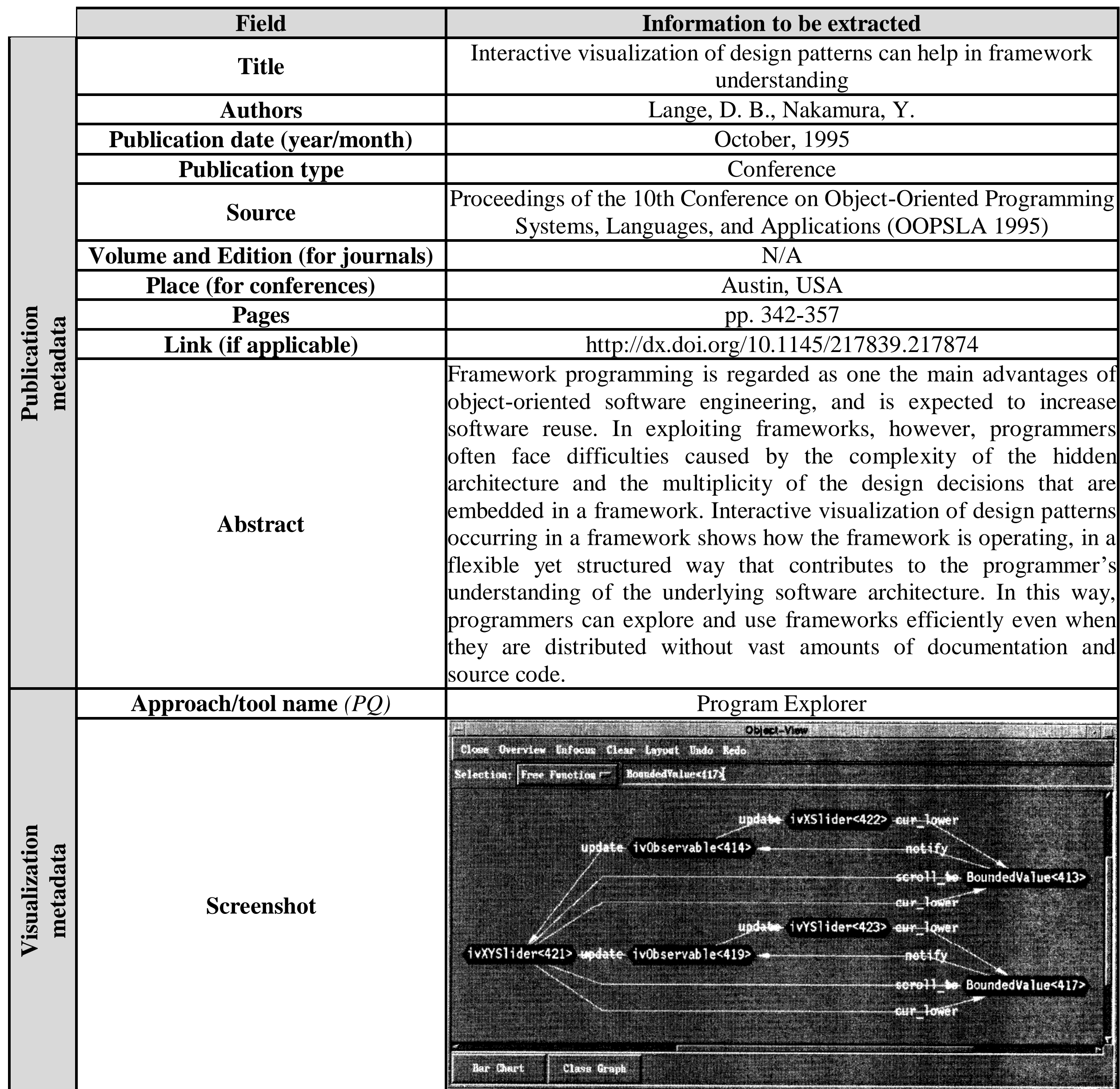
	Field	Information to be extracted
Publication metadata	**Title**	The software information base: A server for reuse
	Authors	Constantopoulos, P., Jarke, M., Mylopoulos, J., Vassiliou, Y.
	Publication date (year/month)	??, 1995
	Publication type	Article (Journal)
	Source	The VLDB Journal
	Volume and Edition (for journals)	v. 4, n. 1
	Place (for conferences)	N/A
	Pages	pp. 1-43
	Link (if applicable)	http://dx.doi.org/10.1007/BF01232471
	Abstract	We present an experimental software repository system that provides organization, storage, management, and access facilities for reusable software components. The system, intended as part of an applications development environment, supports the representation of information about requirements, designs and implementations of software, and offers facilities for visual presentation of the soft-ware objects. This article details the features and architecture of the repository system, the technical challenges and the choices made for the system development along with a usage scenario that illustrates its functionality. The system has been developed and evaluated within the context of the ITHACA project, a technology integration/software engineering project sponsored by the European Communities through the ESPRIT program, aimed at developing an integrated reuse-centered application development and support environment based on object-oriented techniques.



	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Software Information Base (SIB)
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Object-oriented computing constitutes yet another touted path to the reuse silver bullet. Better understanding of the process of software reuse, supported by appropriate tools, is another. Designs, requirements specifications, and development processes are also reusable and can contribute as much to the legendary productivity increase as the reuse of existing programs. One can characterize the degree of reuse in terms of a channel of communication between the original developers and the re-users. The broader and better defined the channel, the greater the potential for reuse and, therefore, for productivity improvements. Assuming a repository-based reuse methodology, key technical challenges (directly related to repository system development) are: providing the right abstraction concepts/mechanisms, carefully organizing, effectively managing, and efficiently selecting and understanding the software artifacts. The issues of representation and presentation of information about reusable artifacts do not have simplistic solutions and need to be addressed separately.
	Approach goals *(SQ1)*	Store and manage information about requirements, designs, and implementations of software and offers facilities for locating and selecting software components, thus broadening and supporting the communication channel between developer and reuser.
	Visualizations' reuse-specific goals *(SQ1)*	Offer valuable assistance to software artifact understanding efforts through the representation and organization of software descriptions, in order to find the software artifacts faster than the time it takes to develop them.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (reuse-based software development)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (assist software artifact understanding) • Searching and retrieving reusable assets (find the software artifacts faster than the time it takes to develop them)

	Field	Information to be extracted
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer / programmer (developer / application developers / application engineers) • Developer with reuse (reuser)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (software artifacts (objects) / semantic relationships)
	Source of visualized items/data *(TQ3.1)*	The Software Information Base itself and the descriptions that provide information about a software system.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Descriptions serve as basic building blocks, and provide information about a software system, which may concern a requirements, design, or implementation specification for a particular software system. Such information may also be used to represent design decisions or run-time performance information about a software object. Developers change the schema by populating the SIB system with software descriptions (either manually or through development tools).
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (directed attributed graph)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Nodes describe software artifacts (objects) and edges represent semantic relationships that hold among them. The types of links are represented by a color code.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (selection strategy) • Browsing / Navigation (navigational facilities / an implicit query is generated through navigational commands in the browsing mode / querying facilities (e.g., browsing, filtering, navigating)) • Browsing / Querying (explicit query involves an arbitrary predicate explicitly formulated in a query language or through an appropriate form interface / approximate retrieval based on similarities among software artifacts) • Details on demand / Labeling (make it possible to associate with any software object annotations and/or animations / edges may have their own labels) • Filtering (filtering)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (ITHACA application development environment / window / color monitor)
	Resources used for interacting with the visualizations *(TQ5.1)*	• Mouse • Keyboard (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: The system runs on Sun3, Sun4 series, SparcStations and 386 machines under UNIX. The X window system is required. • HW: Preferably a color monitor.
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The implementation is based on C++. The representation language chosen for the SIB is Telos, a conceptual modeling language in the family of entity-relationship (ER) models. It also uses the Graphical Browser, built using the LABY graphical editor.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [by others]
	Application scenarios of the visualizations *(TQ7.1)*	In the context of a specific reuse-oriented methodology developed in the ITHACA project. [commercial]
	Evaluated aspects *(TQ7.2)*	The SIB model and system
	Visualization evaluation results/outcomes *(TQ 7.3)*	Application engineers, who also use Telos directly, have found the E-R nature of the language and the graphical visualization very effective.

Table 26. Program Explorer [Lange1995342]

	Field	Information to be extracted
Publication metadata	**Title**	Interactive visualization of design patterns can help in framework understanding
	Authors	Lange, D. B., Nakamura, Y.
	Publication date (year/month)	October, 1995
	Publication type	Conference
	Source	Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Austin, USA
	Pages	pp. 342-357
	Link (if applicable)	http://dx.doi.org/10.1145/217839.217874
	Abstract	Framework programming is regarded as one the main advantages of object-oriented software engineering, and is expected to increase software reuse. In exploiting frameworks, however, programmers often face difficulties caused by the complexity of the hidden architecture and the multiplicity of the design decisions that are embedded in a framework. Interactive visualization of design patterns occurring in a framework shows how the framework is operating, in a flexible yet structured way that contributes to the programmer's understanding of the underlying software architecture. In this way, programmers can explore and use frameworks efficiently even when they are distributed without vast amounts of documentation and source code.
Visualization metadata	**Approach/tool name** *(PQ)*	Program Explorer
	Screenshot	

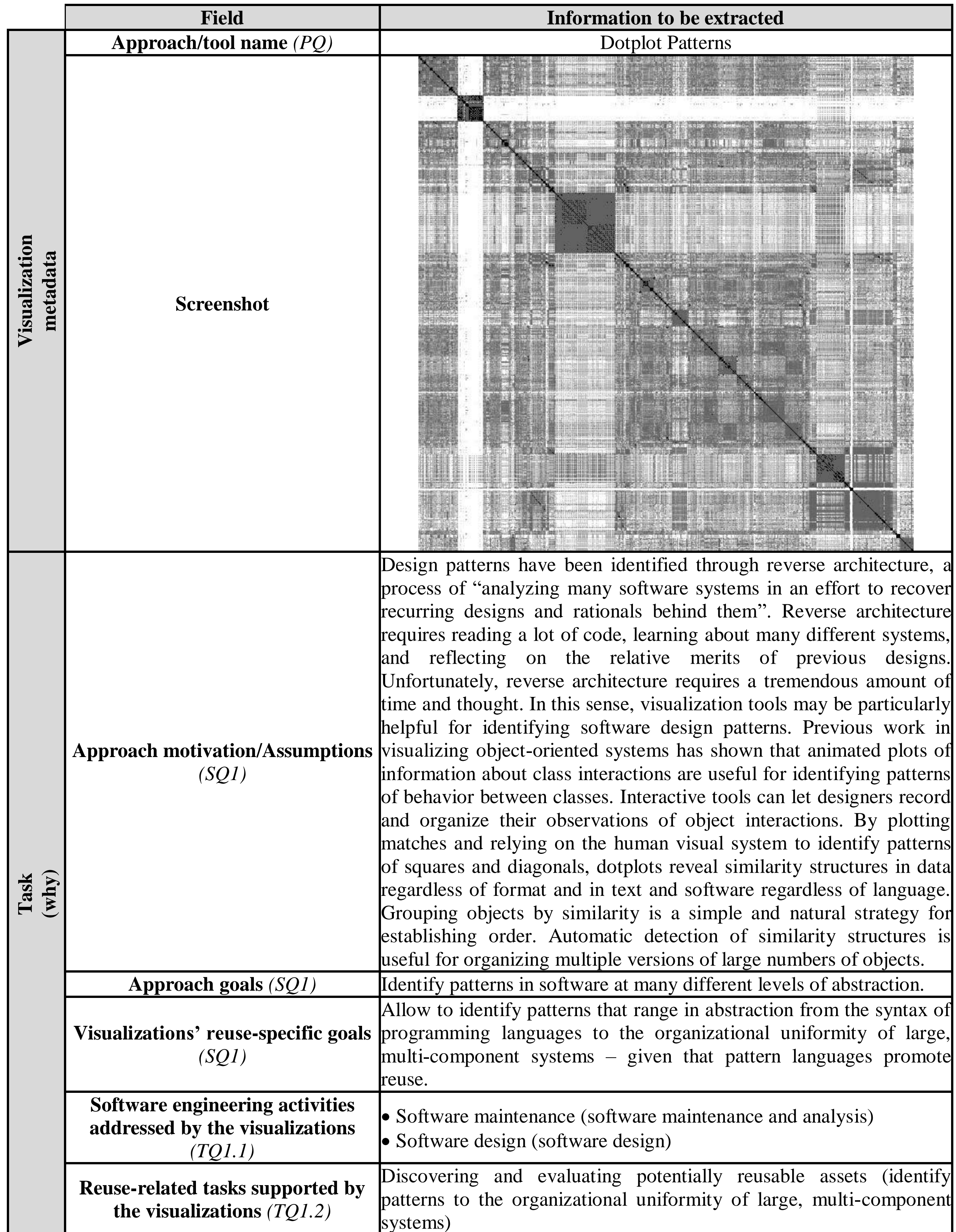
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch [Booch]. Although programmers can normally use a framework without completely understanding how it works, and can even extend it, a framework is most useful to someone who understands it in detail. Running O-O systems involves generating huge amounts of static and dynamic program information that is hard to digest, especially if the information is presented in a purely textual format. Any unguided attempt to understanding will most likely lead to cognitive overload and result in an inability to localize relevant parts of the framework or determine which paths to explore. What aids understanding is the coupling of the abstract and the concrete, that is, of static and execution information. Static information can leverage execution information and vice versa.
	Approach goals *(SQ1)*	Make the process of O-O understanding more empirical and realistic by connecting program execution to the understanding of objects and interactions, and the static program information source code to the understanding of classes and their relationships, aiming to demonstrate how patterns can serve as guides in program exploration and thus make the process of understanding more efficient.
	Visualizations' reuse-specific goals *(SQ1)*	Provide class- and object-centered views of the structure and behavior of large C++ systems with information accurate enough to enable programmers to reuse and maintain undocumented parts of these systems.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software design (design) • Programming / Coding (programming)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (provide class- and object-centered views of the structure of systems) • Understanding assets' behavior (provide class- and object-centered views of the behavior of systems) • Integrating reusable assets (provide information to enable reuse and maintenance of undocumented parts of systems)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (programmers)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (objects, design patterns / static class relationship, interactions, class inheritance, function calls, variable access, object creation, object call)
	Source of visualized items/data *(TQ3.1)*	A program database for C++.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Coupling semantics are described in terms of logic rules. Static program information is retrieved by the Program Database from so-called pdb-files generated by IBM's xlC compiler. The following entities are instrumented: object creation and deletion; implicit and explicit constructors, destructors, copy-constructors, and assignment operators; member functions and variables; and template classes and functions. Program events generated by the inserted code are captured by the Trace Recorder process events to produce a trace, which it also stores. Program execution is controlled and trace information is queried by Program Explorer through the Trace Recorder's external interface.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graph)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Objects are displayed as nodes in the graph, and interactions between objects, such as creation, invocation and variable access, are displayed as arrows. Each bar represents the longevity of a specific object. Arrows represent interactions as in the Object Graph, and are displayed from top to bottom in the order in which the interactions actually took place.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (breakpoint facility, which allows the user to control program execution / it is possible to set breakpoints at virtually and visible entities: classes, call-relationships, objects, invocations, and so on) • Browsing / Navigation (interactive hypertext like navigation / users initially sees only the class in focus and its immediate base class and derived classes; they can then interactively explore the inheritance relationships of each of these classes) • Filtering / Collapse/Expand (expanded) • Presentation / Simultaneous (Program Explorer's GUI consists of four panes)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Program Database, a stand-alone application that implements the schema of static program information. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The implementation is based on a predicate logic and uses Prolog notation.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [by others] (being used [in the time of publication])
	Application scenarios of the visualizations *(TQ7.1)*	Within IBM, as well as in a commercial project outside IBM. [commercial]
	Evaluated aspects *(TQ7.2)*	Not specified
	Visualization evaluation results/outcomes *(TQ 7.3)*	Users were satisfied after using the tool for various purposes, most notably: (1) for support in the process of understanding specific in-house C++ frameworks; (2) as a design review tool, where designers can visualize the actual design as opposed to the planned design; and (3) for visual debugging of application logic in C++ based systems.

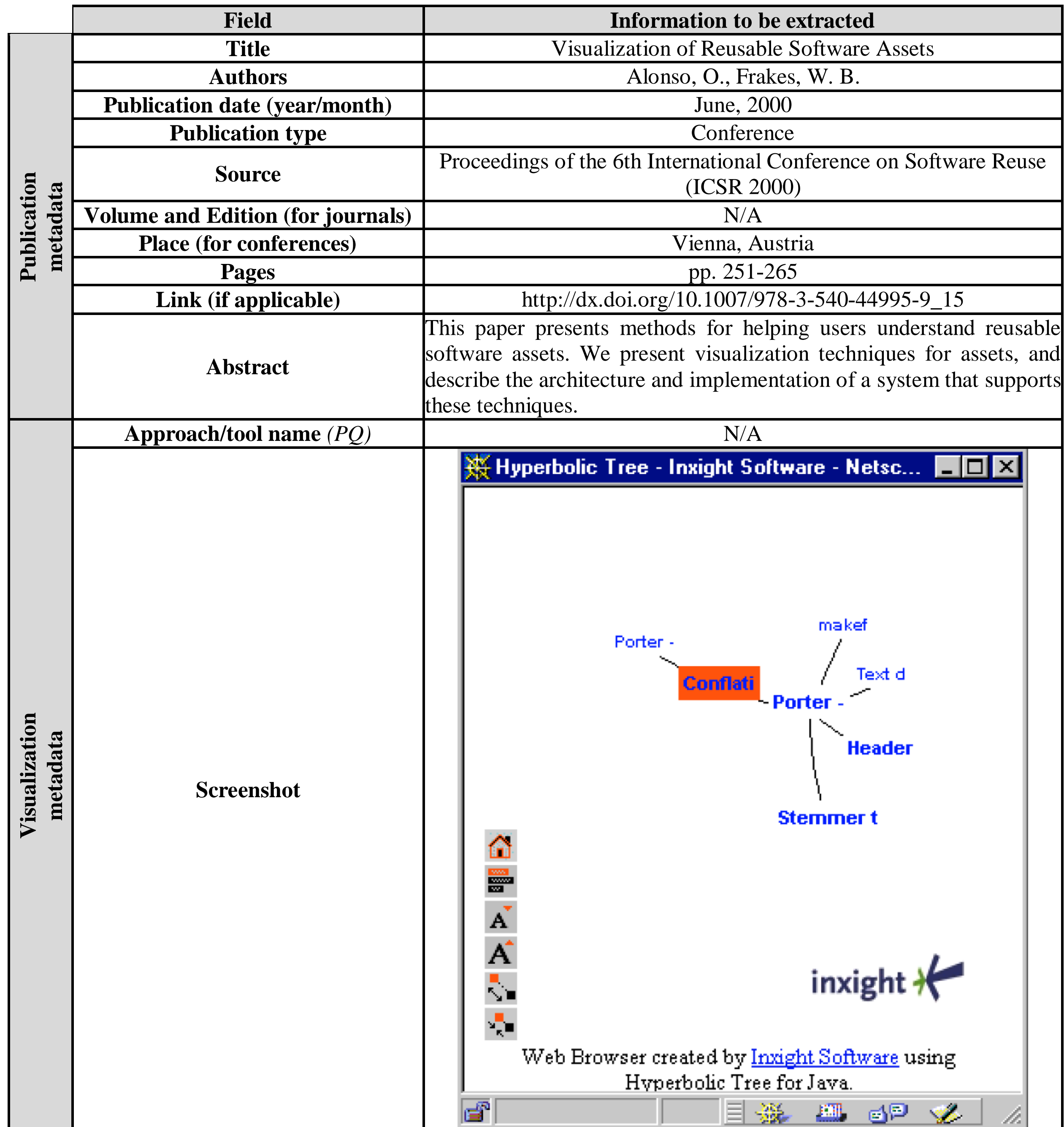
Table 27. Dotplot Patterns [Helfman199631]

	Field	Information to be extracted
Publication metadata	**Title**	Dotplot patterns: a literal look at pattern languages
	Authors	Helfman, J.
	Publication date (year/month)	??, 1996
	Publication type	Article (Journal)
	Source	Theory and Practice of Object Systems
	Volume and Edition (for journals)	v. 2, n. 1
	Place (for conferences)	N/A
	Pages	pp. 31-41
	Link (if applicable)	http://dx.doi.org/10.1002/(SICI)1096-9942(1996)2:1<31::AID-TAPO3>3.0.CO;2-A
	Abstract	This article describes the dotplot data visualization technique and its potential for contributing to the identification of design patterns. Pattern languages have been used in architectural design and urban planning to codify related rules-of-thumb for constructing vernacular buildings and towns. When applied to software design, pattern languages promote reuse while allowing novice designers to learn from the insights of experts. Dotplots have been used in biology to study similarity in genetic sequences. When applied to software, dotplots identify patterns that range in abstraction from the syntax of programming languages to the organizational uniformity of large, multicomponent systems. Dotplots are useful for design by successive abstraction – replacing duplicated code with macros, subroutines, or classes. Dotplots reveal a pervasive design pattern for simplifying algorithms by increasing the complexity of initializations. Dotplots also reveal patterns of wordiness in languages – one example inspired a design pattern for a new programming language. In addition, dotplots of data associated with programs identify dynamic usage patterns – one example identifies a design pattern used in the construction of a UNIX(tm) file system.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Dotplot Patterns
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Design patterns have been identified through reverse architecture, a process of "analyzing many software systems in an effort to recover recurring designs and rationals behind them". Reverse architecture requires reading a lot of code, learning about many different systems, and reflecting on the relative merits of previous designs. Unfortunately, reverse architecture requires a tremendous amount of time and thought. In this sense, visualization tools may be particularly helpful for identifying software design patterns. Previous work in visualizing object-oriented systems has shown that animated plots of information about class interactions are useful for identifying patterns of behavior between classes. Interactive tools can let designers record and organize their observations of object interactions. By plotting matches and relying on the human visual system to identify patterns of squares and diagonals, dotplots reveal similarity structures in data regardless of format and in text and software regardless of language. Grouping objects by similarity is a simple and natural strategy for establishing order. Automatic detection of similarity structures is useful for organizing multiple versions of large numbers of objects.
	Approach goals *(SQ1)*	Identify patterns in software at many different levels of abstraction.
	Visualizations' reuse-specific goals *(SQ1)*	Allow to identify patterns that range in abstraction from the syntax of programming languages to the organizational uniformity of large, multi-component systems – given that pattern languages promote reuse.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software maintenance (software maintenance and analysis) • Software design (software design)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Discovering and evaluating potentially reusable assets (identify patterns to the organizational uniformity of large, multi-component systems)

	Field	Information to be extracted
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Software architect/designer (novice designers / designers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (code, lines of code)
	Source of visualized items/data *(TQ3.1)*	C code and a database that includes information about when and why the code was created or changed.
	Collection procedure/method of visualized items/data *(TQ3.2)*	A sequence is tokenized and plotted from left to right and top to bottom with a dot where the tokens match.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Matrix / Matrix-like (dot, grid boxes)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	A sequence is tokenized and plotted from left to right and top to bottom with a dot where the tokens match. Dots off the main diagonal indicate similarities. Darker areas indicate regions with a lot of matches (a high degree of similarity); dark areas off the main diagonal indicate a degree of similarity between submodules; the darker the area, the higher the degree of similarity. Lighter areas indicate regions with few matches (a low degree of similarity). Dark areas along the main diagonal indicate submodules. Software sequences are tokenized into lines of code so that a dot appears where two entire lines of code match.
	Visualization strategies and techniques *(TQ4.2)*	• Clustering (concatenation to combine sequences / shuffling to combine sequences / insertion (a trivial type of shuffling) / reordering (a crude type of shuffling)) • Overview + detail (visual overview of the structure of enormous systems) • Sorting (concatenation to combine sequences / shuffling to combine sequences / insertion (a trivial type of shuffling) / reordering (a crude type of shuffling))
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 28. Alonso & Frakes's approach [Alonso1998483]



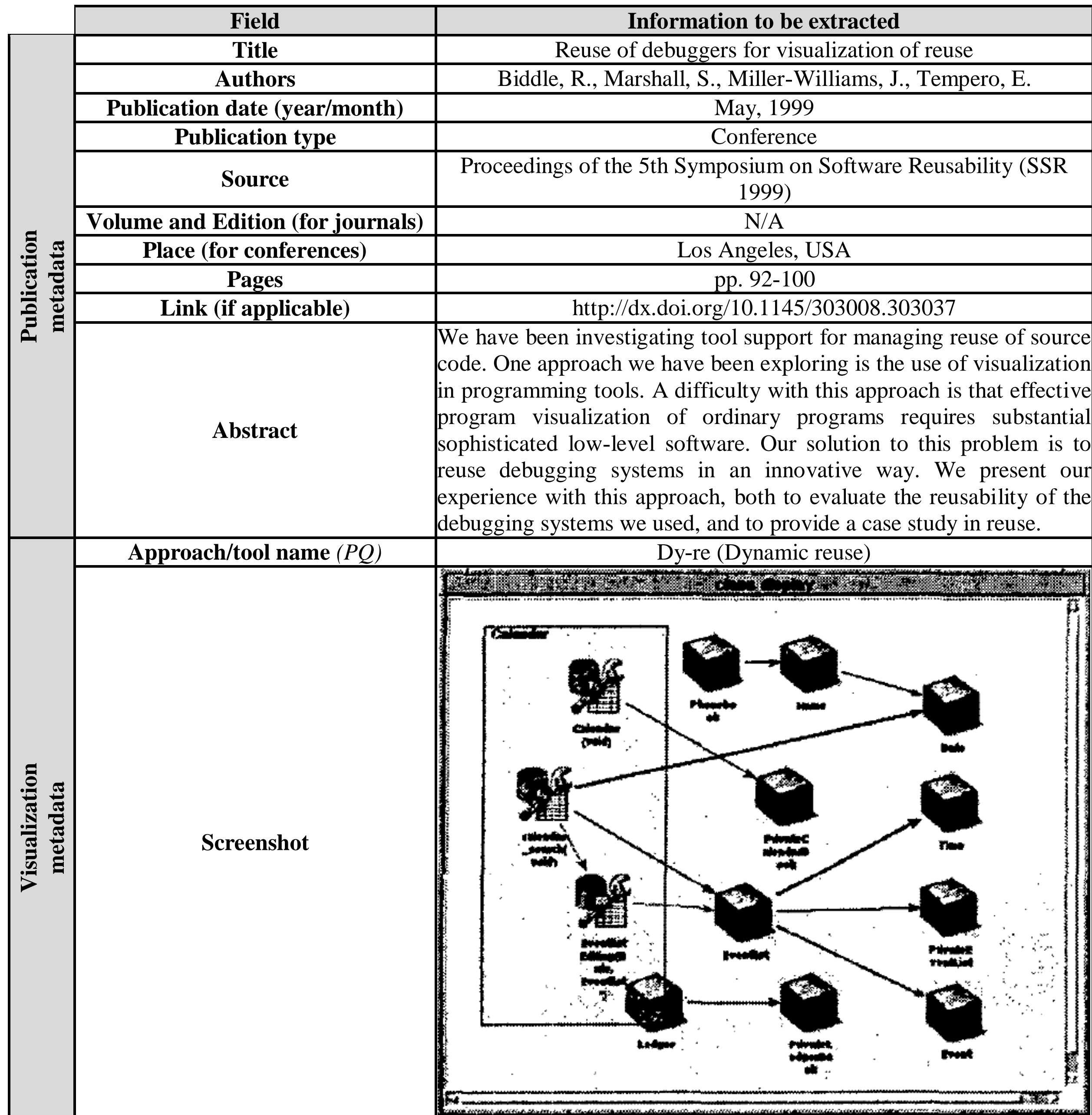
	Field	Information to be extracted
Publication metadata	**Title**	Visualization of Reusable Software Assets
	Authors	Alonso, O., Frakes, W. B.
	Publication date (year/month)	June, 2000
	Publication type	Conference
	Source	Proceedings of the 6th International Conference on Software Reuse (ICSR 2000)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Vienna, Austria
	Pages	pp. 251-265
	Link (if applicable)	http://dx.doi.org/10.1007/978-3-540-44995-9_15
	Abstract	This paper presents methods for helping users understand reusable software assets. We present visualization techniques for assets, and describe the architecture and implementation of a system that supports these techniques.
Visualization metadata	**Approach/tool name** *(PQ)*	N/A
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	If software engineers cannot understand components, they will not be able to reuse them. However, current methods for representing reusable components are inadequate. A study of four common representation methods for reusable software components showed that none of the methods worked very well for helping users understand the components. For visualizing certain attributes some representations are more suitable than others. Statistics about reuse, numbers of hits and percentage of code reused can be very helpful when the user is analyzing the history of reuse of the component. Also, comparison is key for decision making.
	Approach goals *(SQ1)*	Help the user to understand and compare reusable components.
	Visualizations' reuse-specific goals *(SQ1)*	Help users understand and integrate components into applications.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (understand assets so they can reuse them)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (help understand components)[31] • Integrating reusable assets (integrate components into applications)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	User (users)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (reusable software assets, reusable software components) • Source code and related information (source code, concept, content)
	Source of visualized items/data *(TQ3.1)*	The repository stores and manages the assets and its metadata.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The approach emphasizes the understanding process assuming a known search method. Using a search system, the user queries the repository and get, if found, a list of assets. The repository stores, manages, and provides a search mechanism for the assets and its metadata. An intermediate representation allows for data interchange between the repository and a visualization metaphor. If the visualization metaphor supports the format, it will render it accordingly. Otherwise, a transformer will map the intermediate representation to the visualization metaphors format. The intermediate representation consists of 3CML documents that represent assets and its metadata in terms of the 3Cs. 3CML (3Cs Markup Language) describes assets information in terms of the 3Cs model.

[31] The authors mention a model and system for storing, retrieving, and visualizing components in a software repository, but the visualization is not used in this stage. The authors assume the existence of a system that can store, retrieve, and manage different asset types [Alonso1998483].

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Hierarchy (trees, hyperbolic tree)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The files in 3CML are mapped to a visual structure, which augment a spatial substrate with marks and graphical properties. It is important that all the data in 3CML is mapped to a visual structure. The concept is the root. Each content represents a link from the root.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (double click and see the source code in the Web browser) • Browsing / Navigation (navigation) • Details on demand / Drill-down (double click and see the source code in the Web browser) • Filtering / Highlighting/Mitigation (highlighted) • Panning / Drag-and-drop (point to the node and drag it) • Overlap / Flipping (in comparative visualization the components are compared using the same visualization (e.g. a table)) • Focus + context (focus + context technique / root at the center, but the display can be transformed to bring other nodes into focus)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (the screen / web browser)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: It is assumed the existence of a system that can store, retrieve, and manage different asset types. A web browser is also necessary. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented as Java applets, it uses the 3C model (concept, content, and context) through the 3CML (3Cs Markup Language).
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 29. Dy-re (Dynamic reuse) [Biddle199992]

	Field	Information to be extracted
Publication metadata	Title	Reuse of debuggers for visualization of reuse
	Authors	Biddle, R., Marshall, S., Miller-Williams, J., Tempero, E.
	Publication date (year/month)	May, 1999
	Publication type	Conference
	Source	Proceedings of the 5th Symposium on Software Reusability (SSR 1999)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Los Angeles, USA
	Pages	pp. 92-100
	Link (if applicable)	http://dx.doi.org/10.1145/303008.303037
	Abstract	We have been investigating tool support for managing reuse of source code. One approach we have been exploring is the use of visualization in programming tools. A difficulty with this approach is that effective program visualization of ordinary programs requires substantial sophisticated low-level software. Our solution to this problem is to reuse debugging systems in an innovative way. We present our experience with this approach, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse.
Visualization metadata	Approach/tool name *(PQ)*	Dy-re (Dynamic reuse)
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Making reuse technically easier is still important. The key requirements are controllability and observability of running programs; be target independent, and work with ordinary programs and components, not just specially created or annotated code; be unobtrusive, and not change the code itself to assist visualization. Actions as low-level as the program code should be provided, because it cannot be predicted what aspects of the code behavior the programmer may be interested in. For portability and ease of use reasons, actions from the compile phase should not be generated. This leaves the executable phase as the best place to generate actions. A debugger provides much of the functionality required, and any mature programming environment has some form of debugging system. Most debuggers require a compiler to embed information within the generated code to assist debugging. A pure representation of the PMV model might separate the components completely and develop a communication protocol between them. However, it proved easier for the mapping component and the visualization component to be parts of the same physical program. Not all events will be important to a visualization.
	Approach goals *(SQ1)*	Help the programmers to understand some dynamic aspects involved in code reuse and assist to better understand the structure of the software they themselves are developing (supporting programming for reuse, by dynamic display of the internal structure of software in development), and better identify potentially reusable components within the structure.
	Visualizations' reuse-specific goals *(SQ1)*	Make it easy to detect patterns of usage and patterns of dependence within a program – these patterns in turn help the programmer to determine how best to articulate the structure of a program using components that will be useful and independent for later reuse in other contexts.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development for reuse (programming for reuse / Dy-re addresses programming for reuse)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Discovering and evaluating potentially reusable assets (detect patterns of usage and dependence within a program / determine how best to articulate the structure of a program using components)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer for reuse (programmer)

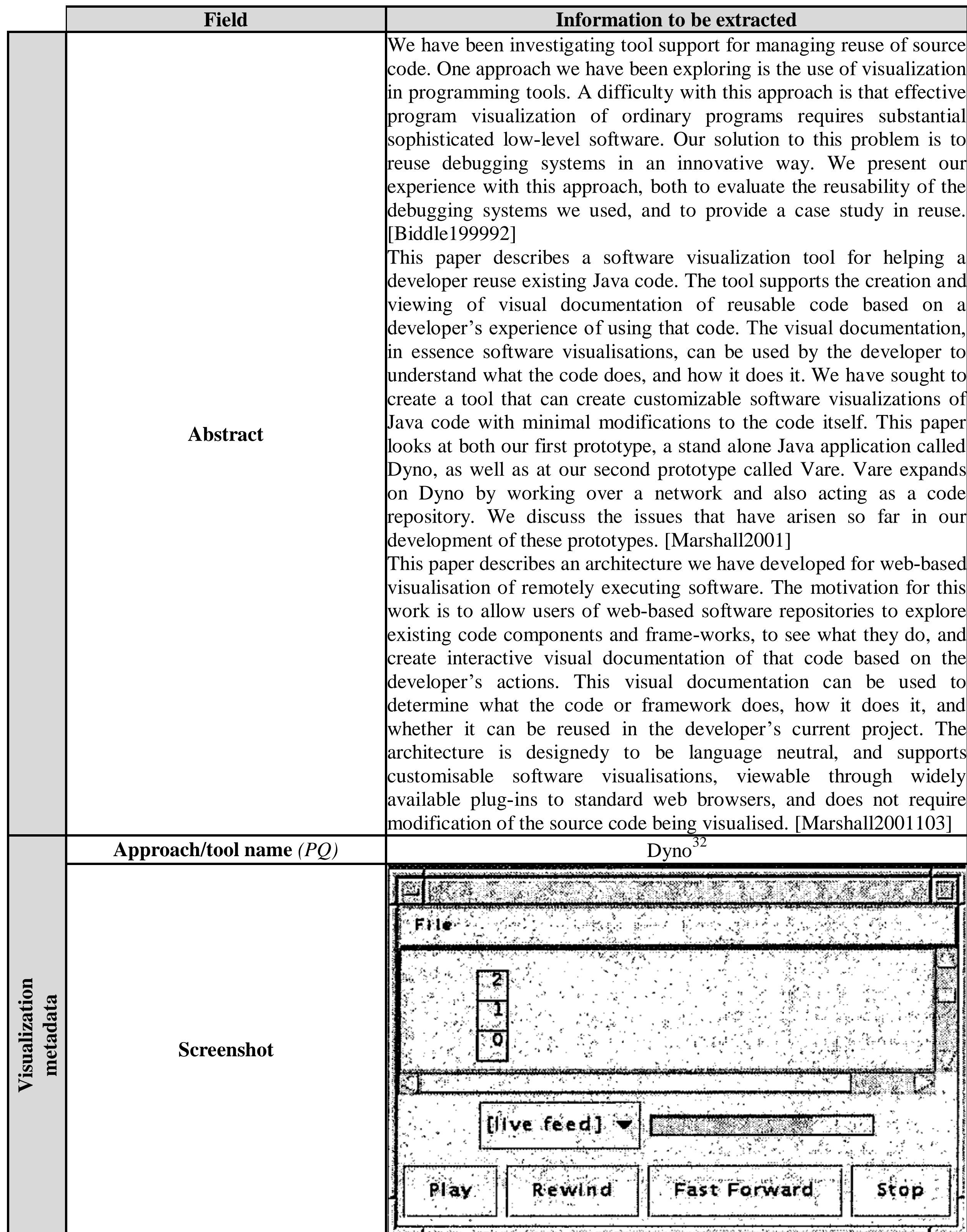
	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (C++ programs / code components and their connections)
	Source of visualized items/data *(TQ3.1)*	Information gathered at runtime from executing code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	General: The program component interacts with the target program. Its purpose is to determine what the target program is doing, and send information, in the form of actions, about what has occurred on to the mapping component. Action generation can come from three possible phases in program creation: at the source code phase, by modifying the source code (manually or automatically) to generate the actions, the compiler phase, by modifying the compiler to insert the actions, and at the executable phase, using post-processing to generate actions from information contained in the executable. Finally, actions can be generated directly from the executable. Post-processing the executable uses information contained in the compiled binary to generate events. Dy-re: The tool works with software being developed in C++. When the user selects the target program, Gdb reads in the debugging information contained within the executable. Gdb extracts the names of all the methods in the target program (excluding the basic system library routines). It then places breakpoints at the start and end of these methods so that the program will pause at these places. When the user runs the program, if a message call or return occurs, the program will pause. At this point it is possible to extract information about the current state of the program (for example, the call stack). The program is then restarted, so that from the user perspective, the visualizations appear to be updated continuously. Dy-re runs the target program, and the program may be paused, resumed, or restarted at any time. As the program runs, Dy-re displays diagrams. The trace view is based on the runtime call stack.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (sideways tree) • Diagrams (diagrams)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The assemblies are typically objects (instances of classes) and the dynamic structure of the program is shown where different objects are involved in different dynamic structures. The diagrams show objects as they are created. When one object calls a method of another object, a line is drawn between the objects, and an envelope is animated moving in one direction at the call, and back when the method returns. Any call still current is represented by a red line, and calls already complete are represented by black lines. The thickness of the line is increased as calls from distinct places are detected, to greater emphasize the occurrence of reuse, and highlight potential usefulness of particular program units. The calls from one object's method to another object's method are depicted from left to right on the horizontal axis. Any period of execution is shown as a sideways tree, with the root at the left, and any path through the tree represents the run-time stack at a particular time. Together the horizontal and vertical axes are used to show the run-time trace over a period of time.
	Visualization strategies and techniques *(TQ4.2)*	• Filtering (filtering out unwanted information) • Filtering / Tuning/Tweaking (dynamic visualization / allow the user to modify the graphical features of the display / the user could set display colors, arrange the layout, or focus the visualization on specific areas) • Filtering / Collapse/Expand (objects can be expanded) • Layout (when a method returns it remains shown, and later calls from the same method are then displayed below using the vertical axis) • Animation (create an animated display) • Sorting (displays the list of loaded classes ordered by the inheritance hierarchy used by Java) • Presentation / Simultaneous (multiple visualizations can be displayed simultaneously) • Overlap / Transparency (expanded objects are displayed transparently against the background diagram to aid comprehension that diagram reorganization might otherwise hinder)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The PMV (program-mapping-visualization) component model is applied (the extended version developed by Noble). For debugging C and C++ code, Gdb (the Gnu source level debugger) is used. Besides, the following frameworks and libraries are used: Expect, Tcl, and the Tk graphics library (Tcl/Tk), "Expectk" [Lib94], a combination of Expect and the Tk toolkit.

<table>
<tr><th></th><th>Field</th><th>Information to be extracted</th></tr>
<tr><td rowspan="4">Evidence (worthwhile)</td><td>Visualization evaluation methods (SQ7)</td><td>N/A</td></tr>
<tr><td>Application scenarios of the visualizations (TQ7.1)</td><td>N/A</td></tr>
<tr><td>Evaluated aspects (TQ7.2)</td><td>N/A</td></tr>
<tr><td>Visualization evaluation results/outcomes (TQ 7.3)</td><td>N/A</td></tr>
</table>

Table 30. Dyno [Biddle199992 / Marshall2001 / Marshall2001103]

	Field	Information to be extracted
Publication metadata	**Title**	Reuse of debuggers for visualization of reuse [Biddle199992] Using and Visualizing Reusable Code: Position Paper for Software Visualization Workshop [Marshall2001] Visualising reusable software over the web [Marshall2001103]
	Authors	Biddle, R., Marshall, S., Miller-Williams, J., Tempero, E. [Biddle199992] Marshall, S. [Marshall2001] Marshall, S., Jackson, K., McGavin, M., Duignan, M., Biddle, R., Tempero, E. [Marshall2001103]
	Publication date (year/month)	May, 1999 [Biddle199992] October, 2001 [Marshall2001] December, 2001 [Marshall2001103]
	Publication type	Conference [Biddle199992] Conference [Marshall2001] Conference [Marshall2001103]
	Source	Proceedings of the 5th Symposium on Software Reusability (SSR 1999) [Biddle199992] Workshop on Software Visualization, 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001) [Marshall2001] Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2001) [Marshall2001103]
	Volume and Edition (for journals)	N/A [Biddle199992] N/A [Marshall2001] N/A [Marshall2001103]
	Place (for conferences)	Los Angeles, USA [Biddle199992] Tampa, USA [Marshall2001] Sydney, Australia [Marshall2001103]
	Pages	pp. 92-100 [Biddle199992] ?? [Marshall2001] pp. 103-111 [Marshall2001103]
	Link (if applicable)	http://dx.doi.org/10.1145/303008.303037 [Biddle199992] http://www.research.ibm.com/people/w/wim/oopsla2001/papers/marshall.pdf [Marshall2001] http://dl.acm.org/citation.cfm?id=564053 [Marshall2001103]



	Field	Information to be extracted
	Abstract	We have been investigating tool support for managing reuse of source code. One approach we have been exploring is the use of visualization in programming tools. A difficulty with this approach is that effective program visualization of ordinary programs requires substantial sophisticated low-level software. Our solution to this problem is to reuse debugging systems in an innovative way. We present our experience with this approach, both to evaluate the reusability of the debugging systems we used, and to provide a case study in reuse. [Biddle199992] This paper describes a software visualization tool for helping a developer reuse existing Java code. The tool supports the creation and viewing of visual documentation of reusable code based on a developer's experience of using that code. The visual documentation, in essence software visualisations, can be used by the developer to understand what the code does, and how it does it. We have sought to create a tool that can create customizable software visualizations of Java code with minimal modifications to the code itself. This paper looks at both our first prototype, a stand alone Java application called Dyno, as well as at our second prototype called Vare. Vare expands on Dyno by working over a network and also acting as a code repository. We discuss the issues that have arisen so far in our development of these prototypes. [Marshall2001] This paper describes an architecture we have developed for web-based visualisation of remotely executing software. The motivation for this work is to allow users of web-based software repositories to explore existing code components and frame-works, to see what they do, and create interactive visual documentation of that code based on the developer's actions. This visual documentation can be used to determine what the code or framework does, how it does it, and whether it can be reused in the developer's current project. The architecture is designedy to be language neutral, and supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, and does not require modification of the source code being visualised. [Marshall2001103]
Visualization metadata	**Approach/tool name** *(PQ)*	Dyno[32]
	Screenshot	

[32] For *dynometer* [Biddle199992]

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Achieving effective software reuse is a difficult problem in itself, one that requires more support than has generally been available. In many cases of code reuse, only the compiled code is available and the developer can not inspect the source. Moreover, the reusable code is not typically an entire application in of itself. In this context, writing trial programs, or "test-harnesses", to explore how to use a component is a common practice. Such programs typically invoke the methods of the public interface of an object and then display the results returned and the resultant state of the object so the programmer can check they are consistent with expectations. This is not a substitute for understanding of a component specification, but can be of assistance in better understanding practicalities of actually using a component. In order to determine the correct filtering, some input from the use is required. However, the component writer and the visualization template writer are not necessarily the same person, and each writer has total control over the nomenclature used in their code. This sometimes creates a problem as the purpose of the test driving is to understand the component, and a developer may not know which methods should map to which sequence. It is bordering on the impossible for a tool to be able to automatically create mappings from the arbitrary for the developer to say which method in the component maps to which sequence. Note that this can be a one-to-one mapping, or a many-to-one mapping. It is necessary to explore the development of tools to explicitly support the reuse process, in particular how to best help a programmer understand the reusable software well enough so as to be able to use it effectively [Marshall2001] [Marshall2001103]. At the point in the process where a programmer has identified and retrieved a candidate component, such programmer needs to verify that the component will actually meet their requirements, and then needs to determine how to actually use the component. Both these steps will require some level of understanding of what the component actually does. For the fact that the user may want to review visualizations of a component without having to re-execute the method and re-set its state, the visualizations should be stored separately. [Biddle199992].
	Approach goals *(SQ1)*	Help the programmers to understand some dynamic aspects involved in code reuse and help to better understanding the correct usage and functionality of a component they are considering reusing, allowing the programmer to take the component on a "test drive" in order to understand the behavior of a Java component [Biddle199992]; help a developer reuse existing Java code by creating customizable software visualizations (dynamic documentation) of Java executing code (getting all the necessary runtime information) with minimal modifications to the code itself (i.e., "test-drive" by exploring the behavior of a reusable Java component interactively), and provide the developer with a deeper understanding of what the component does, and how it does it, thus helping to decide if and how the component can be reused [Marshall2001] [Marshall2001103].
	Visualizations' reuse-specific goals *(SQ1)*	Make code reuse more appealing with a better understanding what the code does, and how it does it.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (programming with reuse / Dyno addresses programming with reuse / reuse of software components)

	Field	Information to be extracted
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' behavior (better understand what the code does, and how it does it)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer with reuse (programmer / developer / a developer wishing to understand how a component works or what a component does) • Developer for reuse (a component writer who wishes to create visual documentation of their own components)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Source code and related information (Java programs / code components (components are individual – or groups of – Java classes) / list of loaded classes / methods and fields on that object / the static methods and the static fields of that class / for objects, the methods and fields for its supertypes / what methods were called when, or who, by who, and with what arguments, what methods returned when, and with what return values, what fields were accessed, and when, what fields were modified, when, what they modified to, from, and by who, what exceptions were thrown/caught and when / data structures) • Architecture / Design artifacts and related information (UML diagrams)
	Source of visualized items/data *(TQ3.1)*	Information gathered at runtime from executing code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	General: The program component interacts with the target program. Its purpose is to determine what the target program is doing, and send information, in the form of actions, about what has occurred on to the mapping component. Action generation can come from three possible phases in program creation: at the source code phase, by modifying the source code (manually or automatically) to generate the actions, the compiler phase, by modifying the compiler to insert the actions, and at the executable phase, using post-processing to generate actions from information contained in the executable. Finally, actions can be generated directly from the executable. Post-processing the executable uses information contained in the compiled binary to generate events. Dyno: Visualizations are created from information gathered at runtime from executing Java code. A visualization template is required so that the tool uses it to determine what to draw, and when to draw it (visualization templates describe the type of information to display). The templates then use the information gathered at runtime to flesh out concrete visualizations. Visualization templates are written in Java, and component writers may even write visualization templates that work specifically with their component rather than use general purpose templates. That also means that the component writer may create a template which will not work if it is used on any other component. Visualizations are created from events that occur as code executes during a test drive. Dyno uses JVMDI to place breakpoints at the beginning of all methods (both instance and static) other than those belonging to classes in the JDK distribution or those that comprise Dyno itself. The user can then request that methods be executed on an object or class in the system, and Dyno's JVMDI code will detect when methods are first entered, forwarding appropriate information to the mapping component for further processing so that visualization data can be produced. A view in the visualization component reads in the information it needs for the attributes from a file written out by the mapping component. This sequence of events is then displayed one at a time creating an animation as the state of the component being examined changes.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (binary tree diagram) • Diagrams (UML sequence diagram) • Others (not limited to any particular set of pre-determined visualizations)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Information about events can be used to map to a frame or sequence of frames in the visualization. In order to successfully map events in the code to visualization sequences, there is the need for at least some mapping information.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (allows the user to select a class, object, or primitive, and will then proceed to display information concerning the methods and fields on that object / the user can select objects from the object browser to be passed to the method invocation) • Filtering (filtering out unwanted information / filtering the information to locate only that information that is needed for the visualization) • Filtering / Tuning/Tweaking (dynamic visualization / allow the user to modify the graphical features of the display / the user could set display colors, arrange the layout, or focus the visualization on specific areas) • Animation (create an animated display / animation) • Presentation / Simultaneous (multiple visualizations can be displayed simultaneously) • Linking (if an object is selected in the object browser, the methods and fields for that object are displayed in the panel to the right)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Dyno is a stand-alone Java application, which uses the Java Virtual Machine Debugger Interface (JVMDI). The PMV (program-mapping-visualization) component model is applied (the extended version developed by Noble). The Java Native Interface (JNI), serialization API, and the Java Reflection API are also used.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (described in [Marshall et al. 2002][33], cited by Marshall2001103) (usability evaluations)
	Application scenarios of the visualizations *(TQ7.1)*	Not specified
	Evaluated aspects *(TQ7.2)*	Usability (described in [Marshall et al. 2002], cited by Marshall2001103)
	Visualization evaluation results/outcomes *(TQ 7.3)*	High usability is an important but difficult element of the software tool design (described in [Marshall et al. 2002], cited by Marshall2001103)

[33] [Marshall et al., 2002] Marshall, S., Biddle, R., Tempero, E. (2002). "How (not) to help people test drive code". In: *3rd Australasian User Interface Conference (AUIC 2002)*, Melbourne, Australia, pp. 39-42, January.

Table 31. Nested Software Self-Organising Map (NSSOM) [Ye2000266]

	Field	Information to be extracted
Publication metadata	**Title**	A visualised software library: Nested self-organising maps for retrieving and browsing reusable software assets
	Authors	Ye, H., Lo, B. W. N.
	Publication date (year/month)	??, 2000
	Publication type	Article (Journal)
	Source	Neural Computing and Applications
	Volume and Edition (for journals)	v. 9, n. 4
	Place (for conferences)	N/A
	Pages	pp. 266-279
	Link (if applicable)	http://dx.doi.org/10.1007/s005210070004
	Abstract	This paper presents an approach to self-structuring software libraries. The authors developed a representation scheme to construct a feature space over a collection of software assets. The feature space is represented and classified by a variety of the self-organising map, called the Nested Software Self-Organising Map (NSSOM), consisting of a top map and a set of sub-maps nested in the top map. The clustering on the top map provides general improvements in retrieval recall, while the lower-level nested maps further elaborate the clusters into more specific groups enhancing retrieval precision. The results of preliminary evaluation showed that NSSOM is capable of enhancing precision without sacrificing recall. In addition, a user-friendly browsing facility has also been developed which helps users predict the desired components by providing an intelligible search space. The present approach attempts to achieve an optimal combination of efficiency, accuracy and user-friendliness, which is not offered by the existing software retrieval systems.
Visualization metadata	**Approach/tool name** *(PQ)*	Nested Software Self-Organising Map (NSSOM)
	Screenshot	

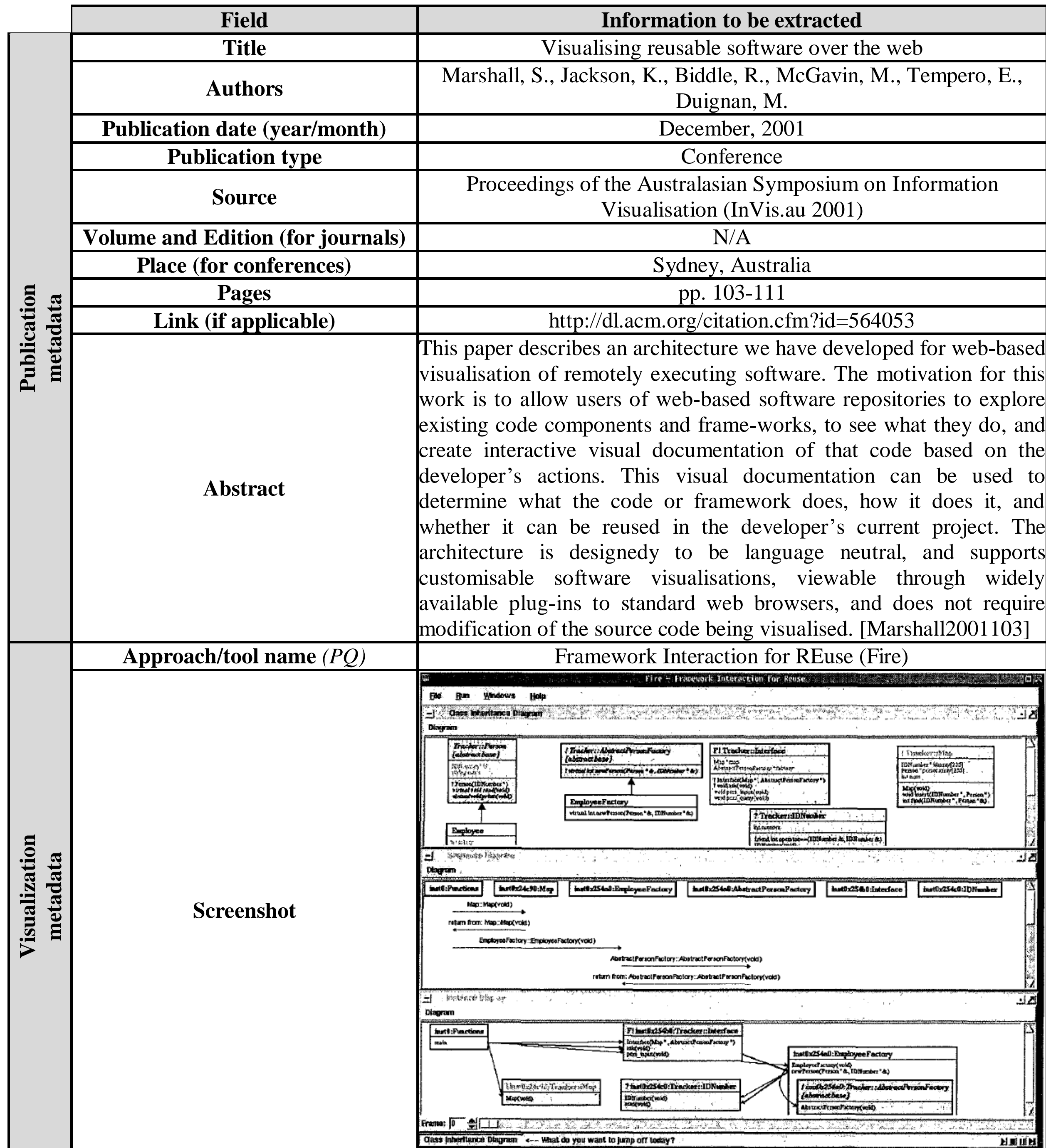
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The two key issues in developing a software library are: (a) what information about software components is necessary to be stored in the library; and (b) how this information is to be organised. Although a large amount of research has been undertaken on the techniques of structuring software libraries for the storage and retrieval of software assets, none of them offers the right combination of efficiency, accuracy and user-friendliness to facilitate a breakthrough in the practice of software reuse. Controlled vocabularies and knowledge-based techniques need intensive human effort in manual indexing or knowledge acquisition and representation. Although the human effort can be minimised in uncontrolled vocabularies by using automatic indexing, the semantic relationship among software components are often obscured. A single Software Self-Organising Map (SSOM) is only suitable for a small software collection. When a software library grows in size, recall can be maintained while precision tends to go wrong. After a SSOM is established (i.e. a classified library of software assets is formed), it should be able to accommodate a continually expanding collection of components. The need of continuous change is an inevitable process in most software organisations. For a large library, a set of nested semantic maps may serve as a set of catalogues of the library at different levels. Browsing is an exploratory, information seeking strategy that depends on serendipity. Navigating through large software libraries without any guide to help users predict the desired components can be very frustrating. The users may not know where to start the navigation or may become disoriented during the browsing when facing a sophisticated information space. Besides, manually interpreting the semantic meanings of a map is a labour-intensive and tedious task, especially for very high-dimensional feature spaces. As the users of software libraries may deal with a particular software library for a relatively longer time than the casual users in a general information library, the accumulated knowledge with regard to the content and organisation of the library gained from the previous browsing will substantially improve their searching skills later.
	Approach goals *(SQ1)*	Make software libraries self-structuring.
	Visualizations' reuse-specific goals *(SQ1)*	Help users predict the desired components by providing an intelligible search space, and provide a whole picture of the library at a relatively general level for finding some interests in certain subareas of the semantic map.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (software reuse / exploration of a software library)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (provide a whole picture of the library for finding some interests in certain subareas) • Searching and retrieving reusable assets (help to predict the desired components / retrieval of software assets)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	User (users)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (software asset, software components / semantic relationships among software components) • Software repository and related information (software library)
	Source of visualized items/data *(TQ3.1)*	The feature space of a software library.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The feature space of a software library is presented to the SOM as its input data. After removing common function words and stemming, each word in a document is assigned a weight based on how often it occurs in the document, and how rarely it occurs in the remainder of the collection. Single terms with a sufficient high weight are selected as indices. The indices need to be grouped into concept classes and stored in a thesaurus. The number of the total features in a corpus is the dimension of the input feature vectors. Each document can be represented as an input feature vector, each element of which corresponds to a certain feature. If a feature exists in a document, the value of the correspondent element is the weight of the feature, otherwise the value is zero. All the vectors will form an input feature space and be presented to a SOM. The node with the maximum similarity is selected as the winning node. Such node c and its neighbouring nodes will learn something from the current input x, and their weights will be modified to make them more responsive to the current input (updating reference vectors). Selecting winning nodes and updating weight vectors should be iterated many times until a steady convergence is obtained. The size of the subspace is defined by a distance threshold (dt). All the neighbouring nodes that have a distance less than the dt with centroid node c will form a subspace for the nested map. If the number of components located in a subspace is less than the predefined threshold, called ct, the corresponding NM does not need to be constructed. The nested depth is defined by an integer to denote which level the corresponding NM is located. The features with higher frequency are considered the better representations of semantic relationships among the components. A set of single feature vectors whose correspondent feature has a frequency greater than a pre-defined threshold is selected. By comparing each reference vector to the set of single feature vectors in terms of their Tanimoto similarity, the feature containing in the winning single vector will be assigned to the corresponding node. This latter step is repeated until all the nodes are assigned a feature.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (a top map and a number of nested maps (NM)) • Map (map, semantic map, dot on the map)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each node in the grid denotes an artificial neurone and is topologically connected with each other. Software components with similar functions are mapped onto adjacent areas. Each input vector is mapped to a node closest to it. The winning nodes responding to the input vectors are marked with the name of the corresponding UNIX commands. The Top Map is the component map of a whole software component collection, and the Nested Maps are software component maps of subset of the whole collection. Each dot on the map indicates a node and the numbers shown near individual nodes indicate how many components are located in the corresponding nodes. Small rectangles shown in the map indicate that there is a nested map for all the components locating in the corresponding area. A component space (representing the content of a software library) and a feature space (explaining the semantic meanings of the components stored in the library) are incorporated into a set of semantic maps. Semantic explanations are the major features distributed to the different regions of the maps, indicating the functions associated with the components located in the corresponding regions.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (when this item is selected, the main features are displayed in the list box and a NSM will be displayed (if it exists)) • Browsing / Navigation (user-friendly browsing facility / direct browsing) • Browsing / Querying (keyword-based browsing / based on the keyword(s) or short statements submitted to the system) • Details on demand / Drill-down (when this item is selected, the main features are displayed in the list box and a NSM will be displayed (if it exists)) • Details on demand / Labeling (nodes responding to the input vectors are marked with the name of the corresponding UNIX commands / numbers shown near individual nodes) • Clustering (clustering / software components with similar functions will be mapped onto adjacent areas of the output layer / nodes geographically close to each other will have similar weights / a top map and a set of sub-maps nested in the top map) • Filtering / Highlighting/Mitigation (makes the most important semantic relationships among the data items geometrically explicit) • Overview + detail (users are able to get a whole picture of the library from the TSM at a relatively general level and may find some interests in certain subareas of the TSM)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	• Mouse • Keyboard

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (retrieval experiment) [similar to a benchmark analysis]
	Application scenarios of the visualizations *(TQ7.1)*	1) A small test collection containing 97 manual pages of the most commonly used Unix commands. 2) A document collection consisting of more than 440 Unix manual pages. [open source]
	Evaluated aspects *(TQ7.2)*	1) Retrieval effectiveness of the NSSOM (measured by recall and precision) compared with a public retrieval system – Personal Librarian (PL). 2) Retrieval effectiveness of the NSSOM compared with Guru, a representative of Information Retrieval based methods in software storage and retrieval, and with the PL (by using an expanded query set with all of the Guru's queries and additional queries obtained from a survey).
	Visualization evaluation results/outcomes *(TQ 7.3)*	NSSOM is capable of enhancing precision without sacrificing recall. Improvement on precision was observed in comparison with a similar software retrieval system and a publicly available full-text retrieval system: 1) SSOM achieved a higher precision at the same level of recall than PL. 2) Not only was NSSOM significantly better than PL, but also achieved better precision at the same level of recall than Guru.

Table 32. Framework Interaction for REuse (Fire) [Marshall2001103]

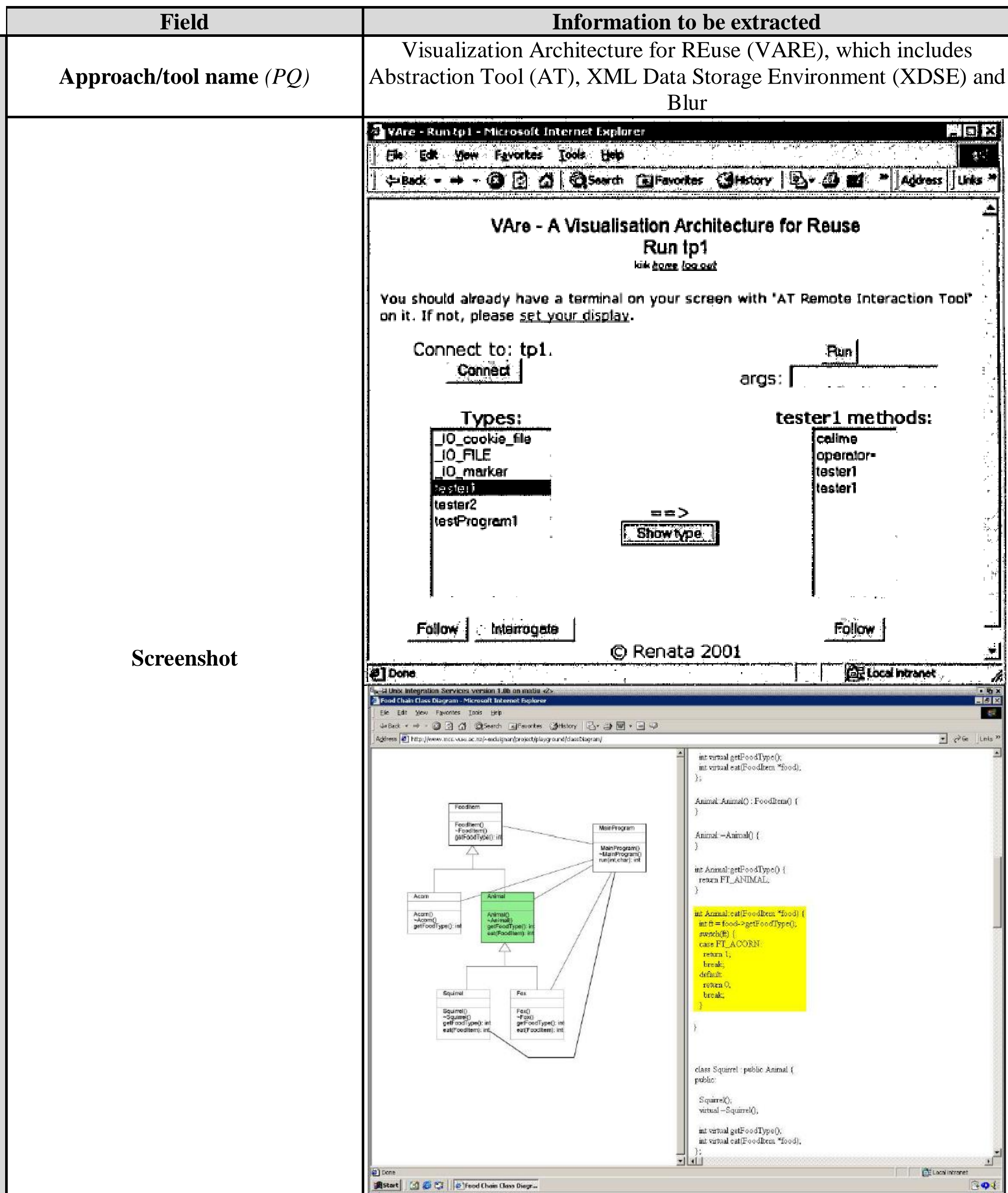
	Field	Information to be extracted
Publication metadata	**Title**	Visualising reusable software over the web
	Authors	Marshall, S., Jackson, K., Biddle, R., McGavin, M., Tempero, E., Duignan, M.
	Publication date (year/month)	December, 2001
	Publication type	Conference
	Source	Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2001)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Sydney, Australia
	Pages	pp. 103-111
	Link (if applicable)	http://dl.acm.org/citation.cfm?id=564053
	Abstract	This paper describes an architecture we have developed for web-based visualisation of remotely executing software. The motivation for this work is to allow users of web-based software repositories to explore existing code components and frame-works, to see what they do, and create interactive visual documentation of that code based on the developer's actions. This visual documentation can be used to determine what the code or framework does, how it does it, and whether it can be reused in the developer's current project. The architecture is designedy to be language neutral, and supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, and does not require modification of the source code being visualised. [Marshall2001103]
Visualization metadata	**Approach/tool name** *(PQ)*	Framework Interaction for REuse (Fire)
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Using a framework requires an intimate understanding of how it will interact with the extensions that the programmer will provide – knowledge that has to be learned. To acquire knowledge of a framework's behaviour can take a great deal of effort, and increases the costs of development the first time a framework is used. This is seen as one of the major problems with frameworks.
	Approach goals *(SQ1)*	Support the visualization of framework interactions, which aids the identification and understanding of the critical interactions between framework and user objects.
	Visualizations' reuse-specific goals *(SQ1)*	Help to understand how the frameworks are used.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (code reuse / produce a new application)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' behavior (understand how the frameworks are used)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (advanced programmer)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (C++ frameworks)
	Source of visualized items/data *(TQ3.1)*	Works with C++ programs compiled in the usual way.
	Collection procedure/method of visualized items/data *(TQ3.2)*	N/A
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Diagrams (diagrams based on UML / a static UML class diagram, a UML sequence diagram, and an instance diagram)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	N/A
	Visualization strategies and techniques *(TQ4.2)*	• Filtering / Tuning/Tweaking (allows the user to enlarge or reduce their sizes) • Filtering / Inclusion/Removal (hide irrelevant classes or objects) • Panning / Drag-and-drop (move objects around on the screen to customise the layout) • Animation (animated, and updated immediately when the events are received) • Presentation / Simultaneous (allows simultaneous viewing)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (over the web / networked computers)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 33. Visualization Architecture for REuse (VARE) [Marshall2001103 / Anslow2004]

	Field	Information to be extracted
Publication metadata	**Title**	Visualising reusable software over the web [Marshall2001103] Software visualization tools for component reuse [Anslow2004]
	Authors	Marshall, S., Jackson, K., Biddle, R., McGavin, M., Tempero, E., Duignan, M. [Marshall2001103] Anslow, C., Marshall, S., Noble, J., Biddle, R. [Anslow2004]
	Publication date (year/month)	December, 2001 [Marshall2001103] October, 2004 [Anslow2004]
	Publication type	Conference [Marshall2001103] Conference [Anslow2004]
	Source	Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2001) [Marshall2001103] 2nd Workshop on Method Engineering for Object-Oriented and Component-Based Development, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004) [Anslow2004]
	Volume and Edition (for journals)	N/A [Marshall2001103] N/A [Anslow2004]
	Place (for conferences)	Sydney, Australia [Marshall2001103] Vancouver, Canada [Anslow2004]
	Pages	pp. 103-111 [Marshall2001103] ?? [Anslow2004]
	Link (if applicable)	http://dl.acm.org/citation.cfm?id=564053 [Marshall2001103] http://dx.doi.org/10.1.1.91.7214 [Anslow2004]
	Abstract	This paper describes an architecture we have developed for web-based visualisation of remotely executing software. The motivation for this work is to allow users of web-based software repositories to explore existing code components and frame-works, to see what they do, and create interactive visual documentation of that code based on the developer's actions. This visual documentation can be used to determine what the code or framework does, how it does it, and whether it can be reused in the developer's current project. The architecture is designedy to be language neutral, and supports customisable software visualisations, viewable through widely available plug-ins to standard web browsers, and does not require modification of the source code being visualised. [Marshall2001103] This paper describes our experiences with our software visualization tools for web-based visualization of remotely executing object-oriented software. The motivation of this work is to allow developers to browse web-based software repositories to explore existing code components and frameworks by creating visual documentation. Components are test driven to capture their static and run-time information in program traces and are then transformed into useful visualizations. Visualizations can help developers understand what a component does, how it works, and whether or not it can be reused in a new program. [Anslow2004]

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Visualization Architecture for REuse (VARE), which includes Abstraction Tool (AT), XML Data Storage Environment (XDSE) and Blur
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The main reasons for wanting to reuse components are to save on time, effort, and costs in both development and maintenance of quality software. This will mean the developer will not have to implement a new solution to an old problem. Instead they can recycle existing components to solve their problem. There are many ways component reuse can be applied. For example, copying and pasting code into a new program, inheritance of classes, instantiation of common methods within programs, using a framework, and using an application programming interface. When reusing a component it may need to be modified or extended in some way so that it will meet the requirements of the new program. The assumption is that even modifying or extending a component will result in the reduction of time, cost and effort compared with designing the component from scratch. A key benefit from reusing components is that when modifications, bug fixes or updates occur, the developer can save time by incorporating them into their program. Problems then don't have to be solved for every instance. This can happen on a global scale and examples include online updates of both proprietary and open source software. To reuse a software component, developers need to understand what a component does, how it works, and how it can be reused. For what a component does, it is important to look at the external side-effects and the results that occur as a consequence of interacting with a component's public interface. For how a component works, it is important to look at the internals of a component. This is because it may open up opportunities for modifying the component's behavior to what is required by replacing sub-components, extending components or overloading methods. For how a component can be reused or modified, it is important to look at how it has previously been used. Helping developers understand components by creating visualizations means that they will potentially be able to reuse a component in a new program. However, this is difficult in practice. Currently, several techniques exist to help understand how software works and these include documentation, experimenting, and visualizations. Documentation is sometimes provided with software either in online or in written form, but is often difficult to use, read and understand. Experimenting with reusable components means that developers will gain practical experience and learn how components work. Visualizing a component's static or run-time information can show developers how a component has been designed, and how it works when executed. However, to visualize a design or a software component, certain information has to be selected. Extracting the correct information and gathering it in program traces is a difficult procedure. Program traces are expensive to generate because they are extremely large and take a long time to create. There are many factors which can affect this procedure, such as the language a component is written in, or the design complexity. [Anslow2004] [Marshall2001103].
	Approach goals *(SQ1)*	Explore existing reusable code components and frameworks by creating visual documentation, besides storing and retrieving program traces.
	Visualizations' reuse-specific goals *(SQ1)*	Help to understand what a component does, how it works, and whether or not it can be reused in a new program.

	Field	Information to be extracted
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software development with reuse (reuse a component in a new program / development) • Software maintenance (maintenance)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' behavior (understand what a component does and how it works) • Integrating reusable assets (understand whether a component can be reused in a new program)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (developers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (reusable components)
	Source of visualized items/data *(TQ3.1)*	Test Driver: C++ and Java programs / the source or binary files. AT: programs written in C++. Blur: events in the run-time environment.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Components are test driven to capture their static and run-time information in program traces and are then transformed into useful visualizations. Test driving (defined as "specifying a sequence of method invocation and field access/modifications and then executing the sequence on a component") generates static and run-time information about a component such as class descriptions and the methods that have been invoked on objects. Two XML based program trace languages were created for describing object-oriented programs. Program traces are stored in an XML database and can be queried and then transformed into Scalable Vector Graphics (SVG) visualizations. The approach examines or spies on programs during execution and gathers events in a program trace. The design of VARE supports multiple programming languages. On the client side, the user manages the activities associated with creating and viewing a visualization. The component repository interface lets the user select a component from the repository to create a component set. Once this is created, the user can select an engine type from the engine repository to control the test driving of these components. The engine generates a program trace/test drive trace as output, which is stored in the test drive report repository. A program trace is then used as input to a transformer. The transformer repository interface lets the user select the transformer to use and the program trace to use with it. The transformer then transforms the program trace into an appropriate visualization. Finally the finished visualization is stored in the visualization repository. Blur takes a PAL program trace and transforms it into a Scalable Vector Graphics (SVG) visualization.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Diagrams (static visualizations / run-time visualizations / UML class diagram / sequence diagram)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The user selects the transformer to use and the program trace to use with it. The transformer then transforms the program trace into an appropriate visualization.
	Visualization strategies and techniques *(TQ4.2)*	• Browsing / Navigation (navigate) • Filtering / Highlighting/Mitigation (highlights) • Filtering / Tuning/Tweaking (provides user control for the different parts in the visualization process) • Filtering / Collapse/Expand (fold and unfold call sequences) • Zooming / Geometric (zoom-in-out)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (web environment / over the web)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Apache/Jakarta Tomcat. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Scalable Vector Graphics (SVG), Program Mapping Visualization (PMV) conceptual model, Process Abstraction Language (PAL), GNU Debugger (GDB). AT is written in the Python scripting language, and uses SOAP for remote method invocation. Reusable Component Descriptions (RCD) are used for static information, and eXtensible Trace Executions (XTE) for dynamic information. XDSE is implemented with an Ipedo native XML database, SOAP, Apache Tomcat, and XQuery. Blur is implemented as a Java Servlet.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A ("in past work on software tools to provide visualisation of reusable software components, usability evaluations were conducted")
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 34. Mittermeir et al.'s approach [Mittermeir200195]

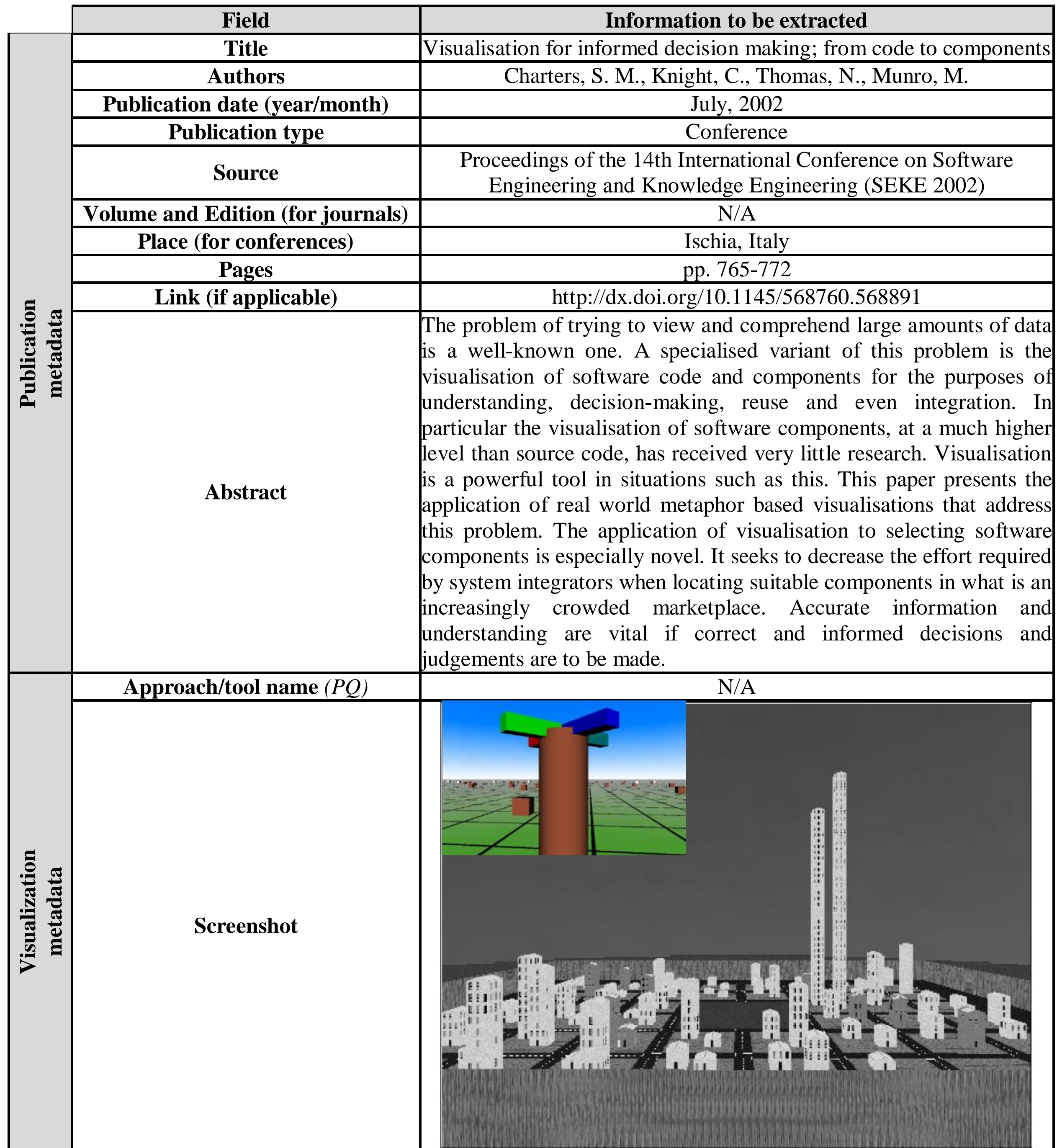
	Field	Information to be extracted
Publication metadata	**Title**	Goal-driven combination of software comprehension approaches for component based development
	Authors	Mittermeir, R. T., Bollin, A., Pozewaunig, H., Rauner-Reithmayer, D.
	Publication date (year/month)	May, 2001
	Publication type	Conference
	Source	Proceedings of the 2001 Symposium on Software Reusability (SSR 2001)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Toronto, Canada
	Pages	pp. 95-102
	Link (if applicable)	http://dx.doi.org/10.1145/375212.375264
	Abstract	This paper reports on our approaches to combine various software comprehension techniques (and technologies) in order to establish confidence whether a given reusable component satisfies the needs of the intended reuse situation. Some parts of the problem we are addressing result from differences in knowledge representation about a component depending on whether this component is a well documented in-house development, some externally built componentry, or a COTS-component.
Visualization metadata	**Approach/tool name** *(PQ)*	N/A
	Screenshot	N/A

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	One of the key issues causing the Not-Invented-Here syndrome remains: How can developers be sure that the component they plan to use in their new construction venture meets the expectations placed into it. In general, people are not interested in (and not capable for) comprehending a larger piece of software in its entirety – they are just interested to know, whether it does what they want it to do and whether it does not what it should not do. A software engineer does not need to know every detail about a component to be integrated. S/he does need to know though, whether the component at hand renders the required functionality and whether in doing so it would not occasionally spoil parts of the system to be built by performing unwanted functionality. The reuser, specifically the COTS integrator, might not have all the information a maintainer might have at hand; the reuser even more than the maintainer might be interested in perusing the component at hand at various levels of granularity. The reuser does need an adequate proxy for a full comprehension though. Hence, a suitable way (perhaps the only way) is to correlate partial evidence to form a hypothesis about what the piece at hand actually is all about and then use further clues to either stepwise support this hypothesis until a level of satisfactory trust is reached or to disprove it. Specific mechanisms have to be devised in order to compensate for the loss of in-depth informal information. It is important to foresee that the software to be integrated will come at different levels of representation and it will be supported by different degrees of documentation. There are differences in knowledge representation about a component depending on whether this component is a well documented in-house development, some externally built componentry, or a COTS-component. If only binaries are available, the armory for checking whether the respective software actually is what it is supposed to be will be rather limited (although the situation is not hopeless). But if other forms of documentation are available, the set of comprehension aids will correspondingly become larger, thus allowing for more efficient analysis. The role of source code in its original textual form is quite limited, if the component to be analyzed is beyond a critical size. It is known from source-code comprehension, that the geometrical arrangement of statements substantially aids comprehensibility. The same argument applies for specification styles and the respective arrangement of properties and terms. It is important to verify whether the specification is still a valid high level representation of the code at hand. The comprehension problem is to build oneself a conceptual model about a piece of software; the representational form changes depending on the level of abstraction and the level of comprehensiveness of such a model. Humans do not comprehend (and mentally manipulate) conceptual models when facing them as long strings of text. We have to accept that “full comprehension” of some sufficiently sizable piece of code is impossible anyway. Thus, the proper combination of partial representations of software, be it representations by means of test/trace-data, by means of source-code or by means for formal specifications as well as variations in the level of abstraction and presentation will help software engineers to more effectively and more efficiently establish the trust needed to integrate a component which is not invented here.

	Field	Information to be extracted
	Approach goals *(SQ1)*	Establish confidence whether a given reusable component satisfies the needs of the intended reuse situation (i.e., investigate reusable components) and identify whether the hidden state of such an object (class) satisfies the properties a reuser is expecting from the piece of code at hand, by combining various software comprehension techniques (and technologies).
	Visualizations' reuse-specific goals *(SQ1)*	Support the comprehension task.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (investigating reusable components / COTS integration)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (support the comprehension task) • Integrating reusable assets (establish confidence whether a given reusable component satisfies the needs of the intended reuse situation)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer / programmer (software developer) • Developer with reuse (reusers) • Software engineer (software engineer)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (component / trace sequences represented by different test cases, dynamic models (state charts), data-points) • Source code and related information (state-bearing software (objects, classes), source code) • Requirements/Analysis artifacts and related information (specification)
	Source of visualized items/data *(TQ3.1)*	Source code and component's test logs.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Some approaches to better grasp the semantic content of source code are partitioning and visualization. Partitioning may require some kind of slicing. Other partitioning strategies are declaration analysis, signature analysis, chunking etc. They are valuable for focused partial comprehension. Figuratively, a chunk can be seen as a "horizontal" portion of code while a slice is a "vertical" portion cut along the data and control-flow. For deriving an understandable and interpretable behavioral model automatically by analyzing the effect (the test data) of software directly, descriptions can be done based on the (relatively simple) model of finite state machines, knowing that this covers only a modest portion of potential software. Trace data (i.e., sequences of function calls invoking a complex component) are available by analyzing test logs. Traces are obtained by analyzing the component's test logs. Dynamic models (state charts) can be derived from (object-oriented) source code. Variables, respective object attributes maintaining state information are identified. From that attributes, the potential states is inferred by analyzing conditions in the control flow graph. Such identified states are the basis for revealing state transitions by looking at the variable's value changes. By considering all identifiable objects of one class, many potential modification curriculi can be derived. The combination of all potential modification curriculi represents the final dynamic model of the component.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (deterministic finite automaton)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each line represents the sequence of method calls during a "life cycle" of a component. Colors can be used to model more than two dimensions on the plane of a sheet of paper or a video screen.
	Visualization strategies and techniques *(TQ4.2)*	• Details on demand / Drill-down (cross-level trace from a specification to its implementation) • Filtering / Highlighting/Mitigation (syntax highlighting in source-code helping to provide some specific focus on textual representations / using graphics (or color) to highlight relatedness) • Layout (partitioning for supporting comprehensibility by reduction in volume) • Animation (animation) • Presentation (multiple representations (of different views) / visualization (transformation into whatever form of two or even higher dimensional representations) as a means to spread out information in various dimensions)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 35. Charters et al.'s approach [Charters2002765]

	Field	Information to be extracted
Publication metadata	**Title**	Visualisation for informed decision making; from code to components
	Authors	Charters, S. M., Knight, C., Thomas, N., Munro, M.
	Publication date (year/month)	July, 2002
	Publication type	Conference
	Source	Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Ischia, Italy
	Pages	pp. 765-772
	Link (if applicable)	http://dx.doi.org/10.1145/568760.568891
	Abstract	The problem of trying to view and comprehend large amounts of data is a well-known one. A specialised variant of this problem is the visualisation of software code and components for the purposes of understanding, decision-making, reuse and even integration. In particular the visualisation of software components, at a much higher level than source code, has received very little research. Visualisation is a powerful tool in situations such as this. This paper presents the application of real world metaphor based visualisations that address this problem. The application of visualisation to selecting software components is especially novel. It seeks to decrease the effort required by system integrators when locating suitable components in what is an increasingly crowded marketplace. Accurate information and understanding are vital if correct and informed decisions and judgements are to be made.
Visualization metadata	**Approach/tool name** *(PQ)*	N/A
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	New and novel methods for the selection of components are required to ease the burden on the component purchaser. Whilst visualisation is not a 'silver bullet' to the problem it has many advantages that can be brought to bear on the problem of component selection. The use of abstract visualisation features has already received a small amount of attention in the software visualisation field and a real world visualisation was decided to be worth investigation. The basic ideas of exploiting metaphors and space to represent essentially intangible attributes have proved useful with both code and components.
	Approach goals *(SQ1)*	Increase the understanding of a given code and aid any future development and maintenance, by providing a mechanism in which informed decisions can be made.
	Visualizations' reuse-specific goals *(SQ1)*	Provide an easily navigable environment with a shallow learning curve for non-expert users allowing them to select components based on multiple attributes and find a selection of components that could possibly be used in the development of their system.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (start of a project / finding and evaluating components for the system)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Searching and retrieving reusable assets (find a selection of components that could possibly be used in the development of their system / allow to select components based on multiple attributes)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer with reuse (system integrators / developers of software systems)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (software components) • Source code and related information (Java source code) • Software repository and related information (static representation of a software component repository)
	Source of visualized items/data *(TQ3.1)*	An XML representation of the software component repository.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The generation for the visualisation uses an XML representation of the software component repository detailing software components and their functional properties. This XML file is fed into a self-organizing map that is used to group the components based on their functional properties. The results from the self-organizing map are then transformed using XSLT into a VRML model of Component City that can be viewed within any VRML Viewer or Web Browser with appropriate plug-in.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Real world metaphor (cityscape, world, country, city, districts, streets/buildings/gardens/monuments, elements inside buildings/gardens)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The colour of the buildings represents whether the method the building represents is private or not. The height is a representation of the number of lines of code. Similar components, sharing the same functional properties, appear close to each other. Each building has a minimum height of one storey and a roof, but for each extra ten lines of code an extra storey is added to the building. All buildings have a blue door – the reason being that a method may have no parameters and there still needs to be a logical way for a user to enter the building when exploring the environment. Parameters are shown in number and type (at a basic level) by extra doors. A door with either yellow or green paintwork shows all method parameters; the formal type of the parameter determines the exact colour. World is a flattened overview of Component City showing the distribution of components, City is a more detailed overview highlighting areas representing functionality groups, District is a functionality group (components are clustered into districts based on their functional properties and their relationship to other components), and Building, that represents a single component or multiple very similar components. Buildings can be houses, which represent a single component, Mansions, which represent two components, and Skyscrapers, which represent more than two components (with skyscrapers the number of levels indicates the number of components at that location). The top of the monument has four arms, each of a different colour; that arm corresponds to the colour of the corner monument to which the arm is pointing. The front door acts as an entry point, allowing to select the component, while the windows indicate that a component has been selected by changing their colour.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (select the desired functionality and then descend to street level) • Browsing / Navigation (navigation) • Details on demand / Drill-down (select the desired functionality and then descend to street level) • Clustering (clustering which brings together those components that share functionalities allowing the user to review all the components of a similar type in one area / functional groupings) • Overview + detail (overview / examine in detail) • Layout (block (grid) structure for layout (blocks as in a city and roads))
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	In a virtual reality environment (VRML enabled browsers or standalone viewers)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Any VRML Viewer or Web Browser with appropriate plug-in. Users can view the visualisation using standard VRML enabled browsers or stand alone viewers. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Software World was implemented using a desktop virtual reality system using C code. This C code was automatically generated from the Java source code in a two-step process. The source was parsed in order to populate a repository, and then from this the necessary code for the visualisation was generated.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 36. Test Driver + SpyApp + Transformer [Marshall200381]

	Field	Information to be extracted
Publication metadata	**Title**	Aspects to visualising reusable components
	Authors	Marshall, S., Jackson, K., Anslow, C., Biddle, R.
	Publication date (year/month)	February, 2003
	Publication type	Conference
	Source	Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2003)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Adelaide, Australia
	Pages	pp. 81-88
	Link (if applicable)	http://dl.acm.org/citation.cfm?id=857091
	Abstract	We are interested in helping developers reuse software by providing visualisations of reusable code components. These visualisations will help determine if and how a given code component can be reused in the developer's new context. To provide these visualisations, we need both formatted information and tools. We need a format to describe the visualisations in. We need tools to create the visualisations. We need a format to describe information about the component and its runtime usage, and we need a tool to gather this information in the first place. In this paper, we discuss our two wish-lists for the required information formats. We set this against the background of software visualisation and code reuse research. Currently we are working with components from object oriented languages, specifically Java.
Visualization metadata	**Approach/tool name** *(PQ)*	Test Driver + SpyApp + Transformer[34]
	Screenshot	N/A

[34] Test Driver and Transformer are also mentioned in VARE and Spider as modules of these approaches. According to [Marshall2001103], as opposed to Dyno and Fire, they are not described as a standalone application.

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	*Described separately at the end of this table (before Table 37), due to space issues.*
	Approach goals *(SQ1)*	Help developers to reuse software by providing visualisations of reusable code components, allowing to identify what information is important in deciding if and how a component can be reused, and develop tools to allow developers to create and view visualisations of this information.
	Visualizations' reuse-specific goals *(SQ1)*	Help to determine if and how a given code component can be reused in the developer's new context, guide a developer's decision as to whether a component is reusable in the developer's current context, and help foster understanding in the developers as to how they could save time and effort through the process of reusing old code in new contexts.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software development with reuse (reuse software / development) • Software maintenance (maintenance)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Integrating reusable assets (determine if and how a given code component can be reused in the developer's new context / whether a component is reusable in the developer's current context)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer with reuse (developers / reuser / code reusers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (reusable code components, a given fragment of old code (referred to as a component) / static and dynamic information present in a component)
	Source of visualized items/data *(TQ3.1)*	Events in the Java Virtual Machine and .class files.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Java debugger libraries collect information gathered from developers' experiences of using the component. The static information gathered by the Test Driver can be done so from .class files. Dynamic information is gathered purely from executing .class files as well. SpyApp watch for events in the Java Virtual Machine. When these events occur, SpyApp then collects event information through calls to the JDI, and sends the collected information to the filesystem or a database as output.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	N/A
	Data-to-visualization mapping (input/output) *(TQ4.1)*	N/A
	Visualization strategies and techniques *(TQ4.2)*	N/A
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	XML, Java Reflection API, Java Debugger Interface (JDI) and Java are used.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Test Driver + SpyApp + Transformer [Marshall200381] / Approach motivation/Assumptions *(SQ1)*: For all the benefits that reusing code is claimed to be able to deliver, it is perceived that code reuse is not as widespread or as efficiently implemented as it could be. Certainly, code reuse does happen on some levels. A common example of this is the increasing range and availability of libraries and APIs for the Java platform that offer rich opportunities for reuse. But even where code reuse is possible, often the rewards in time and effort saved are not as great as they could be due to problems in the process of reuse. There have been several areas of cost identified in the reuse process, where cost is measured in time, effort and financial terms. Despite the other costs of reuse do exist – notably the time to search for potential candidate components for reuse and the financial cost of purchasing reusable components –, there is the cost of understanding. Research into software visualisations for understanding program traces does exist, but much of this is not focused specifically on reuse and the information required in that process. Text based is the most common form of documentation currently available, and it should be complemented by visualisations, rather than be replaced entirely.

While code reusers are interested in what a component does, it will be of equal importance to understand how to use that component. A code reuser is also approaching the reusable component from the perspective of having it collaborate with other components that it was possibly not intended to be with originally. This means that the component's external influences are important to visualise as well. Code reusers are, like software visualisers, trying to understand whether a component matches its specification, interested in the side-effects and results of a component. To decide whether a potentially reusable component is useful in the new context, a developer must know what the component does (component as a black box). The results of executing certain sequences of method calls on a component's public interface, or the side-effects of these sequences on other components' data or the component's own state, will affect its applicability for reuse in a new context. If a component is to be reused, then any information sent or requested by that component to such entities as a network, filesystem or database, needs to be handled in the new context. It is important that the code reuser understands the requirements and actions of the component with regards to the external environment. This ensures that any components whose needs can not be met (either directly or through relatively minor modification) by the new context, can be discarded from the selection process. Should a component require interaction with a user to perform its functionality, then this needs to be understood by a code reuser if they are to make an informed decision as to its appropriateness in a new context.

There is also the question of how the component works. This is important as the resource or permission requirements of operation may be prohibitive in the new context, and rule the component out as a candidate for reuse. Understanding how the internals of the components work may open up opportunities for modifying its behaviour to what is required by replacing sub-components or overloading methods (component as a white box). Visualisations aimed at promoting understanding how a component works should incorporate feedback from the author and users. Some visual techniques could be applied to the descriptions, to aid readability and understanding, but people's reports on their experiences of components remains a powerful way of sharing knowledge. The system permissions required by a component affect its appropriateness for reuse in a given situation. Some environments may restrict permissions for security reasons. Describing what permissions a component requires allows the code reuser to make an informed decision regarding its usefulness. Should a component require other software to fulfill its functionality, then visualising this information will enable a code reuser to better understand whether that component is appropriate for reuse. Visualising the information may help to identify the specifics of what software

is required, why, and where. A code reuser can then investigate whether this other software is available and usable in the new context. The performance of a component may make it prohibitively expensive to reuse in a new context. Visualising this information gives the code reuser a better understanding of the appropriateness of a code component within the restrictions placed by the new context. Visualising what methods get called, on what classes, and when, may give the code reuser a better understanding as to what methods or classes need modifying to change the behaviour. This relates primarily to the execution hidden by the public interface of the component. By tracing the execution internal to a component, the code reuser can gain a better understanding of potential consequences and alternative executions that can be created by overloading or replacing certain parts of the component. Issues of threading, synchronisation, resource sharing and deadlock avoidance are important factors in deciding whether a component is reusable in a new context. A component's reliance on threads may make it unsuitable for a particular architecture. Threading and synchronisation data may comprise a large amount of information. Using visualisation techniques to highlight the important parts of this information can help the code reuser make a more informed decision about the reusable component's ability to work in collaboration with other components. Visualising the time line of execution can help a code reuser measure the component's performance against what is required, and against other potentially reusable components fulfilling the same functionality.

When a developer has decided that what a component does (or can be easily modified to do) is what they need, and that how it does it is acceptable to them, they will still need to understand how to reuse it. Examples showing previous uses of the public interface of a component can show us how to link in the component to other code in the new context. Reusing code may also involve extending the currently available functionality to match the new requirements. Another barrier is the time and effort involved in installing the component for use in the new project. Components may need more complex installation procedures. These could include recompilation for the local architecture, and downloading of other ancillary components that the reused component needs to work.

To create visualisations we need to think about what information should be visualised. We also need to consider the details of extracting, storing and transporting this information. Some software visualisation researchers have designed their visualisation architectures so that the visualisation tool is built directly into the information gathering tool. We wish to remove the tight coupling between the source and destination, by transferring the information in an independent, consistent format – so that can be replaced and reused in different circumstances. This will solve a common problem, where the captured information is abstracted too soon. Because a specific visualisation abstracts away information unnecessary in that visualisation, information relevant to other types of visualisation may be difficult to extract, or not even present. There is the need for an intermediary format for the information gathered from a component, that can be used to generate a visualisation. The visualisations, as well as the information about the component and its runtime usage, needs formats to be described in. There is also the need for tools to create the visualisations and to gather this information in the first place. At the time of exploration the software visualiser may not know what kinds of visualisations to view, or the need to view a different visualisation may become apparent at a point in the future. It should be possible to store trace output on a filesystem or database, so that it can be replayed in the future to produce a different visualisation. Often, understanding software through visualisation is an explorative process, where a software visualiser will tinker with a component and view the changes that the tinkering makes to the visualisation. Therefore, the changes detected may need to be transferred directly as they occur, rather than being sent at the close of execution. Supporting live streaming may impose certain restrictions upon the representation of the data. A file format should be able to be transferred in a culture neutral representation, and the data format should support an easy method of filtering, so that only relevant information need be passed. Any architecture involving the internet, and platform independent languages should transfer information in a platform-neutral manner. Execution traces can get large very quickly. The format chosen will need to be easily used, filtered and queried, even when it scales. The program trace information should be easily queried, so that a transformer can efficiently request subsets of information. In spite of the variety of programming languages, there is sufficient similarity between many object-oriented languages that it makes sense for the trace format to represent the execution of any of these languages.

Table 37. Spider [Anslow2004 / Marshall200435]

	Field	Information to be extracted
Publication metadata	**Title**	Software visualization tools for component reuse [Anslow2004] Using software visualisation to enhance online component markets [Marshall200435]
	Authors	Anslow, C., Marshall, S., Noble, J., Biddle, R. [Anslow2004] Marshall, S., Biddle, R., Noble, J. [Marshall200435]
	Publication date (year/month)	October, 2004 [Anslow2004] January, 2004 [Marshall200435]
	Publication type	Conference [Anslow2004] Conference [Marshall200435]
	Source	2nd Workshop on Method Engineering for Object-Oriented and Component-Based Development, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004) [Anslow2004] Proceedings of the Australasian Symposium on Information Visualisation (InVis.au 2004) [Marshall200435]
	Volume and Edition (for journals)	N/A [Anslow2004] N/A [Marshall200435]
	Place (for conferences)	Vancouver, Canada [Anslow2004] Christchurch, New Zealand [Marshall200435]
	Pages	?? [Anslow2004] pp. 35-41 [Marshall200435]
	Link (if applicable)	http://www.open.org.au/Conferences/oopsla2004/PapersME/4-Anslow.pdf [Anslow2004] http://dl.acm.org/citation.cfm?id=1082101.1082106 [Marshall200435]
	Abstract	This paper describes our experiences with our software visualization tools for web-based visualization of remotely executing object-oriented software. The motivation of this work is to allow developers to browse web-based software repositories to explore existing code components and frameworks by creating visual documentation. Components are test driven to capture their static and run-time information in program traces and are then transformed into useful visualizations. Visualizations can help developers understand what a component does, how it works, and whether or not it can be reused in a new program. [Anslow2004] Online component markets can be costly for consumers to use, in terms of the time and effort spent understanding the components on offer. This cost of understanding will deter consumers from reusing the available components. Software visualisations derived from the components' run-time behaviour can lessen the cost of understanding. We have developed a prototype tool called Spider for providing this functionality to producers and consumers. We discuss some of the issues involved, along with our experiences in implementing the prototype. [Marshall200435]

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Spider
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	*Described separately at the end of this table (before Table 38), due to space issues.*
	Approach goals *(SQ1)*	Browse web-based software repositories to explore existing reusable code components and frameworks by creating visual documentation [Anslow2004], and provide software visualizations of a component's behaviour [Marshall200435].
	Visualizations' reuse-specific goals *(SQ1)*	Help to understand what a component does, how it works, and whether or not it can be reused in a new program [Anslow2004], and complement other existing documentation, thus helping consumers to evaluate a candidate component (by giving them an insight into the existing behaviour as well as possible means of extending that behaviour) and helping producers advertise components [Marshall200435].
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (reuse of existing artifacts)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Integrating reusable assets (help consumers evaluate components)

	Field	Information to be extracted
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer with reuse (developers / consumers / the visualisation audience are the consumers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (Java components, component's behaviour)
	Source of visualized items/data *(TQ3.1)*	Information stored in a component (the content of .class and jar files).
	Collection procedure/method of visualized items/data *(TQ3.2)*	Spider interprets information stored in a component, detects events in the run-time environment, and interrogates the runtime environment's state. The information to be visualized is generated on the server-side. Debugger and XML technologies are used to capture and store interesting events. RCD documents are generated by analyzing the content of the associated .class and jar files. XTE documents are generated by listeners attached to a VM monitor. The listeners are responsible for extracting useful information from the events sent by the VM monitor. Test driving is performed through an HTML form. The test drive is specified as a sequence of constructor or method invocations on the component that is currently being investigated.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	N/A
	Data-to-visualization mapping (input/output) *(TQ4.1)*	N/A
	Visualization strategies and techniques *(TQ4.2)*	Filtering / Inclusion/Removal (filter out certain data, and focus on the important information / remove extraneous information that is not relevant to that goal)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (accessible through a standard web browser / W3C standards-compliant web browser)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Apache/Jakarta Tomcat. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Spider uses SGML/XML, HTML/CSS, Scalable Vector Graphics (SVG) and XML. Reusable Component Descriptions (RCD) are used for static information, and eXtensible Trace Executions (XTE) for dynamic information. Java, servlets and JSP pages, besides JSP tag libraries, are also used in Spider. The Java Reflection API and the Java Debugger Interface (JDI) are also used.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Spider [Anslow2004 / Marshall200435] / Approach motivation/Assumptions *(SQ1)*: In [Anslow2004], motivation is the same from VARE's motivation (described in Table 33).

Online markets bring together consumers and producers of reusable components. Consumers benefit by having a central repository that they can refer to when they need to find functionality to implement a requirement in a new system. Producers benefit by having a centralised audience for their sales pitch, whether the intention is to create revenue by charging a license fee, or to solely build a user-base. Markets can categorise components by field,

language, and function. Markets also provide search engines that allow a consumer to narrow down a set of candidate components through supplying some general keywords. Markets can also publish newsletters that highlight new and popular components as they arrive.

Component markets attempt to reduce the costs for producers and consumers, but the approaches used to achieve this still have some limitations, and the costs can still be sufficiently high to deter consumers. Such markets can be costly for consumers to use, in terms of the time and effort spent understanding the components on offer. This cost of understanding will deter consumers from reusing the available components. Consumers must search for components applicable to their situation in a potentially huge search-space. This can be costly in time alone, irrespective of whether services to support this are charged for. Once consumers have identified a small set of components worth further study with respect to applicability, they must evaluate each component. This evaluation involves gaining a better understanding of each component's behaviour, and potential for extension or modification. Thus, consumers must have access to material that can help them understand a component's behaviour. This documentation can include a producer's text-based descriptions of form and function, as well as reviews written by market reviewers, and by fellow consumers who have previously evaluated or used that component. This requires time and effort be spent in program comprehension. There is also a cost associated with trusting a component, and determining not only whether the candidate component matches the required functionality, but also whether it poses a security risk in a trusted environment. Producers' cost includes analysis and design of problem domains to extract common functionality, as well as the points at which this common functionality should be extendible. This cost then includes implementing and testing the code that performs this functionality. Finally, this cost also includes advertising the component to a consumer base, through the creation of material to be used by consumers in making an adoption decision; as well as also hosting the component for consumers to find. The cost of advertising to a consumer base is partly alleviated by having a central point to which consumers come. The published newsletters also benefit producers, especially those with an established reputation for excellence.

Component markets limitations in support are especially with regards to consumers understanding components' behaviour. Software visualisations derived from the components' run-time behaviour can lessen the cost of understanding. Consumers can use software visualisations to comprehend more information than they would normally be able to do in a strictly textual format. Producers can create test-driven software visualisations to advertise the features of their components. These software visualisations would then complement other text-based documentation currently supplied. A key aspect to a successful visualisation is that useful information is not obscured by unnecessary data. The producer is in a position to know which sequence of actions (i.e. calls to public methods) results in a particular task being performed, and can tailor the visualisation to show this specific goal being achieved. It is relevant to consider the intended audience for these visualisations, and what they intend to learn from viewing them. A component's runtime behaviour may consist of a massive number of events. If every event is stored in the visualisation, then the resources required to store, process, view, and comprehend the visualisation would be prohibitive. The consumers are viewing the visualisations to understand how the component works, and are not in an informed position to significantly help with filtering out unnecessary data. It is not particularly useful to capture all events that occur within another component (such as a standard library) even though it is due to the candidate component's behaviour. It would be sufficient to identify that the other component has been invoked, along with the details of the service requested. When test driving a component, the server could note the version of Java the component was implemented for. User interface components require a different means of control, as well as components that use an event-listener model, which need special configuration before they can be test driven. There may be a few scenarios where the component will not perfectly fit into the new context, as the server environment may subtly differ from the consumer's environment. It is unavoidable that the consumer will need to test the final selected component in their own environment. [Marshall200435]

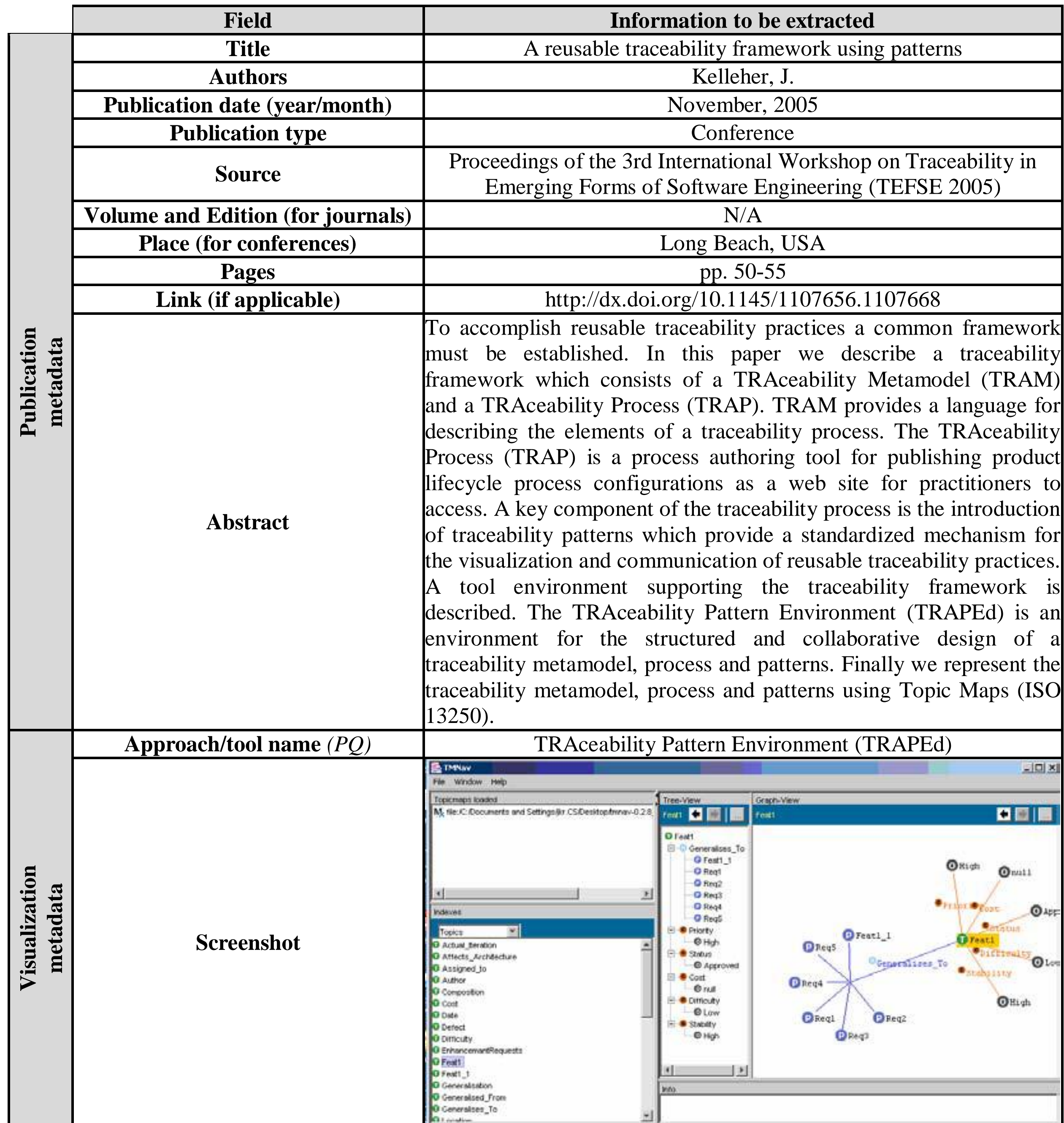
Table 38. Claims Exploration of Relationships Visualization (CERVi) [Wahid2004414]

	Field	Information to be extracted
Publication metadata	**Title**	Visualization of design knowledge component relationships to facilitate reuse
	Authors	Wahid, S., Smith, J. L., Berry, B., Chewar, C. M., McCrickard, D. S.
	Publication date (year/month)	November, 2004
	Publication type	Conference
	Source	Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IRI 2004)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Las Vegas, USA
	Pages	pp. 414-419
	Link (if applicable)	http://dx.doi.org/10.1109/IRI.2004.1431496
	Abstract	Within the software development process, reuse at the requirements level has become an increasingly more compelling notion. Following a human-centric approach, this work focuses on both requirements and design solution reuse using a design knowledge repository. In recent years, many improvements have been made to increase reuse through design knowledge repositories, but retrieval of knowledge in the context of design activities continues to be a formidable challenge. We propose a new system, called CERVi, to browse a repository through visualization by exploiting relationships between units of knowledge (in our case, claims). These relationships are key to finding the most appropriate reusable knowledge based on design conditions. Testing shows that CERVi enhances the design knowledge selection process and helps users proceed through structured design decision making. Our approach will be most useful to those interested in unlocking the potential of design knowledge reuse.
Visualization metadata	**Approach/tool name** *(PQ)*	Claims Exploration of Relationships Visualization (CERVi)
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Component selection is a very important characteristic of reuse. The ability to locate, compare, and select stored components is vital to the success of a software reuse repository. However, most design knowledge repositories do not support an outlined search strategy, a series of steps they can follow depending on their needs, to ensure that they will find all of the components they need. As with most knowledge management systems, acquisition is the bottleneck. In order to facilitate their reuse, Design knowledge claims must be generalized, classified, stored in a design knowledge repository, and retrieved as appropriate for use within a new design context. By studying relationships types among knowledge components, users can begin to follow links to find more components.
	Approach goals *(SQ1)*	Browse a repository through visualization by exploiting relationships between units of knowledge (in this case, claims).
	Visualizations' reuse-specific goals *(SQ1)*	Find the most appropriate reusable knowledge based on design conditions.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software design (design) • Analysis / Specification / Requirements engineering (requirements)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Searching and retrieving reusable assets (find the most appropriate reusable knowledge based on design conditions)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Software architect/designer (designers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Architecture / Design artifacts and related information (reusable claims / claim relationships)
	Source of visualized items/data *(TQ3.1)*	A design knowledge repository.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Designers can first use components discovered through a traditional search query to find more by browsing and following relationship links based on a certain design need.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (networked structure of relationships among claims)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Colors are used to represent relationship types between claims.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (see claim details by clicking on a claim) • Browsing / Navigation (navigating) • Details on demand / Drill-down (see claim details by clicking on a claim) • Filtering (filter based on relationships / showing claims related to the center claim)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [by others] (user testing, followed by a survey)
	Application scenarios of the visualizations *(TQ7.1)*	Performed by sixteen undergraduate HCI students. [academic]
	Evaluated aspects *(TQ7.2)*	The impact of CERVi on finding reusable knowledge for interface design, to verify that the defined relationships can be incorporated into a strategy for locating claims, and to validate the tool as a selection mechanism to facilitate claims reuse. Participants' perceived understanding of the relationships and their use, the incorporation of those relationships into participants' search strategies, and the impact of the relationships on the resulting design work.
	Visualization evaluation results/outcomes *(TQ 7.3)*	Descriptions of the envisioned systems were consistent and generally appropriate for the given scenario. Search strategies were also somewhat consistent. Approximately one third of the participants produced effective designs; designs created using CERVi were considerably better than those created using the traditional library search mechanism. Most participants located the majority of their claims using CERVi, and could easily retrieve a claim based on a displayed relationship. Users who primarily used CERVi to search for claims committed fewer errors and received higher design scores. Participants gained a basic understanding of the four high level relationships while that understanding decreased significantly for the lower level relationships. There was uncertainty in using the relationships to understand the underlying purpose of a related claim, but they indicated that examination of claim details was a key aspect of the selection process. Design scores for a second group of six expert users (that were asked to perform the same tasks) rose considerably.

Table 39. TRAceability Pattern Environment (TRAPEd) [Kelleher200550]



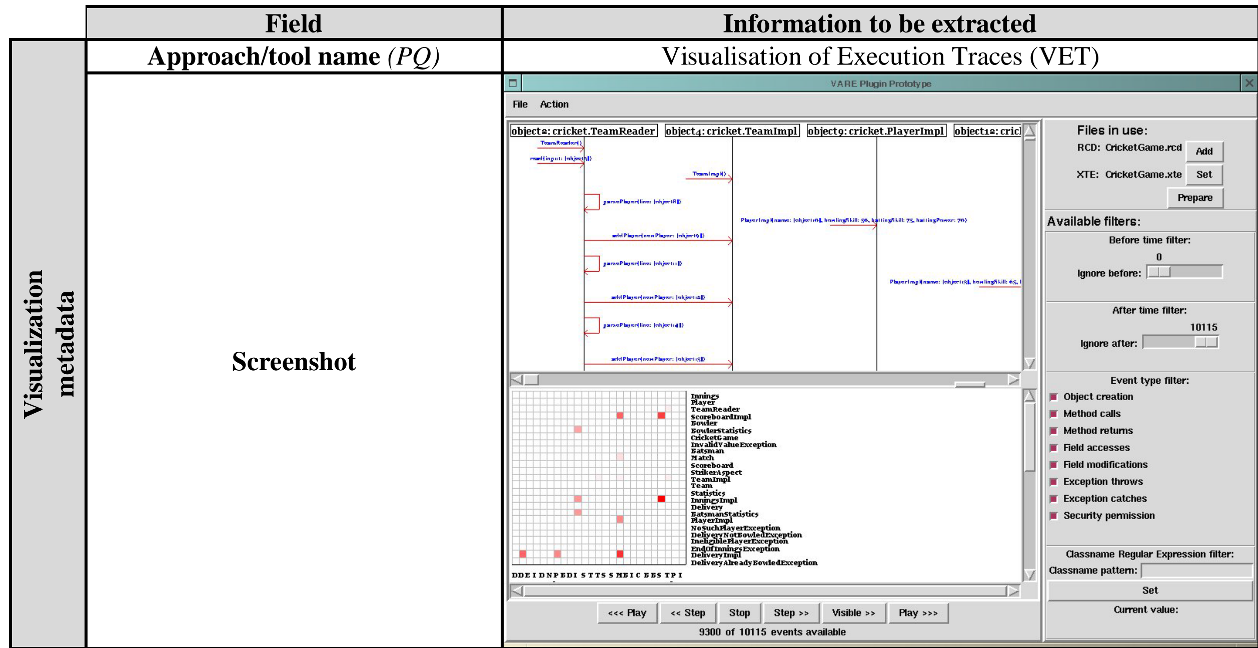
	Field	Information to be extracted
Publication metadata	**Title**	A reusable traceability framework using patterns
	Authors	Kelleher, J.
	Publication date (year/month)	November, 2005
	Publication type	Conference
	Source	Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2005)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Long Beach, USA
	Pages	pp. 50-55
	Link (if applicable)	http://dx.doi.org/10.1145/1107656.1107668
	Abstract	To accomplish reusable traceability practices a common framework must be established. In this paper we describe a traceability framework which consists of a TRAceability Metamodel (TRAM) and a TRAceability Process (TRAP). TRAM provides a language for describing the elements of a traceability process. The TRAceability Process (TRAP) is a process authoring tool for publishing product lifecycle process configurations as a web site for practitioners to access. A key component of the traceability process is the introduction of traceability patterns which provide a standardized mechanism for the visualization and communication of reusable traceability practices. A tool environment supporting the traceability framework is described. The TRAceability Pattern Environment (TRAPEd) is an environment for the structured and collaborative design of a traceability metamodel, process and patterns. Finally we represent the traceability metamodel, process and patterns using Topic Maps (ISO 13250).
Visualization metadata	**Approach/tool name** *(PQ)*	TRAceability Pattern Environment (TRAPEd)
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Traceability patterns describe best practices, good traceability designs, and captures successful work experiences. In spite of that, little research focused on the reusability of engineering activities or the recognition of commonalities in practices within the traceability domain.
	Approach goals *(SQ1)*	Provide a standardized mechanism for the visualization and communication of reusable traceability practices.
	Visualizations' reuse-specific goals *(SQ1)*	Visualize and communicate reusable traceability practices.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Analysis / Specification / Requirements engineering (requirements management)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Integrating reusable assets (visualize and communicate reusable traceability practices)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Analyst (requirement engineer)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Requirements/Analysis artifacts and related information (information related to requirements and traceability / traceability items, their attributes, and their relationships with other requirements / requirement attributes, such as status and priority / relationships between requirements)
	Source of visualized items/data *(TQ3.1)*	Information from different sources (customer requirements versus project requirements).
	Collection procedure/method of visualized items/data *(TQ3.2)*	The traceability items can be stored in many different locations. Information is retrieved on the progress of a project with regard to priorities, workloads, and deadlines, the addition of new requirements, and changing or unstable requirements. There is a round-trip engineering from UML to XMI to CSV to XTM and vice versa. The traceability metamodel and traceability patterns are represented in UML. UML is converted to XMI, and XMI is transformed into CSV. The CSV format is imported into a requirement management tool (Rational RequisitePro) and the resulting traceability matrix is reviewed. The XMI is finally converted to XTM, visualizing the results topic maps.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Map (topic maps)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Traceability items, their attributes, and their relationships with other requirements are displayed in nodes. Each requirement or any traceability item can be represented as a topic. The relationship between topics or traceability can be represented by an association.
	Visualization strategies and techniques *(TQ4.2)*	• Browsing / Querying (query functions) • Clustering (topic maps can be merged into a single topic map and integrated into a meaningful whole) • Filtering (filtering) • Sorting (sorting)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	• Mouse (assumed) • Keyboard (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The following technologies and frameworks are used: Software Process Engineering Metamodel (SPEM) specification, Meta-Object Facility (MOF), Unified Modeling Language (UML), XML Metadata Interchange (XMI), XML Topic Maps (XTM) 1.1, TMQL (Topic Map Query Language) and the TM4J (an open-source framework for developing topic map processing applications) is used. The standard topic map format used is XTM.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 40. Visualisation of Execution Traces (VET) [McGavin2006153]

	Field	Information to be extracted
Publication metadata	**Title**	Visualisations of execution traces (VET): an interactive plugin-based visualisation tool
	Authors	McGavin, M., Wright, T., Marshall, S.
	Publication date (year/month)	January, 2006
	Publication type	Conference
	Source	Proceedings of the 7th Australasian User Interface Conference (AUIC 2006)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Hobart, Australia
	Pages	pp. 153-160
	Link (if applicable)	http://dl.acm.org/citation.cfm?id=1151780
	Abstract	An execution trace contains a description of everything that happened during an execution of a program. Execution traces are useful, because they can help software engineers understand code, resulting in a variety of applications such as debugging software, or more effective software reuse. Unfortunately, execution traces are also complex, typically containing hundreds of thousands of events for medium size computer programs, and more for large scale programs. We have developed an execution trace visualisation tool, called VET, that helps programmers manage the complexity of execution traces. VET is also plugin based. Expert users of VET can add new visualisations and new filters, without changing VET's main code base.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	Visualisation of Execution Traces (VET)
	Screenshot	

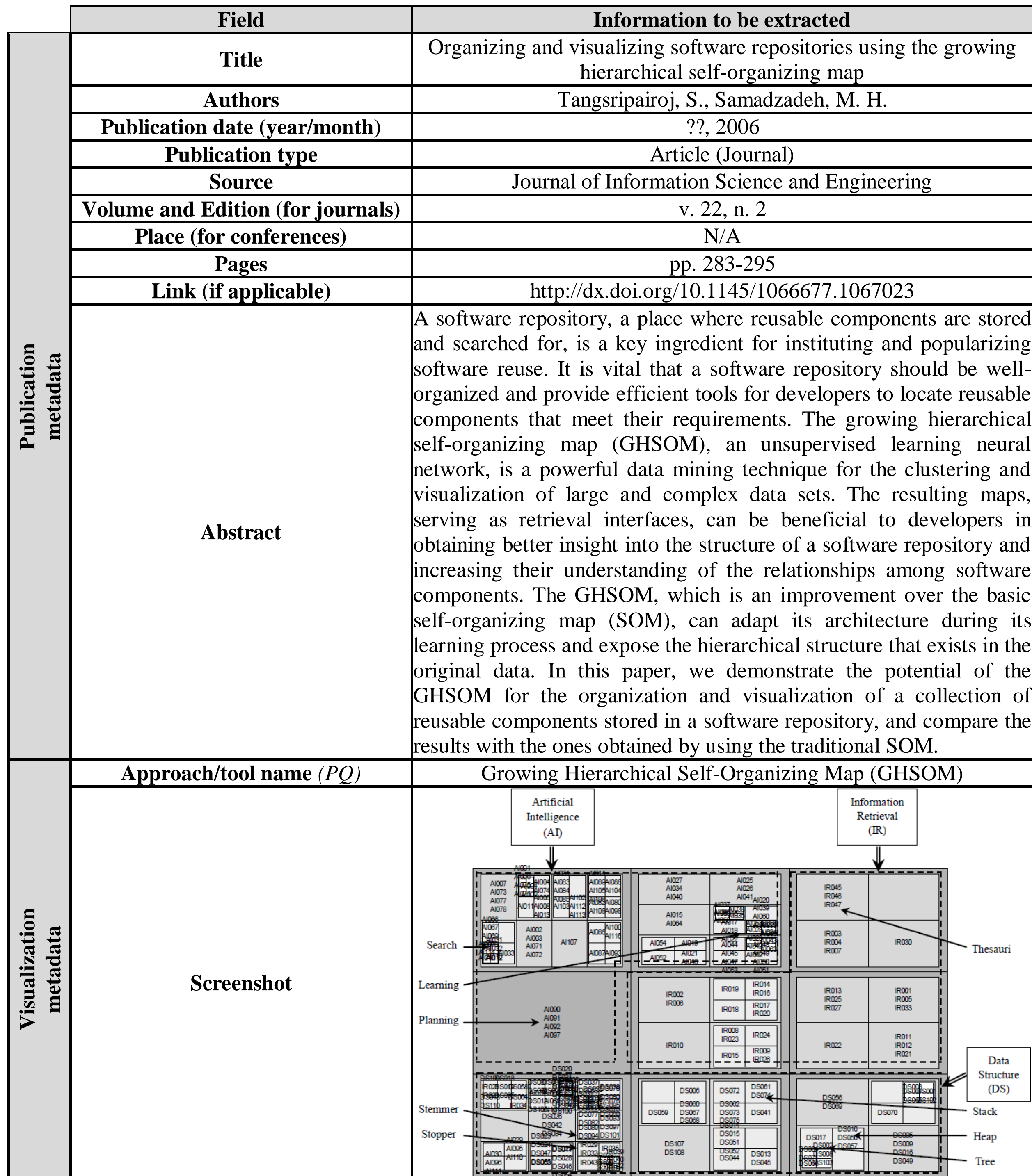
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Code can be difficult to read either because of developers' differing writing styles, or because the code isn't available for legal reasons, or because control flow jumps from one block of code to some other distant block of code, or because (in the case of languages that support polymorphism) it can be difficult to say exactly what one statement will do until the software is actually executing. The traditional approach for understanding code is to read source code or documentation written by the software's author. This approach has several problems: source code is static, complex, and hard to understand; and documentation might be out of date or incomplete. Associated documentation may or may not exist, and may not cover the particular facet of the behaviour that the developer is interested in. Execution traces are useful, because they can help software engineers understand code, resulting in a variety of applications such as debugging software, or more effective software reuse. Unfortunately, execution traces are also complex, typically containing hundreds of thousands of events for medium size computer programs, and more for large scale programs. Moreover, a developer may only be interested in one particular facet of an execution. Although debuggers can be useful, they often require the developer to know where to insert interesting breakpoints prior to understand the software. Capturing, logging, and visualising run-time information requires significant tool support to automatically extract and present useful information. While sequence diagrams are useful for understanding what has happened when software executes, they quickly become difficult to view as the execution traces become large. A solution to the information overload is to let developers configure their tools. As developers typically use these tools because they do not understand the software, it can be difficult for the developer to identify what is useful information before actually seeing a visualisation of the information. Moreover, different developers have different information requirements – affecting what a tool needs to display as well as how that information should be displayed. With a plugin-based architecture, users can build their own filters and visualisations.
	Approach goals *(SQ1)*	Help programmers to manage the complexity of execution traces, by visualising the dynamic execution of software and letting users interact with and understand execution traces.
	Visualizations' reuse-specific goals *(SQ1)*	Help to understand code behaviour, resulting in more effective software reuse.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Programming / Coding (programming) • Quality assurance / Testing / Debugging / Profiling (debugging / profiling[35])
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' behavior (understand code behaviour)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (software engineers / programmers / developers)

[35] Profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls. The most common use of profiling information is to aid program optimization.

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (method calls between objects / how objects interact)
	Source of visualized items/data *(TQ3.1)*	Information stored in an execution trace.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The execution trace format uses XML to store execution traces in an architecture- and language-independent manner. XTE documents information for events and also stores the values and references for any objects that are created during the execution of the software. Information stored in an execution trace is retrieved from other parts of the VARE architecture. VET parses an execution trace and converts it to an event-driven interface for visualisation and filter plugins. It makes the events of an execution trace available through an abstract API. An expert user of VET can define their own visualizations that draw pictures using a provided API, and their own filters. At appropriate times, VET will send messages to the user-designed visualisation components, instructing them to update their displays. It also allows for non-expert users to alter parameters of these filters before, during and after the processing of an execution trace. For every event, each active filter is queried to determine if the event should be displayed by visualisations. After checking the event, VET passes the event to every visualisation plugin. A plugin architecture is used in the process.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Diagrams (sequence diagrams) • Matrix / Matrix-like (class association diagram (otherwise known as a call graph) / scatter-graph) • Others (users can build their own filters and visualisations)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	All the objects are listed on both the X and Y axis. The point (x, y) is colour coded to represent how often object x has invoked a method on object y. There is a horizontal list of objects, and a vertical time line. Method calls are represented on the timeline as an arrow from the object that makes the method call to the object that receives the method call. The darkness of the shade in each part of the grid indicates the amount of messages that have been passed from the class on the x axis to the class on the y axis. In other words, the frequency of method call events from any class to another class is represented by the darkness in the shade of each square relative to other squares in the grid. The green arrows on the diagram indicate alternative flows that take place if the user adjusts the settings of one of the active filters.
	Visualization strategies and techniques *(TQ4.2)*	• Browsing / Navigation (real-time dynamic queries) • Details on demand / Drill-down (easily get detailed information about any particular data point on demand / drill down / “details on demand”) • Filtering / Tuning/Tweaking (visualisations are updated in real time as filtering criteria are adjusted) • Filtering / Inclusion/Removal (remove information they are not interested in / let the user filter out unwanted information) • Overview + detail (show the user everything at once when they start the program / get an overview of all information) • Zooming / Semantic (text in the latter diagram is automatically shown as rectangles when it is reduced below a size beyond human readability) • Presentation / Simultaneous (display multiple visualisations in parallel) • Overlap / Flipping (keeps the two visualisations synchronised at the same location)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (built and tested in both NetBSD and Linux Operating System environments)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	VET is built using the Python scripting language, and the Tk graphical toolkit. It uses XTE, Reusable Component Descriptions (RCD), and XML DOM objects.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 41. Growing Hierarchical Self-Organizing Map (GHSOM) [Tangsripairoj2006283]

	Field	Information to be extracted
Publication metadata	**Title**	Organizing and visualizing software repositories using the growing hierarchical self-organizing map
	Authors	Tangsripairoj, S., Samadzadeh, M. H.
	Publication date (year/month)	??, 2006
	Publication type	Article (Journal)
	Source	Journal of Information Science and Engineering
	Volume and Edition (for journals)	v. 22, n. 2
	Place (for conferences)	N/A
	Pages	pp. 283-295
	Link (if applicable)	http://dx.doi.org/10.1145/1066677.1067023
	Abstract	A software repository, a place where reusable components are stored and searched for, is a key ingredient for instituting and popularizing software reuse. It is vital that a software repository should be well-organized and provide efficient tools for developers to locate reusable components that meet their requirements. The growing hierarchical self-organizing map (GHSOM), an unsupervised learning neural network, is a powerful data mining technique for the clustering and visualization of large and complex data sets. The resulting maps, serving as retrieval interfaces, can be beneficial to developers in obtaining better insight into the structure of a software repository and increasing their understanding of the relationships among software components. The GHSOM, which is an improvement over the basic self-organizing map (SOM), can adapt its architecture during its learning process and expose the hierarchical structure that exists in the original data. In this paper, we demonstrate the potential of the GHSOM for the organization and visualization of a collection of reusable components stored in a software repository, and compare the results with the ones obtained by using the traditional SOM.
Visualization metadata	**Approach/tool name** *(PQ)*	Growing Hierarchical Self-Organizing Map (GHSOM)
	Screenshot	

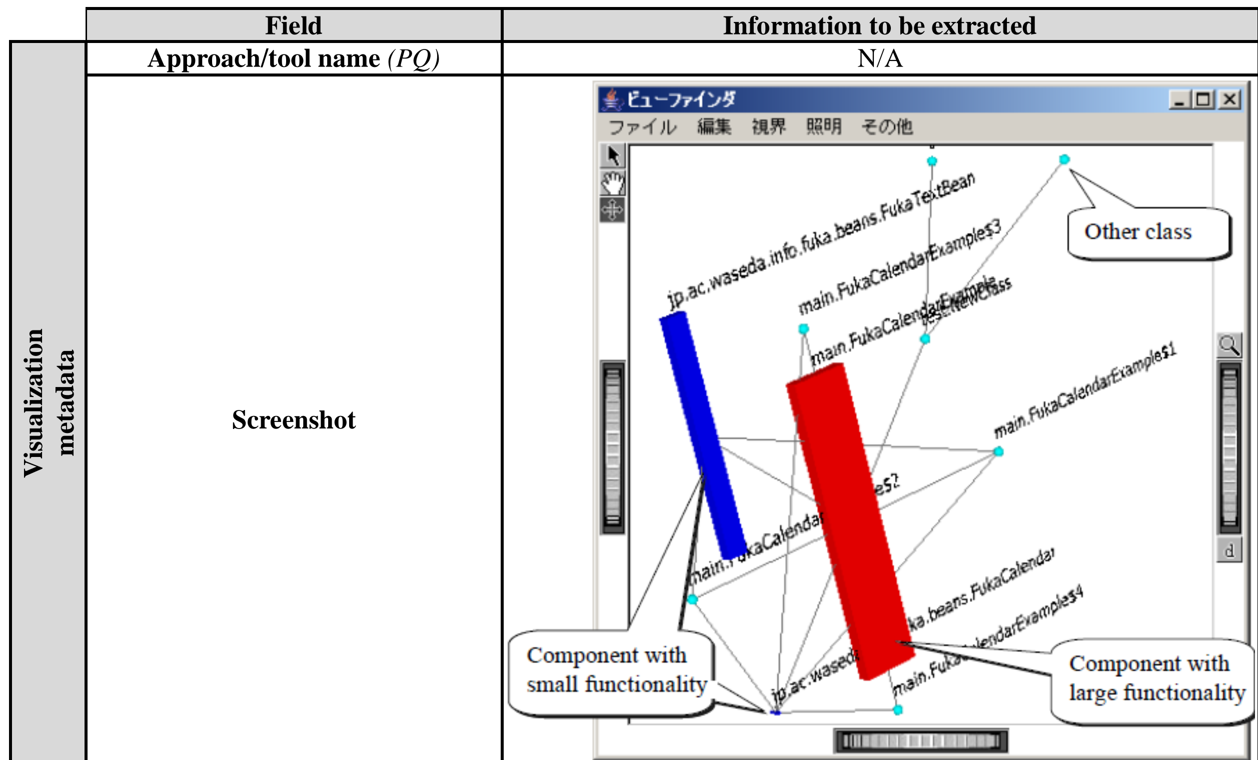
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	A software repository, a place where reusable components are stored and searched for, is a key ingredient for instituting and popularizing software reuse. It is vital that a software repository should be well-organized and provide efficient tools for developers to locate reusable components that meet their requirements. There should be tools for developers to find the desired reusable components quickly and easily, and hence to make better decisions in selecting the right components for reuse. In a software repository that is possibly large and ever-growing, the process of specifying, locating, and retrieving reusable components can be complex and time consuming, and hence frustrating if the software repository is not well-organized. It is crucial that the software repository should be well-structured such that the reusable components closest to the developers' needs are easy to discover. The use of the traditional Self-Organizing Map (SOM) may not be practical when the number of software components stored in a software repository is large. Therefore, applying dynamic SOM models to software repository organization seems to be a more promising alternative.
	Approach goals *(SQ1)*	Organize and visualize a collection of reusable components stored in a software repository.
	Visualizations' reuse-specific goals *(SQ1)*	Obtain better insight into the structure of a software repository and increase understanding of the relationships among software components.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (reuse-based software development)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' structure / asset information / repository (obtain a better insight into the structure of a software repository and increase understanding of the relationships among software components)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (developers)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (reusable components – a reusable component can be any software document or work product generated during the software development process, examples include requirement analysis documents, architectural designs, code modules, test plans, test cases, and documentation)
	Source of visualized items/data *(TQ3.1)*	Stored in a software repository.
	Collection procedure/method of visualized items/data *(TQ3.2)*	SOM takes a set of inputs and maps them onto the neurons of a two-dimensional grid. The weight vectors are randomly initialized at the first stage. Then, the SOM network performs learning in two main steps, and determines the winning neuron for a given input vector, selected randomly from the set of all input vectors. For every neuron on the grid, its weight vector is compared with the input vector by using some similarity measures, e.g., Euclidean distance. The neuron whose weight vector is closest to the input vector is selected to be the winning neuron. After a winning neuron is determined, the weight vectors of the winning neuron and all of its neighboring neurons are adjusted by moving toward the input vectors according to the learning rule. This learning process progresses repeatedly until it converges to a stable state where there are no further changes made to the weight vectors when they are presented with the given input vectors. After the training has been completed, an orderly map is formed in such a way that the topology of the data is preserved and becomes geographically explicit in that similar input data are mapped onto nearby regions of the map.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (tree structure where the maps at each layer can branch out to additional maps at the subsequent layer) • Map (map)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Similar input data are mapped onto nearby regions of the map. The size of these Self-Organizing Maps and the depth of the hierarchy are determined during its learning process according to the requirements of the input data.
	Visualization strategies and techniques *(TQ4.2)*	• Clustering (mapping of high-dimensional input space onto a low-dimensional (usually two-dimensional) map, where similar input data can be found on nearby regions of the map / clustering / the clusters are the areas with high data densities on the map that are further hierarchically expanded by growing SOMs) • Overview + detail (the upper layers show a coarse organization of the major clusters in the data, whereas the lower layers offer a more detailed view of the data) • Hierarchical visualization (a hierarchy of multiple layers where each layer consists of several independent growing SOMs)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	A SOMLib4 Java package is used to extract keywords and create the feature vectors, while MATLAB SOM Toolbox5 and GHSOM Toolbox6 are used for the construction of the SOM and the GHSOM.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (experiment) [similar to a benchmark analysis]
	Application scenarios of the visualizations *(TQ7.1)*	273 samples of C/C++ program source code files (gathered from textbooks widely used in computer science classes) were used. [academic]
	Evaluated aspects *(TQ7.2)*	Results of the application of the SOM against the GHSOM.
	Visualization evaluation results/outcomes *(TQ 7.3)*	Both SOM and GHSOM were successful in creating a topology-preserving representation of the topical clusters of the software components. However, when dealing with a large number of software components, GHSOM behaved better than SOM: GHSOM was able to reveal the inherent hierarchical structure of the data into layers and provided the ability to select the granularity of the representation at different levels of the GHSOM.

Table 42. Washizaki et al.'s approach [Washizaki20061222]

	Field	Information to be extracted
Publication metadata	**Title**	A system for visualizing binary component-based program structure with component functional size
	Authors	Washizaki, H., Takano, S., Fukazawa Y.
	Publication date (year/month)	??, 2006
	Publication type	Article (Journal)
	Source	WSEAS Transactions on Information Science and Applications
	Volume and Edition (for journals)	v. 3, n. 7
	Place (for conferences)	N/A
	Pages	pp. 1222-1230
	Link (if applicable)	N/A
	Abstract	Component-based software development is a development approach which aims to reduce development costs and increase software reliability. With component-based development, often new program is created quickly by reusing components in binary form that have been developed by third parties, without access to the source code of those components. In order to maintain such program on an on-going basis, it is important to be able to visualize the overall structure and behavior of the program. However, because existing program visualization systems need to analyze the program source code, it has been difficult to apply them to program that incorporates components in binary form. In this paper, we propose a program visualization system which does not make use of the source code, but uses two techniques, reflection and byte-code analysis, to measure the functional size of each component and to determine the dependency relationships among components and helper classes. These results are used to provide an accurate visualization of the overall structure of the component-based program. Our system can be applied to programs built with JavaBeans components. As a result of comparative evaluations, it is found that our system is useful for visualizing binary component-based program structure with component functional size to support maintenance activities.

	Field	Information to be extracted
Visualization metadata	Approach/tool name *(PQ)*	N/A
	Screenshot	

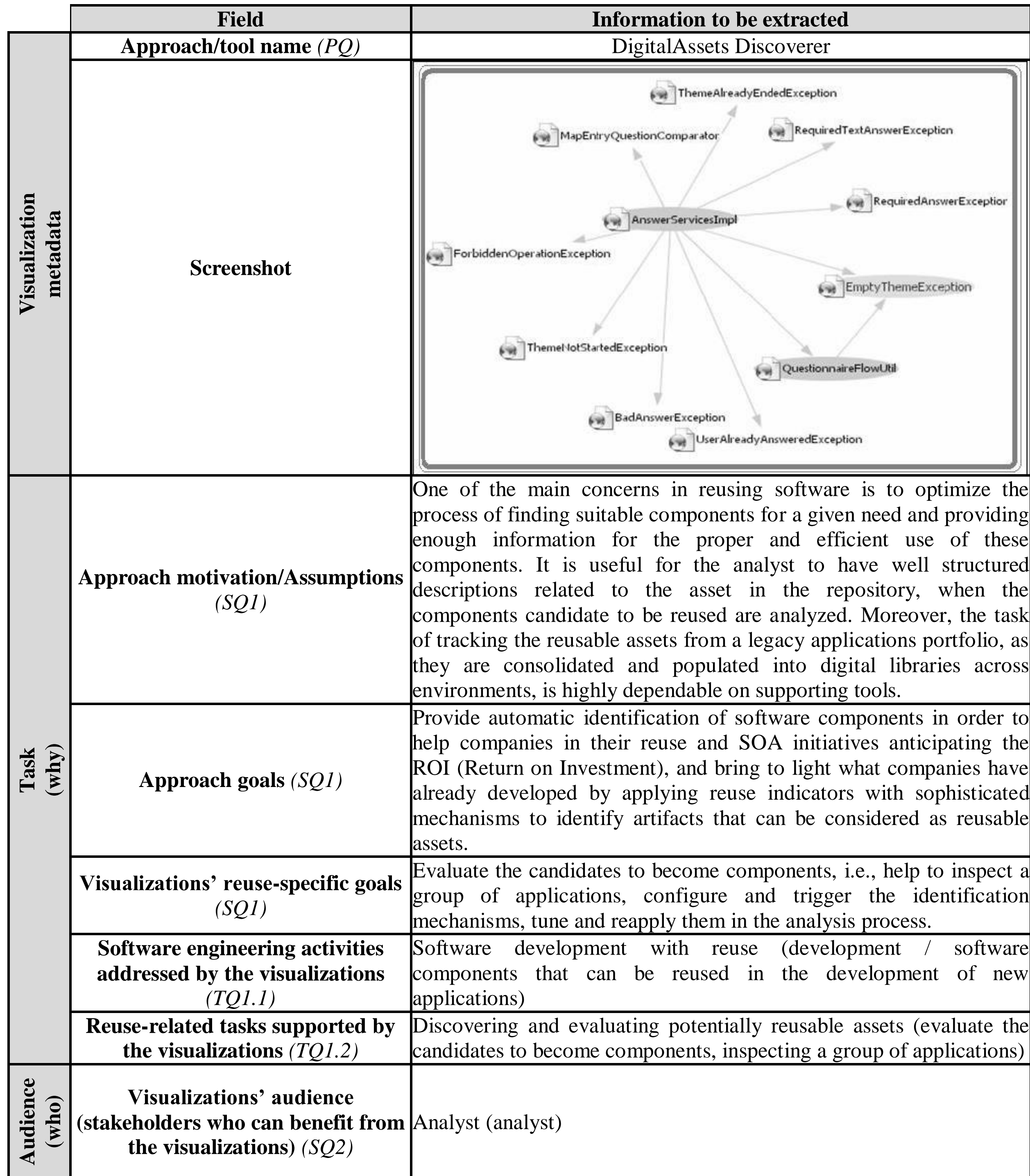
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Often in component-based development, components that have been developed by a third party and delivered in binary format (without access to the source code) are reused to build new software quickly. It is well known that much of the time spent maintaining software is consumed in simply understanding the software. In order to effectively maintain software that has been obtained through component-based development on an on-going basis, it is necessary to provide the maintainer with an intuitive understanding of the software as a collection of components. However, visualizing a program that was created by incorporating binary components is very difficult. For binary components whose internals are hidden, as the functional size increases, the component's applicability for reuse increases, but the effort required understanding the functionality also increases, so it may also indicate additional problems in terms of maintenance. Conventional visualization systems do not distinguish between classes and components that make up the program. They do not allow the user to visually differentiate them, and provide no support for an intuitive understanding of the internal structure. Further, analysis of the source code is a prerequisite for using these existing systems, so they cannot handle parts of binary component based program for which the source code is not available.
	Approach goals *(SQ1)*	Support maintenance activities, i.e., perform maintenance tasks such as fixing bugs or adding extensions efficiently.
	Visualizations' reuse-specific goals *(SQ1)*	Help programmers gain the understanding of the binary component-based program and give an intuitive understanding of the overall functional size, as well as whether the break-down and allocation of functionality within the program is appropriate.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software maintenance (maintenance)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' structure / asset information / repository (help to gain the understanding of the binary component-based program and give an intuitive understanding of the overall functional size)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (programmers)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (the static structure of component-based program, binary components, the static structure of a program made up of helper classes and components provided in byte-code format / dependency relationships between components (and helper classes), functional size of the components (which gives an indication of the amount of functionality provided by the component), the number of methods, properties and events made public by a component)
	Source of visualized items/data *(TQ3.1)*	Components and Java byte-code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Reflection and byte-code analysis are used for obtaining the dependency relationships, i.e., by using the Java reflection and JavaBeans introspection functions and by analyzing the byte-code of the components. The functional size measurement values are obtained using a component-size metric. Component analysis determines whether a Java class satisfies the JavaBeans specifications as described above, and if it is a Bean, its functional size is determined using the Bean reflection mechanism. Dependency analysis, in its turn, is done by analyzing the data in the constant pool within the Java byte-code, and obtaining the dependency relationships between classes. To compute a reference value for the functional size of a component (FOC) metric, evaluation data from the contributed components made available on JARS.COM are used. On JARS.COM, a large number of Beans in various categories such as Programming and Utilities are judged (by development experts) and given an 8-level evaluation with respect to expressiveness, functionality and originality. This 8-level evaluation is normalized to fit into the range [0, 1] (1 being best), and is called the JARS evaluation.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Geometric forms (box, spheres, straight lines)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Dependency relationships and functional size information are arranged within a 3-D space. Components are visualized as boxes which reflect the value of the functional size of a component (FOC). The number of methods, events and properties are related to the width, depth and height of the box, respectively, so that the volume of the box is a visual representation of the FOC, the component functional size. If the FOC of a component is less than the FOC reference value, the box is displayed in blue, and if it exceeds the standard value, the box is displayed in red. Java classes which are not components are displayed as light-green spheres. Dependency relationships are displayed as straight lines joining the visualization objects of the classes or components in the relationship.
	Visualization strategies and techniques *(TQ4.2)*	• Layout / 3D (3-D coordinate space / the 3-D coordinate data obtained using the visual mapping is output to the 2-D image) • Zooming / Geometric (zoom-in/out) • Rotating (rotate)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Jun for Java. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, it uses the introspection mechanism and reflection mechanism provided by the language. It also uses the Javassist byte-code analysis tool and Jun for Java, a 3-D graphics/multi-media framework.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 43. DigitalAssets Discoverer [Gonçalves2007872 / Oliveira2007461]

	Field	Information to be extracted
Publication metadata	**Title**	DigitalAssets discoverer: Automatic identification of reusable software components [Gonçalves2007872] Automatic Identification of reusable Software development assets: Methodology and tool [Oliveira2007461]
	Authors	Gonçalves, E. M., Oliveira, M. D. S., Bacili, K. R. [Gonçalves2007872] Oliveira, M., Gonçalves, E. M., Bacili, K. R. [Oliveira2007461]
	Publication date (year/month)	October, 2007 [Gonçalves2007872] August, 2007 [Oliveira2007461]
	Publication type	Conference [Gonçalves2007872] Conference [Oliveira2007461]
	Source	Proceedings of the 22nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007) [Gonçalves2007872] Proceedings of the 2007 IEEE International Conference on Information Reuse and Integration (IRI 2007) [Oliveira2007461]
	Volume and Edition (for journals)	N/A [Gonçalves2007872] N/A [Oliveira2007461]
	Place (for conferences)	Montreal, Canada [Gonçalves2007872] Las Vegas, USA [Oliveira2007461]
	Pages	pp. 872-873 [Gonçalves2007872] pp. 461-466 [Oliveira2007461]
	Link (if applicable)	http://dx.doi.org/10.1145/1297846.1297932 [Gonçalves2007872] http://dx.doi.org/10.1109/IRI.2007.4296663 [Oliveira2007461]
	Abstract	DigitalAssets Discoverer is a tool that implements a group of indicators for automatic identification of software components that can be reused in the development of new applications and Web Services. This tool brings into light the J2EE applications portfolio developed in-house, increasing productivity and anticipating the ROI in companies. The process of components harvesting and analysis uses an interactive user graphical interface that enables the tuning of selected indicators, visualization of the results and publishing the identified components into a reusable software development assets repository. [Gonçalves2007872] Software reuse is seen as one of the main alternatives to increase productivity in the development of new applications. The reuse of legacy assets plays a vital role anticipating the ROI (Return on Investment) on SOA (Service Oriented Architecture) and reuse enterprise programs. This paper presents a tool that implements an Automatic Identification of Software Components (AISC). AISC is an approach that brings to light what companies have already developed by applying reuse indicators with sophisticated mechanisms to identify artifacts that can be considered as reusable assets. Thus, they will have the potential of being reused in new applications, avoiding redevelopment of already existing features, enabling savings and increasing agility. Other tool features are the process of reusable assets analysis and harvesting. This is an interactive graphic visualization of the results and an export mechanism of the identified assets through a widely adopted Metadata Representation Model. [Oliveira2007461]

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	DigitalAssets Discoverer
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	One of the main concerns in reusing software is to optimize the process of finding suitable components for a given need and providing enough information for the proper and efficient use of these components. It is useful for the analyst to have well structured descriptions related to the asset in the repository, when the components candidate to be reused are analyzed. Moreover, the task of tracking the reusable assets from a legacy applications portfolio, as they are consolidated and populated into digital libraries across environments, is highly dependable on supporting tools.
	Approach goals *(SQ1)*	Provide automatic identification of software components in order to help companies in their reuse and SOA initiatives anticipating the ROI (Return on Investment), and bring to light what companies have already developed by applying reuse indicators with sophisticated mechanisms to identify artifacts that can be considered as reusable assets.
	Visualizations' reuse-specific goals *(SQ1)*	Evaluate the candidates to become components, i.e., help to inspect a group of applications, configure and trigger the identification mechanisms, tune and reapply them in the analysis process.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (development / software components that can be reused in the development of new applications)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Discovering and evaluating potentially reusable assets (evaluate the candidates to become components, inspecting a group of applications)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Analyst (analyst)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (software assets (components, services, procedures, etc.), assets recommended for reuse, assets' artifacts, components suggested as reusable / module dependency, assets' relationships) • Source code and related information (relationships within the classes, knowledge base [structure of the source code])
	Source of visualized items/data *(TQ3.1)*	Existing applications selected for the analysis / Scanned source-code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The approach uses an API to obtain the identified cluster (which are considered as reusable components). The graph is derived as follows: (i) Detect the application roots in the dependencies graph and obtain reachable sub-graphs that will become the entry point for the dominance analysis process; (ii) Obtain a dominance tree (or a list of immediate dominators); (iii) Apply grouping heuristics; (iv) The suggested components and the consisting artifacts are presented and registered in the repository. A matrix representation and algorithms for sort and identification of cycles in DSM are also indicators to automatically find reusable components, as follows: (i) Get a "plain" representation of the packages structure (not considering the hierarchy) and the interdependencies found statically; (ii) Execute the partitioning algorithms using the Reachability Matrix Method and the algorithm of identification and grouping of cycles Path Searching in a DSM data structure; (iii) Some interpretations are considered for grouping (dependencies in series, in parallel and cycles); (iv) The suggested components and the constituting artifacts are presented and registered in the repository. The tool scans existing applications, executes a code analyzer to extract static information from source code (artifacts and relationships), and applies a series of indicators to obtain a group of artifacts with reuse potential, as follows: (i) Scanning the existing applications selected for the analysis; (ii) Creation of a knowledge base starting from the analysis, with the identification of internal and external references in addition to the internal architecture; (iii) Execution of the indicators (can be done repeatedly based on the reconfiguration of the indicators); (iv) Harvesting, when a group of artifacts suggested as reusable assets is presented, and the analyst decides for the relevance of this suggestion capturing the components and exporting them as a RAS package.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graphic-based visualization, graphs)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	N/A
	Visualization strategies and techniques *(TQ4.2)*	• Selection (interactive user graphical interface that enables the tuning of selected indicators) • Browsing / Navigation (navigation) • Clustering (hierarchical clustering allowing various grouping granularities) • Filtering / Highlighting/Mitigation (highlighting related classes) • Filtering / Inclusion/Removal (an interface offers the option of ignoring a group of assets) • Filtering / Tuning/Tweaking (tuning of selected indicators / an interface offers the option of modifying or importing a group of assets)

	Field	Information to be extracted
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The Reusable Asset Specification (RAS) model, Bunch and Prefuse.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	1) Practical use [probably by the authors themselves] (tests for proof of value) 2) Practical use (tests executing the tool)
	Application scenarios of the visualizations *(TQ7.1)*	1) Typical web applications, including component-based solutions as well as legacy applications with little architectural planning in terms of modularization. 2) Tests were conducted in partnership with an outsourcing software development company, using J2EE applications with more than 1000 Java classes. [commercial]
	Evaluated aspects *(TQ7.2)*	1) Not specified 2) Not specified
	Visualization evaluation results/outcomes *(TQ 7.3)*	1) Results were more successful in architecturally consistent applications. In the case of chaotic architecture applications the tool pointed quality features and some suggestions to improve its design before identifying reusable components. 2) The executions enabled the extraction of reusable assets and its inclusion in tools for governance.

Table 44. Gilligan [Holmes2007100]

	Field	Information to be extracted
Publication metadata	**Title**	Task-specific source code dependency investigation
	Authors	Holmes, R., Walker, R. J.
	Publication date (year/month)	June, 2007
	Publication type	Conference
	Source	Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Banff, Canada
	Pages	pp. 100-107
	Link (if applicable)	http://dx.doi.org/10.1109/VISSOF.2007.4290707
	Abstract	We present a simple, visual approach to help developers view and navigate structural dependency information, aimed specifically at pragmatic reuse tasks. Our visual approach, implemented as the Gilligan tool, uses standard GUI widgets (such as lists and editors) that developers are familiar with. Gilligan represents complex dependency data in a simplified format, appropriate for investigating reuse tasks. We present a small-scale, semi-controlled experiment that indicates that the approach permits more accurate identification of relevant structural dependencies with a lower time investment, as compared to traditional manual approaches. Last, we discuss the potential for the approach to aid in other specific software understanding tasks.
Visualization metadata	**Approach/tool name** *(PQ)*	Gilligan
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Developers often wish to reuse source code in ways that it has not been designed to be reused. As a given project supports functionality similar to what the developer wants to provide, she investigates this project to see if she can reuse any of its source code; while such project was not designed for reuse, it still may offer functionality she can benefit from. Developers undertaking such pragmatic reuse tasks can benefit from tool support to quickly and accurately identify the structural dependencies (e.g., classes and methods that they reference) of any source code fragments they are considering reusing. In this sense, Integrated development environments (IDEs) facilitate the navigation of the structural dependencies of source code. Understanding the scope of a source fragment's dependencies is essential for a developer to make an informed decision about whether to reuse the fragment or to reimplement its functionality. A source code entity need be investigated only if it is reachable transitively through other entities that are to be reused. If the referenced entity is of no use to the developer, she then has to consider whether the dependency can be dead-ended (i.e., the method call or field reference will be eliminated), or remapped to a different entity in her target system. Ultimately, she wants to minimize the unwanted functionality that will be incorporated into her system. While using a general graph-based visualization seems like a natural fit for this task, such visualizations fail to adequately support propagational navigation, quickly disorienting the developer with a proliferation of relationships and entities. Developers would follow paths just to check some fact, but would have trouble getting back to their starting point. Given that features of interest are often not well-encapsulated, an intricate and inexact decision-making process is still needed to draw the boundary between the feature and the rest of the system.
	Approach goals *(SQ1)*	Help developers view and navigate structural dependency information, aimed specifically at pragmatic reuse tasks, and allow developers to record their decisions as they investigate individual dependencies.
	Visualizations' reuse-specific goals *(SQ1)*	Reduce the cognitive effort required of the developer while investigating the structural dependencies for a source code fragment, and quickly identify and triage both the direct and indirect structural dependencies of any source code fragment.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (reuse source code)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' structure / asset information / repository (reduce the cognitive effort required while investigating, identifying and triaging direct and indirect structural dependencies for a source code fragment)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer with reuse (developers who are investigating and planning pragmatic reuse tasks)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (source code dependency)
	Source of visualized items/data *(TQ3.1)*	A system to be investigated for reuse.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The developer selects the system to be investigated for reuse, as well as a target system within which code reuse is intended. Then, the developer identifies at least one source code entity of interest (to him/her) from the source system. This is added to the leftmost (core concern) pane. The direct dependencies for any node that is selected in the core concern pane are shown in the central pane. The indirect dependencies of any selection in the direct dependency view are shown in the rightmost pane.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (tree lists) • Geometric forms (coloured rectangle)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	There are three columns in each of the tree panes. The leftmost column corresponds to the element's name, and provides a descriptive icon. This icon indicates the type of the node (package, class/interface, method, or field). The icon can also be decorated to provide extra information. Dependencies on types that are solely present as binaries (i.e., class files) are annotated with a slash through their icon; if a class has sub- or supertypes, a down- or up-arrow (respectively) is overlain on the icon. The second column is a coloured rectangle corresponding to the decision annotation the developer has placed on the node. The third column enumerates the number of direct dependencies of a node, while the fourth enumerates the number of dependencies in the transitive closure of the node's dependencies. Nodes are also annotated to show if they have been visited before; nodes that have not been visited are shown in lighter text than those that have. Green annotations correspond to dependencies on code that the developer wants to reuse. Red annotations correspond to code that she does not want to reuse. Blue annotations indicate code that performs functionality already provided within the developer's target system, but with a different interface. These three annotations are manually chosen by the developer according to her decisions as to how to triage dependencies. Yellow annotations are automatically generated by the system; these correspond to dependencies that are already provided within the target system.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (dependencies are automatically displayed based upon their selection) • Browsing / Navigation (nodes only appear in the direct dependency pane as a result of the developer's selections within the concern pane / navigable / propagational navigation / dependencies are automatically displayed based upon their selection) • Details on demand / Drill-down (visualization at any level between the package level to the detailed source code / view the system at varying degrees of detail) • Details on demand / Labeling (annotations can be added to any node in the visualization to allow the developer to tag nodes with different decisions they have made) • Filtering (filter the list) • Filtering / Collapse/Expand (expanded or contracted as required) • Presentation / Simultaneous (a three interdependent tree-list panes for abstractly representing source code dependencies; and an editor view for displaying source code itself) • Focus + context (nodes are always shown with some form of context, that is, their package and containing class is always visible)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in an Integrated Development Environment (Eclipse IDE)
	Resources used for interacting with the visualizations *(TQ5.1)*	• Mouse • Keyboard

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Eclipse IDE. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java as an Eclipse plug-in.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Semi-controlled experiment
	Application scenarios of the visualizations *(TQ7.1)*	Six developers (all of them software engineering graduate students) acted as subjects, analyzing four different source code fragments using either the proposed tool or standard IDE tools ("manually"). [academic]
	Evaluated aspects *(TQ7.2)*	The tool effectiveness, measured by the developers' ability to determine the structural dependencies.
	Visualization evaluation results/outcomes *(TQ 7.3)*	Developers were able to more completely identify structural relationships in less time using the proposed tool than with a manual approach.

Table 45. Stollberg & Kerrigan's approach [Stollberg2007236]

	Field	Information to be extracted
Publication metadata	**Title**	Goal-based visualization and browsing for semantic Web services
	Authors	Stollberg, M., Kerrigan, M.
	Publication date (year/month)	??, 2007
	Publication type	Article (Journal)
	Source	Lecture Notes in Computer Science
	Volume and Edition (for journals)	v. 4832 LNCS
	Place (for conferences)	N/A
	Pages	pp. 236-247
	Link (if applicable)	http://dx.doi.org/10.1007/978-3-540-77010-7_23
	Abstract	We present a goal-based approach for visualizing and browsing the search space of available Web services. A goal describes an objective that a client wants to solve by using Web services, abstracting from the technical details. Our visualization technique is based on a graph structure that organizes goal templates – i.e. generic and reusable objective descriptions – with respect to their semantic similarity, and keeps the relevant knowledge on the available Web services for solving them. This graph is generated automatically from the results of semantically enabled Web service discovery. In contrast to existing tools that categorize the available Web services on the basis of certain description elements, our tool allows clients to browse available Web services on the level of problems that can be solved by them and therewith to better understand the structure as well as the available resources in a domain. This paper explains the theoretic foundations of the approach and presents the prototypical implementation within the Web Service Modeling Toolkit WSMT, an Integrated Development Environment for Semantic Web services.

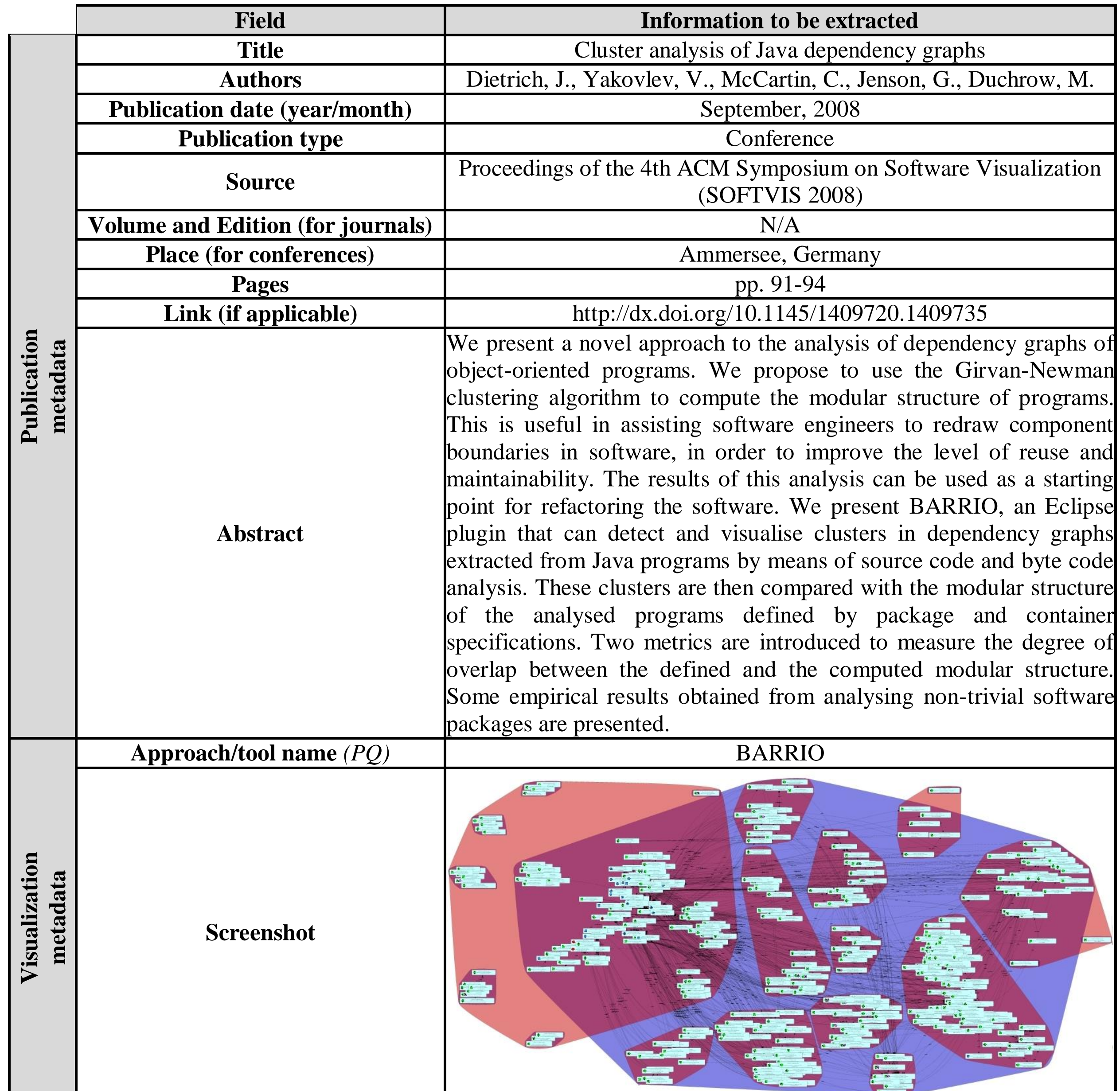
	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	N/A
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The provision of suitable search facilities for Web services is one of the major challenges for realizing sophisticated SOA technologies, which requires support for the search and inspection of potential candidate services for a specific problem.
	Approach goals *(SQ1)*	Browse and understand the available Web services on the level of the problems that can be solved by them, in terms of the structure as well as the available resources in a domain.
	Visualizations' reuse-specific goals *(SQ1)*	Aid clients in the goal instance formulation process and allow them to better understand the available resources (Web services) as well as the problems that can be solved by them.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Analysis / Specification / Requirements engineering (goal-based / focus on the problem to be solved, abstracting from technical details)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (better understand available resources (Web services)) • Searching and retrieving reusable assets (aid in the goal instance formulation process / better understand the problems that can be solved by Web services)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer with reuse (web service application developers / clients)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Requirements/Analysis artifacts and related information (goal templates (generic and reusable descriptions of objectives that clients want to achieve by using Web services)) • Web services and related information (available Web services, usability of Web services in a problem domain with respect to the goals that can be solved by them)
	Source of visualized items/data *(TQ3.1)*	Semantically enabled Web service discovery.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The graph structure is organized in terms of the semantic similarity, calculated from Web service discovery runs/results. The semantic matchmaking on the goal template and the goal instance level is based on rich functional descriptions, i.e., sufficiently rich formal descriptions of goals and Web services. The Semantic Discovery Caching (SDC) graph is automatically generated from the results of design time Web service discovery on goal templates. It organizes goal templates in a subsumption hierarchy with respect to their semantic similarity, which constitutes the indexing structure of the available Web services. The usability cache, in it turn, is generated from the results of Web service discovery on the goal template level that is performed at design time. The discovery operations use this knowledge structure by inference rules. The functional usability of a Web service W for a goal template G by the matchmaking degrees.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (directed graph structure)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The upper layer of the Semantic Discovery Caching graph is the goal graph that defines the subsumption hierarchy of goal templates by directed arcs. The lower layer is the usability cache that explicates the usability of each available Web service by directed arcs that are annotated with the usability degree. Leaf nodes represent the available Web services that are functionally usable for solving the goal templates; such suitability for solving goal templates is explicated by directed arcs. Disconnected subgraphs indicate that there are two goal templates that do not have any common solution.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (by double-clicking on a goal template in the SDC graph, the user can step down to the next level) • Browsing / Navigation (browsing facilities / browse / navigate / multi-leveled browsing facilities / presenting all relevant information to the user in a browsable fashion) • Details on demand / Drill-down (by double-clicking on a goal template in the SDC graph, the user can step down to the next level) • Details on demand / Labeling (directed arcs annotated with the usability degree) • Filtering / Inclusion/Removal (filtering / redundant arcs are omitted) • Overview + detail (complete overview / more detailed perspectives / from the complete overview of the search space down to detailed views on individual resources) • Layout (layout algorithms / employs a simple vertical-tree layout; however, a spring-layout algorithm where the nodes in the graph repel each other while the edges between nodes draw them back together can be employed to display larger and more complex SDC graphs) • Zooming / Geometric (zoom) • Panning / Drag-and-drop (dragging and dropping nodes) • Hierarchical visualization (hierarchy) • Rotating (rotate)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in an Integrated Development Environment (Web Service Modeling Toolkit WSMT, an Integrated Development Environment implemented in the Eclipse framework)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Web Service Modelling Toolkit WSMT, an IDE for the Semantic Web service technology based on the Eclipse framework. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The SDC graph visualization is implemented as a plug-in for the Web Service Modeling Toolkit (WSMT). The WSMT Visualizer (that provides a graph-based editor and browser for ontologies) was extended. The JPowerGraph graphing library and other WSMO framework elements in WSMT are also used.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 46. BARRIO [Dietrich200891]

	Field	Information to be extracted
Publication metadata	**Title**	Cluster analysis of Java dependency graphs
	Authors	Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G., Duchrow, M.
	Publication date (year/month)	September, 2008
	Publication type	Conference
	Source	Proceedings of the 4th ACM Symposium on Software Visualization (SOFTVIS 2008)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Ammersee, Germany
	Pages	pp. 91-94
	Link (if applicable)	http://dx.doi.org/10.1145/1409720.1409735
	Abstract	We present a novel approach to the analysis of dependency graphs of object-oriented programs. We propose to use the Girvan-Newman clustering algorithm to compute the modular structure of programs. This is useful in assisting software engineers to redraw component boundaries in software, in order to improve the level of reuse and maintainability. The results of this analysis can be used as a starting point for refactoring the software. We present BARRIO, an Eclipse plugin that can detect and visualise clusters in dependency graphs extracted from Java programs by means of source code and byte code analysis. These clusters are then compared with the modular structure of the analysed programs defined by package and container specifications. Two metrics are introduced to measure the degree of overlap between the defined and the computed modular structure. Some empirical results obtained from analysing non-trivial software packages are presented.
Visualization metadata	**Approach/tool name** *(PQ)*	BARRIO
	Screenshot	

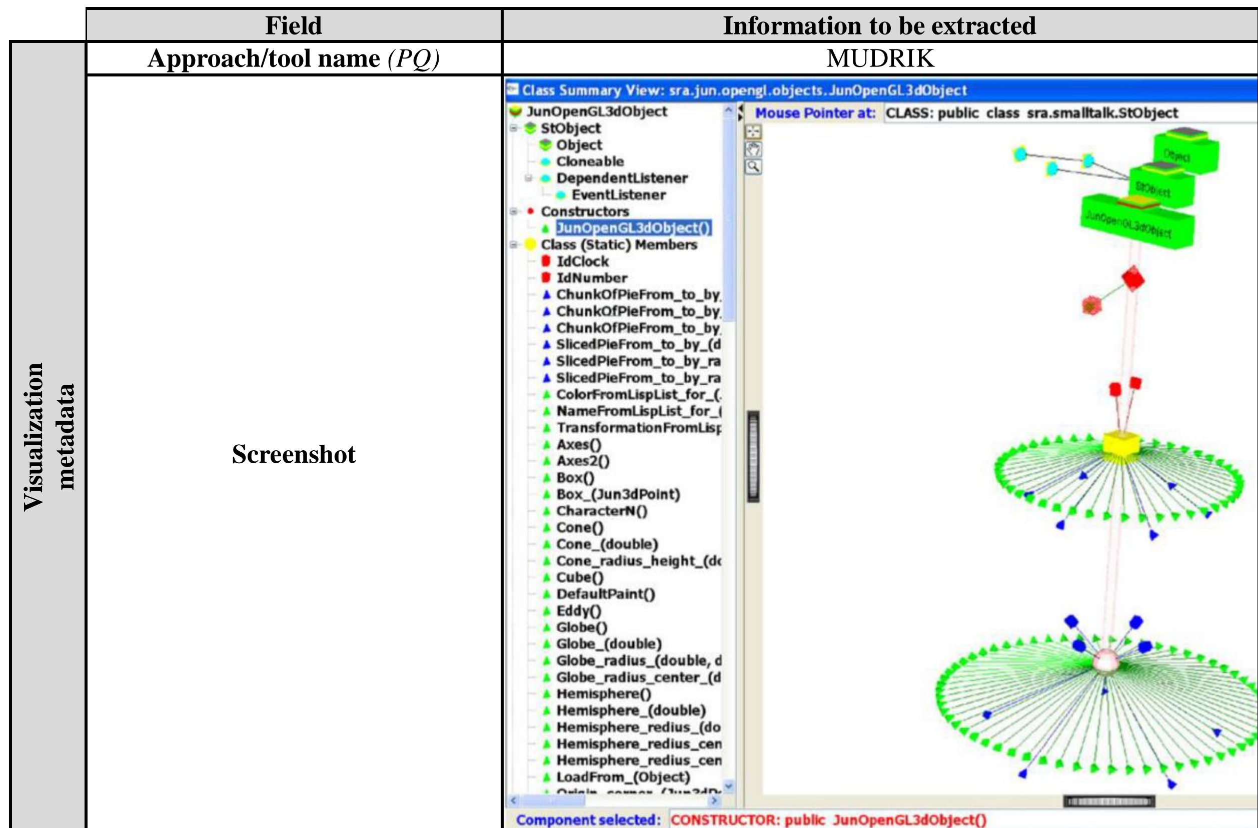
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The question arises of how existing, monolithic programs can be refactored into component models. Since the edges that lie between clusters are expected to be those with highest betweenness (a centrality measure for edges in a graph), it is possible to find a good separation of the graph into clusters by removing them recursively. Unfortunately, byte code analysis cannot discover all relationships. Source code analysis, in its turn, obviously requires the availability of source code. To some extent, the different dependency graphs that can be extracted with byte code and source code analysis reflect different aspects of dependency analysis related to design time and runtime modularity of software. Design time and runtime modules do often overlap, but not always. Therefore, dependency analysis from source code or byte code sometimes serves different purposes: modular organisation of source code to optimise development, and modular organisation of runtime artefacts to facilitate deployment and maintenance.
	Approach goals *(SQ1)*	Detect and visualise clusters in dependency graphs, and produce a list of refactorings that can be used to transform programs into a more modular structure, one that is easier to customise and to maintain.
	Visualizations' reuse-specific goals *(SQ1)*	Assist software engineers to redraw component boundaries in software, in order to improve the level of reuse and maintainability.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software maintenance (refactoring the software / optimise development / facilitate deployment and maintenance)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Restructuring assets for reuse (assist to redraw component boundaries in software, in order to improve the level of reuse and maintainability)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Software engineer (software engineers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (clusters of source code and byte code based on their dependencies)
	Source of visualized items/data *(TQ3.1)*	Java programs (source code and byte code).
	Collection procedure/method of visualized items/data *(TQ3.2)*	Information is extracted from Java programs by means of source code and byte code analysis. The betweenness (a centrality measure for edges in a graph) is defined as the number of shortest paths between all pairs of nodes in the graph passing through that edge. The steps are as follows: (1) Calculate the betweenness value for each of the edges; (2) Remove the edge(s) having the highest value; (3) Repeat the analysis on the resulting graph until a suitable separation of the graph into clusters has been achieved.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graph)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The nodes in the dependency graph are types, while the edges represent relationships between those types. The nodes in the generated graph represent Java types. They have annotations defining their classification (class, interface, annotation, etc.), visibility, abstractness, and whether they are final. Every node also contains a list of one way relationships (dependencies) with the full name of the class (packageName.className) referenced and a dependency classification annotation (uses, extends or implements). The nodes are displayed with labels that contain the name of the class, the namespace, the name of the container that the class belongs to, and an icon that reflects class properties. The relationships between classes are represented as directed edges with labels describing the type of relationship (extends, implements or uses).
	Visualization strategies and techniques *(TQ4.2)*	• Selection (the visibility of aggregates is an optional selection by the user) • Details on demand / Labeling (annotations attached to nodes and edges / the nodes are displayed with labels / edges with labels describing the type of relationship) • Clustering (to display groups of nodes, the visualisation uses elements called aggregates / an aggregate draws a border around a group of nodes that contain the same value for a particular property) • Filtering / Highlighting/Mitigation (highlighting occurs upon user action / highlighting paints an edge and its end nodes with a darker colour and brings end nodes to the front) • Filtering / Inclusion/Removal (edges that have the maximum betweenness value are removed from the graph / the annotations attached to nodes and edges can be used to define filters / filters can be applied to edges and nodes) • Layout (force directed layout is used to position visual elements of the graph, and is replaced by a static radial tree layout for graphs that contain over 1200 nodes) • Zooming / Geometric (zoom in or out on parts of the graph / zoom) • Panning (pan) • Panning / Drag-and-drop (visual element drag / on user request any visual node can be moved to any position on the screen) • Animation (smooth animated zoom and pan which helps the user to preserve a sense of position and context)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in an Integrated Development Environment (Eclipse IDE)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Eclipse IDE. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java as an Eclipse plugin. It uses the Object Dependency Exploration Model (ODEM) – a tool and platform independent XML vocabulary –, the Class Dependency Analyzer (CDA) tool, the Eclipse AST (Abstract Syntax Tree) API and the Prefuse visualisation toolkit. The JUNG implementation of the Girvan-Newman algorithm is also used.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (we have analysed a number of programs, including some popular open source programs)
	Application scenarios of the visualizations *(TQ7.1)*	A PC with a Intel Core 2 6600@2.4 GHz processor and 2 GB of memory, analyzing a number of programs, including a product supplied by a New Zealand company providing software solutions for the packaging industry, and some popular open source programs such as Xerces, Xalan, Commons-collections, and the MySQL ConnectorJ JDBC driver. [commercial] [open source]
	Evaluated aspects *(TQ7.2)*	Scaleability of the tool
	Visualization evaluation results/outcomes *(TQ 7.3)*	Calculating the connected components of the initial graph took only 25s. Cluster analysis with the separation level set to ten, i.e the clustering algorithm cycled through ten iterations, took 3min 10s. In none of the programs analysed did increasing the separation level have a big impact. This is interpreted as an indication that these programs already have a well defined modular structure.

Table 47. MUDRIK [Ali200950]

	Field	Information to be extracted
Publication metadata	**Title**	Cognitive support through visualization and focus specification for understanding large class libraries
	Authors	Ali, J.
	Publication date (year/month)	??, 2009
	Publication type	Article (Journal)
	Source	Journal of Visual Languages and Computing
	Volume and Edition (for journals)	v. 20, n. 1
	Place (for conferences)	N/A
	Pages	pp. 50-59
	Link (if applicable)	http://dx.doi.org/10.1016/j.jvlc.2008.02.001
	Abstract	Effective object-oriented (OO) programming requires understanding class libraries. This paper presents our approach to design and build a cognitive tool that supports a programmer to understand OO class libraries. The MUDRIK system provides (1) three-dimensional visualization mechanisms for representing class structures and relationships from a variety of views and (2) flexible focus specification mechanisms that allow users to adapt a space of components to be displayed according to the task at hand. Interactive views of MUDRIK enable programmers to examine components' detail while maintaining a global representation of the rest of the library. The paper describes why understanding class library is critical in OO programming, presents a cognitive framework of our approach and design rationale behind the system design, and provides a detailed description of the system followed by a discussion on our approach.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	MUDRIK
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	OO programmers who are not familiar with a class library have hard time finding reusable classes/objects because (1) they do not understand the structure of the library, (2) they do not know what keywords to use to retrieve relevant objects and (3) they do not know what to look for because they do not know what functionality is available in the class library. For successfully reusing objects provided in libraries, understanding their structure and functionality is a prerequisite. However, without adequate system support in the programming process, it is difficult to locate and understand appropriate objects. The richer the library is, the more expensive to access it in terms of both computational and cognitive costs. As open-source development style increases and more and more not-well-organized programming projects contribute their products as OO libraries, more and more class libraries are released without proper documentation and source code. Understanding class libraries with the traditional means, such as reading textual documentation or source code, is a difficult and time-consuming task. It is often impossible to keep track of all complex codependences among components of a library acquired through reading them. Some Integrated Development Environments help programmers to see class-subclass relationships or internal details of a particular class. However, the mechanism is not enough for helping a programmer who downloads a class library that contains several hundred classes. The value of the growing number of available class libraries depends on programmers' ability to reuse them. Reusing class libraries requires understanding their structure and functionality. Understanding a class requires a programmer to understand the context where the class is used. The meaning of the whole is determined by its components while the meaning of each component can only be understood in terms of the whole.
	Approach goals *(SQ1)*	Support the understanding of a potentially large class library (i.e., existing OO systems/class libraries) in a relatively short span of time, allowing programmers to find useful information in the library by helping them understand what is important and relevant.
	Visualizations' reuse-specific goals *(SQ1)*	Locate and understand appropriate objects.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Programming / Coding (OO programming)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (understand appropriate objects) • Searching and retrieving reusable assets (locate appropriate objects)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (Java programmers)

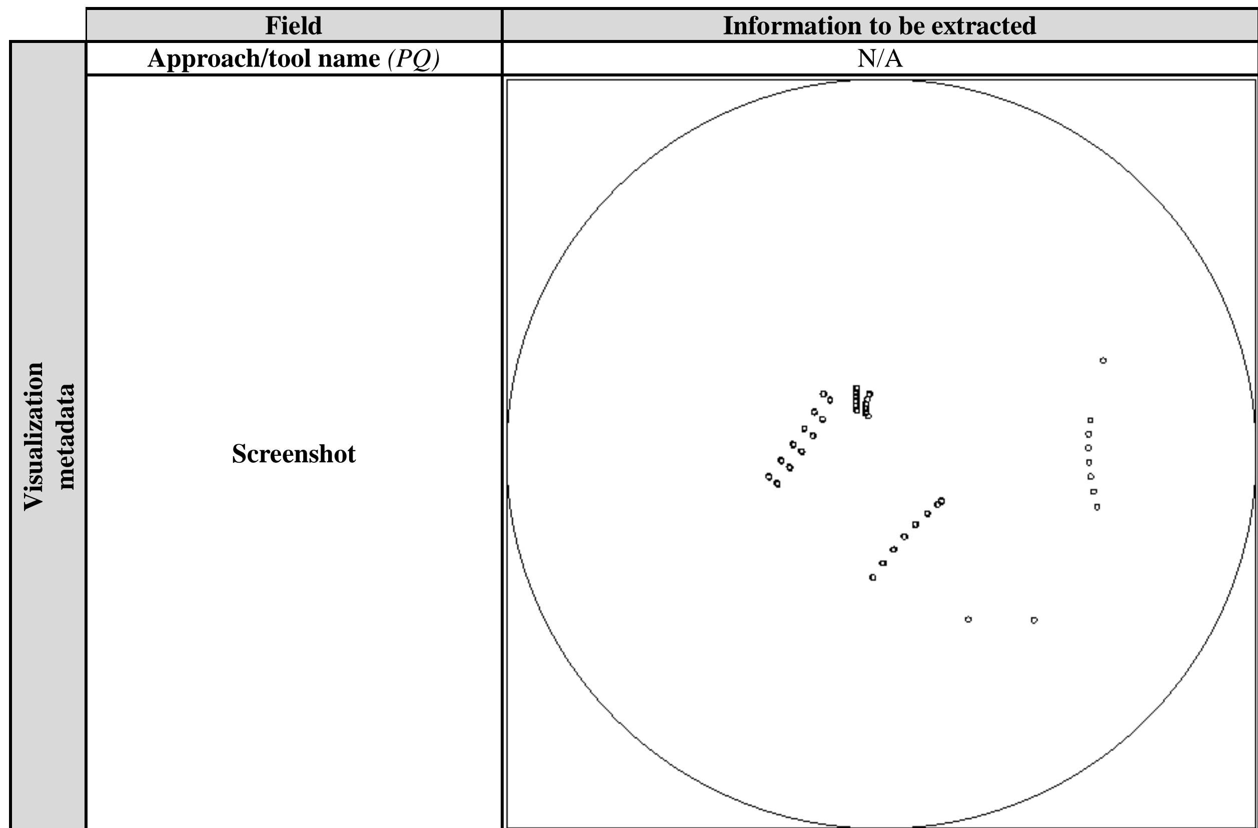
	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (class libraries, the library as a whole / objects contained in the library, all referential relationships among classes of the library (which occurs when class A uses class B as the type of its instance variable(s), parameter(s) or method(s) returned value), inheritance tree of all the classes) • Source code and related information (class structures / for a given (target) class: its super and subclass names, member fields and methods, and all its referential relationships with other classes, all the classes that are referenced by the target class, all the classes referencing the target class, subclasses of a class, all the superclasses of the target class, all the interfaces implemented by the target class directly or via one of its ancestor class, and members (constructors, fields and methods) of the target class)
	Source of visualized items/data *(TQ3.1)*	Java libraries/systems (in the form of Java class files or JAR files).
	Collection procedure/method of visualized items/data *(TQ3.2)*	MUDRIK uses one or more Java libraries/systems in the form of Java class files or JAR files as its input. By opening a new library and specifying the files or directories that contain the library, the system loads all the classes, interfaces and packages stored in the library. Then the system analyzes the loaded entities of the library and collects detailed information about them.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Hierarchy (tree structure, interactive cone trees)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	All the classes/interfaces are shown in a tree structure based on the packages of the library. A number of 3D icons are defined to represent various entities in a library, such as classes, interfaces and packages. The parts and colors of an icon represent different properties of the corresponding entity. For example, the top of the icon for an abstract class is yellow while that of a final class is red. The layout of icons in a view depends on the underlying relationships among the corresponding entities. Different colors are used to distinguish between private, protected or public members of a class. Positions and colors are used to represent all types of relationships. All the classes that are referencing other classes are placed along the X-axis, and all the classes that are referenced are placed along the Y-axis. This makes a grid like structure where each crossing represents a relationship between the two classes lying perpendicular to the crossing. The red, blue and yellow bars on the grid crossing show the existence of referential relationships by instance variables, parameters and methods returned types, respectively. The heights of these bars (along the positive Z-axis) represent the number of instance variables, parameters or methods that are linking the two classes. Both the referencing and the referenced classes are sorted based on their reference values. The reference values are graphically shown as vertical bars along the negative Z-axis. The bars in cyan color represent reference values for referencing classes and those in green color represent the reference values for referenced classes.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (an item is clicked on in any view) • Filtering (flexible focus specification mechanisms that allow users to adapt a space of components to be displayed / filter out unneeded classes / filters) • Filtering / Highlighting/Mitigation (highlighting objects of "interest") • Overview + detail (examine components' detail while maintaining a global representation of the rest of the library / give a feeling of "what's there" without overwhelming detail preciseness / access detailed accurate information if needed / transitions from one view to another maintaining the current context / overview + detail) • Layout (show large amount of information in a limited space and avoid overlapping representations in displaying complex structures) • Layout / 3D (3D visualization) • Zooming / Geometric (zoomed in/out) • Rotating (rotated around any axis) • Presentation (multiple visualizations) • Overlap / Flipping (the programmer can go back and forth between the high-level abstract context and low-level detailed information of the library without interfering his/her cognitive processes (i.e., with minimum cognitive load) / smooth transitions from one view to another, maintaining the current context by integrating views with consistent interaction styles) • Linking (whenever an item is clicked on in any view, the item is selected in both of the views / all the views are integrated)

	Field	Information to be extracted
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (an interactive environment / window)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: It runs in Windows 95/98/NT/2000/XP, and requires OpenGL and Jun for Java. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, it uses the Java reflection mechanism. It also uses OpenGL and Jun.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 48. Damaševičius's approach [Damaeviius2009507]

	Field	Information to be extracted
Publication metadata	**Title**	Analysis of components for generalization using multidimensional scaling
	Authors	Damaševičius, R.
	Publication date (year/month)	??, 2009
	Publication type	Article (Journal)
	Source	Fundamenta Informaticae
	Volume and Edition (for journals)	v. 91, n. 3-4
	Place (for conferences)	N/A
	Pages	pp. 507-522
	Link (if applicable)	http://dx.doi.org/10.3233/FI-2009-0054
	Abstract	To achieve better software quality, to shorten software development time and to lower development costs, software engineers are adopting generative reuse as a software design process. The usage of generic components allows increasing reuse and design productivity in software engineering. Generic component design requires systematic domain analysis to identify similar components as candidates for generalization. However, component feature analysis and identification of components for generalization usually is done ad hoc. In this paper, we propose to apply a data visualization method, called Multidimensional Scaling (MDS), to analyze software components in the multidimensional feature space. Multidimensional data that represent syntactical and semantic features of source code components are mapped to 2D space. The results of MDS are used to partition an initial set of components into groups of similar source code components that can be further used as candidates for generalization. STRESS value is used to estimate the generalizability of a given set of components. Case studies for Java Buffer and Geom class libraries are presented.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	N/A
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Reuse cannot be achieved without some form of generalization. Generic component design requires systematic domain analysis to identify similar components as candidates for generalization. However, component feature analysis and identification of components for generalization usually is done ad hoc. Unsuccessful partitioning may lead to unsuccessful generalization and un-usable (un-reusable) generic components. Separation and identification of common and variable concerns in the domain is a step towards achieving generalization. The components may have different feature dimensions, e.g., syntactical (based on component source code properties) or semantic (based on functionality of the components). Discovering and understanding program similarity allows for efficient development of new component architectures and systems and for well-organized maintenance of existing systems. There is still lack of understanding among software developers that all these aspects of program similarity are related and must be managed explicitly.
	Approach goals *(SQ1)*	Analyze software components in the multidimensional feature space, partitioning an initial set of components into groups of similar source code components that can be further used as candidates for generalization (generalization is mainly used for developing reusable software components and reuse libraries).
	Visualizations' reuse-specific goals *(SQ1)*	Visualize multidimensional software component feature space and identify clusters of similar components as candidates for generalization (the more there are similarities between the generalized components, the better generalization can be achieved, which ultimately allows for better component reuse, library scaling and maintenance).
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software development for reuse (development of new component architectures and systems) • Software maintenance (maintenance of existing systems)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Discovering and evaluating potentially reusable assets (identifying clusters of similar components as candidates for generalization)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (software developers)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (component classes on their feature space)
	Source of visualized items/data *(TQ3.1)*	Component source code, feature models or domain business models (ontology, thesaurus).
	Collection procedure/method of visualized items/data *(TQ3.2)*	(1) Identify a set of components C available for generalization; (2) Identify a set of features F of each component, which may be extracted from component source code, feature models or domain business models (ontology, thesaurus) using visual inspection, domain analysis tools (e.g., parsers) and may include syntactical features that characterize the source code of components or semantic features that characterize the functionality (behavior) of a component; (3) Build a component feature matrix M (component × feature). The feature matrix must include at least 6 features. It represents a set of points in a multidimensional feature space; (4) Digitize a feature matrix. The numerical values for natural language descriptions of features, if any, must be provided; (5) Select a distance metric (see Eq. 9) to measure the dissimilarity between components in component feature space; (6) Select a stress criterion (see Eq. 4, 5 or 6) that estimates the error of the mapping between the multidimensional feature space and its 2D image; (7) Perform MDS on a feature matrix to obtain its 2D projection and a stress value; (8) Identify clusters in the 2D projection. Identification is usually performed by visual inspection of the 2D projection; (9) Use clusters of components to build generic components. The number of identified clusters determines the number of generic components; (10) Evaluate generalizability using stress value.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Map (points in a 2D space)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each object is represented by a point, and the distances between points resemble the original similarity information; i.e., the larger the dissimilarity between two objects, the farther apart they should be in the lower dimensional (usually 2D) space. Distances between objects in multidimensional space are related to their dissimilarities linearly. A component can be represented as a point in a n-dimensional feature space. The similarity between two components is defined as a distance between two points in a multidimensional feature space, and its calculation is made in terms of their features (a feature is a computable metric of a given component). A cluster of components is a group of similar components separated by a small distance. The classes that are more similar are located closer to each other.
	Visualization strategies and techniques *(TQ4.2)*	Clustering (MDS maps the high-dimensional data into a lower-dimensional space / this geometrical configuration of points reflects the hidden structure of the data and may help to make it easier to understand)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	N/A
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (case studies)[36]
	Application scenarios of the visualizations *(TQ7.1)*	Java Buffer and Geom class libraries, with stress criterion. [open source]
	Evaluated aspects *(TQ7.2)*	Measurement of MDS, Euclidean distance metrics.
	Visualization evaluation results/outcomes *(TQ 7.3)*	No clusters could be identified with MDS of Buffer classes using syntactic features. However, by partitioning the Buffer library based on the MDS of Buffer classes using semantic features, 6 different generic components can be developed. After analysis of the MDS of Geom classes using syntactic features, 7 different clusters can be identified, which can be implemented as generic components.

[36] Section 4.8 discusses the adequacy of the "case study" denomination, based on the experimental software engineering literature.

Table 49. Ontology-Driven Visualization (ODV) [DeBoer200951]

	Field	Information to be extracted
Publication metadata	**Title**	Ontology-driven visualization of architectural design decisions
	Authors	De Boer, R. C., Lago, P., Telea, A., Van Vliet, H.
	Publication date (year/month)	September, 2009
	Publication type	Conference
	Source	Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Cambridge, UK
	Pages	pp. 51-60
	Link (if applicable)	http://dx.doi.org/10.1109/WICSA.2009.5290791
	Abstract	There is a gradual increase of interest to use ontologies to capture architectural knowledge, in particular architectural design decisions. While ontologies seem a viable approach to codification, the application of such codified knowledge to everyday practice may be non-trivial. In particular, browsing and searching an architectural knowledge repository for effective reuse can be cumbersome. In this paper, we present how ontology-driven visualization of architectural design decisions can be used to assist software product audits, in which independent auditors perform an assessment of a product's quality. Our visualization combines the simplicity of tabular information representation with the power of on-the-fly ontological inference of decision attributes typically used by auditors. In this way, we are able to support the auditors in effectively reusing their know-how, and to actively assist the core aspects of their decision making process, namely trade-off analysis, impact analysis, and if-then scenarios. We demonstrate our visualization with examples from a real-world application.
Visualization metadata	**Approach/tool name** *(PQ)*	Ontology-Driven Visualization (ODV)
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Browsing and searching an architectural knowledge repository for effective reuse can be cumbersome. It can be hard to explore and search an architectural knowledge repository so that previously captured knowledge can be reused. Decision tables are the most often used type of visualization for browsing. Yet, such a view has several drawbacks. Most notably, a list or table is not very effective in showing relationships. As such, it ignores much of the added value of using an ontology. A decision-structure visualization, which seems to be the most natural visualization for a decision ontology, has drawbacks too. While it accurately represents decisions and their relationships, the resulting graph can become cluttered and thus incomprehensible for all but the smallest data sets. When performing several software product audits, some quality criteria may be reused. It is assumed that the audit organization has used QuOnt to codify quality criteria from previous audits.
	Approach goals *(SQ1)*	Support the auditors in effectively reusing their know-how and assist the core aspects of their decision making process, namely trade-off analysis, impact analysis, and if-then scenarios.
	Visualizations' reuse-specific goals *(SQ1)*	Allow to perform a trade-off analysis for determining which quality criteria to include in an audit, select and prioritize the quality attributes to be used in such audit and support the auditor in deciding which quality criteria to use.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Quality assurance / Testing / Debugging / Profiling (software product audits / assessment of a product's quality)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Searching and retrieving reusable assets (determining which quality criteria to include in an audit, select and prioritize the quality attributes to be used in his audit and support the auditor in deciding which quality criteria to use)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Auditor (independent auditors)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Architecture / Design artifacts and related information (quality attributes of interest / hierarchy of quality attributes, quality criteria relevant to the current audit, relations between quality criteria)
	Source of visualized items/data *(TQ3.1)*	A knowledge base.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The QuOnt ontology is used to codify quality criteria for reuse, and forms the basis of the ontology-driven visual analysis.

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (tree) • Matrix / Matrix-like (matrix, 2D matrix)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The 'Quality attribute tree' shows the hierarchy of quality attributes according to a particular quality model. The 'Quality attributes of interest' area shows the quality attributes of interest, which capture the customer's idea of 'quality'. The 'Effect matrix' shows the quality criteria relevant to the current audit. The 'Criteria matrix' shows the relations between quality criteria. 2D matrix layouts are used for showing relations. A color scheme is used for representing criteria relations.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (interactive selection) • Details on demand / Labeling / Tooltip (tooltips with details on the relations) • Filtering / Highlighting/Mitigation (visual highlighting / a small set of contrasting colors, which is effective in attracting the user's attention to salient events / brushing with the mouse over the matrix cells) • Panning / Drag-and-drop (drag-and-drop) • Presentation / Simultaneous (two tabular views that show design decisions and their mutual relations, respectively) • Linking (linked views)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: It is assumed that the audit organization has used QuOnt to codify quality criteria from previous audits. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The QuOnt ontology.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [by others]
	Application scenarios of the visualizations *(TQ7.1)*	Assessed by auditors of DNV-CIBIT. [commercial]
	Evaluated aspects *(TQ7.2)*	Not specified
	Visualization evaluation results/outcomes *(TQ 7.3)*	The auditors reacted very positively. Especially the easy selection of quality criteria and the way the tool invites the user to 'play around' and consider 'what if' scenarios were cited as the tool's main benefits.

Table 50. NFRs and Design Rationale (NDR) Ontology / Toeska/Review tool [López20091198]

	Field	Information to be extracted
Publication metadata	Title	Visualization and comparison of architecture rationale with semantic web technologies
	Authors	López, C., Inostroza, P., Cysneiros, L. M., Astudillo, H.
	Publication date (year/month)	??, 2009
	Publication type	Article (Journal)
	Source	Journal of Systems and Software
	Volume and Edition (for journals)	v. 82, n. 8
	Place (for conferences)	N/A
	Pages	pp. 1198-1210
	Link (if applicable)	http://dx.doi.org/10.1016/j.jss.2009.03.085
	Abstract	Deciding how to operationalize non-functional requirements (NFR) is a complex task, and several formalisms have been proposed to represent design decisions and their rationale. Unfortunately, these models can become complex (even unreadable) for designs with many alternatives and/or a well-documented rationale, which makes very hard to review and compare rationale. This paper introduces a Semantic Web-based technique to visualize and compare architecture rationale, combining Softgoal Interdependency Graphs (SIGs) with ontologies reified as named graphs. Reuse of rationale is thus facilitated by allowing architects to understand rationale of previous decisions and/or projects, though automated reuse remains unfeasible until extensive automated capture rationale happens. The approach is illustrated with a case study of Contexta, a museum integration project, using Toeska/Review, a Semantic Web-based tool.
Visualization metadata	Approach/tool name *(PQ)*	NFRs and Design Rationale (NDR) Ontology Toeska/Review tool[37]
	Screenshot	

[37] The Toeska/Review tool has been built to allow visualization and comparison of SIGs represented with the NDR Ontology

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Most representations of software architecture focus on the system structure and hide the decision making process, discarded alternatives, tradeoff analysis and rationale behind the finally adopted choices. Softgoal Interdependency Graphs (SIGs) can become quite difficult to read, and their complexity hampers their broader use by practicing architects. Most non-trivial projects can produce quite large and complex design graphs, which also complicates recovering alternatives, tradeoff and rationale information in SIGs. Recording and collecting past architectural decisions and their rationale is required to review, compare and eventually reuse prior knowledge.
	Approach goals *(SQ1)*	Describe SIGs through an ontology and represent them as named graphs, enabling their view-based exploration and comparison of decisions and rationales.
	Visualizations' reuse-specific goals *(SQ1)*	Facilitate reuse of rationale by allowing architects to understand rationale of previous decisions and/or projects, supporting, for example, the selection between reuse candidates by identifying domain constraints or contexts that are more similar to the problem at hand.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software design (architecture design)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (understand rationale of previous decisions and/or projects) • Integrating reusable assets (supporting the selection between reuse candidates by identifying domain constraints or contexts that are more similar to the problem at hand)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Software architect/designer (architects)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Architecture / Design artifacts and related informatio (architecture rationale / Softgoal Interdependency Graphs (SIGs) [used to represent quality attributes])
	Source of visualized items/data *(TQ3.1)*	Semantic descriptions (as NDR instances) in SIGs.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Rationale information is recorded in a SIG using claims and argumentations; hence, a rationale view can be operationalized by querying claims and argumentations instances of a named graph that represents a SIG. SIG comparison and rationale visualization tools process SIGs information by reading their semantic descriptions (as NDR instances).
Representatio n (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graphs)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each kind of softgoal has visually different icons that identify them.
	Visualization strategies and techniques *(TQ4.2)*	• Details on demand / Labeling (labeling each softgoal) • Filtering / Highlighting/Mitigation (highlights differences)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (web application)
	Resources used for interacting with the visualizations *(TQ5.1)*	Keyboard (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Protégé (or similar) and the OWL language (for describing the SIG). • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented as a Web application that uses the SPARQL query language to recover NFR knowledge embedded into NDR instances, and uses Softgoal Interdependency Graphs (SIGs). The NDR Ontology is written in OWL.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (the approach is illustrated with a case study[38])
	Application scenarios of the visualizations *(TQ7.1)*	Evaluated by architects of project Contexta, a museum integration project, to review its design rationale and to compare their decisions about extensibility with those of another project, Tutelkan. [commercial]
	Evaluated aspects *(TQ7.2)*	Not specified
	Visualization evaluation results/outcomes *(TQ 7.3)*	The architects were able to clearly identify the divergence points of the two decision processes. Although the visualization and comparison tool did help to speed up the comparison of design rationales, the comparison itself of evaluation and interdependencies remained hard to visualize in the graphs. The architects also reported that change impact and tradeoff analysis would still require a deep problem understanding.

[38] Section 4.8 discusses the adequacy of the "case study" denomination, based on the experimental software engineering literature.

Table 51. AMPLE Traceability Framework (ATF) [Anquetil2010427]

	Field	Information to be extracted
Publication metadata	**Title**	A model-driven traceability framework for software product lines
	Authors	Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., Sousa, A.
	Publication date (year/month)	??, 2010
	Publication type	Article (Journal)
	Source	Software and Systems Modeling
	Volume and Edition (for journals)	v. 9, n. 4
	Place (for conferences)	N/A
	Pages	pp. 427-451
	Link (if applicable)	http://dx.doi.org/10.1007/s10270-009-0120-9
	Abstract	Software product line (SPL) engineering is a recent approach to software development where a set of software products are derived for a well defined target application domain, from a common set of core assets using analogous means of production (for instance, through Model Driven Engineering). Therefore, such family of products are built from reuse, instead of developed individually from scratch. SPL promise to lower the costs of development, increase the quality of software, give clients more flexibility and reduce time to market. These benefits come with a set of new problems and turn some older problems possibly more complex. One of these problems is traceability management. In the European AMPLE project we are creating a common traceability framework across the various activities of the SPL development. We identified four orthogonal traceability dimensions in SPL development, one of which is an extension of what is often considered as “traceability of variability”. This constitutes one of the two contributions of this paper. The second contribution is the specification of a metamodel for a repository of traceability links in the context of SPL and the implementation of a respective traceability framework. This framework enables fundamental traceability management operations, such as trace import and export, modification, query and visualization. The power of our framework is highlighted with an example scenario.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	AMPLE Traceability Framework (ATF)
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The use of traceability is considered a factor of success for software engineering projects. However, traceability can be impaired by various factors ranging from social, to economical, and to technical. None of the investigated tools has built-in support for Software Product Line (SPL) development, and a vast majority of them are closed, so they cannot be adapted to deal with the issues raised by SPL. None of them provides a clear and comprehensive view of the trace links in a SPL development. A thorough analysis of the dimension in SPL is needed, with specific emphasis on variability and versioning. Visualizing traceability links is important, but getting a useful view is a non trivial task. However, more advanced support is the responsibility of the information and visualization community.
	Approach goals *(SQ1)*	Solve complex traceability problems, by allowing the definition of hierarchical artefact and link types as well as constraints between these types.
	Visualizations' reuse-specific goals *(SQ1)*	Observe the structure of the feature model and the evolution of the realization of the features, and compare the refinement sets of different versions.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software product line engineering (software product line engineering)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (observe the structure of the feature model) • Understanding assets' evolution (observe the evolution of the realization of the features, and compare the refinement sets of different versions)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (developer)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Feature model / Product line artifacts and related information (trace information, trace set, artefacts created and the refinement links between them from domain and application engineering, the properties (name, type, identifier) of each link/artefact, time links related to a product, versions of artefacts which have evolved)
	Source of visualized items/data *(TQ3.1)*	A data base repository to store trace information (trace repository).
	Collection procedure/method of visualized items/data *(TQ3.2)*	Trace information must be recovered from certain information sources by trace extractors. A trace query provides means to perform specific (advanced) queries on a set of trace links and artefacts.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Network / Graph (bipartite graph) • Hierarchy (textual hierarchical representation, tree view, graphical hierarchical representation)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	The large nodes represent the artefacts and the small ones represent the links. Links stemming from a selected artefact are in light red, and the target artefacts of these links are pink colored (nodes in darker grey). Changed artefacts are placed in the center of a semi-circle with the related artefacts arranged in a semi-circle around it. The links that stem from this artefact and the artefacts that are target of these links are colored in red (dark grey). When comparing refinement sets, green nodes (or links) are common to both products.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (links and artefacts are selectable) • Browsing / Navigation (navigate) • Details on demand / Drill-down (user demand / when an item is selected, its properties (name, type, identifier) appear in the top part of the graph) • Details on demand / Labeling (all elements of the trace graph may be annotated with additional information) • Filtering / Highlighting/Mitigation (the color is propagated recursively to the targets of the targets) • Layout ("radial view")
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in an Integrated Development Environment (Eclipse IDE)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Eclipse IDE. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, it utilizes Prefuse and Ecore, a metamodel from Eclipse Modeling Framework (EMF) designed in MOF 2.0.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	N/A
	Application scenarios of the visualizations *(TQ7.1)*	N/A
	Evaluated aspects *(TQ7.2)*	N/A
	Visualization evaluation results/outcomes *(TQ 7.3)*	N/A

Table 52. Interface Descriptions Management System (IDMS) [Areeprayolkij2010208]

	Field	Information to be extracted
Publication metadata	**Title**	IDMS: A system to verify component interface completeness and compatibility for product integration
	Authors	Areeprayolkij, W., Limpiyakorn, Y., Gansawat, D.
	Publication date (year/month)	??, 2010
	Publication type	Article (Journal)
	Source	Communications in Computer and Information Science
	Volume and Edition (for journals)	v. 117 CCIS
	Place (for conferences)	N/A
	Pages	pp. 208-217
	Link (if applicable)	http://dx.doi.org/10.1007/978-3-642-17578-7_21
	Abstract	The growing approach of Component-Based software Development has had a great impact on today system architectural design. However, the design of subsystems that lacks interoperability and reusability can cause problems during product integration. At worst, this may result in project failure. In literature, it is suggested that the verification of interface descriptions and management of interface changes are factors essential to the success of product integration process. This paper thus presents an automation approach to facilitate reviewing component interfaces for completeness and compatibility. The Interface Descriptions Management System (IDMS) has been implemented to ease and fasten the interface review activities using UML component diagrams as input. The method of verifying interface compatibility is accomplished by traversing the component dependency graph called Component Compatibility Graph (CCG). CCG is the visualization of which each node represents a component, and each edge represents communications between associated components. Three case studies were studied to subjectively evaluate the correctness and usefulness of IDMS.
Visualization metadata	**Approach/tool name** *(PQ)*	Interface Descriptions Management System (IDMS)
	Screenshot	

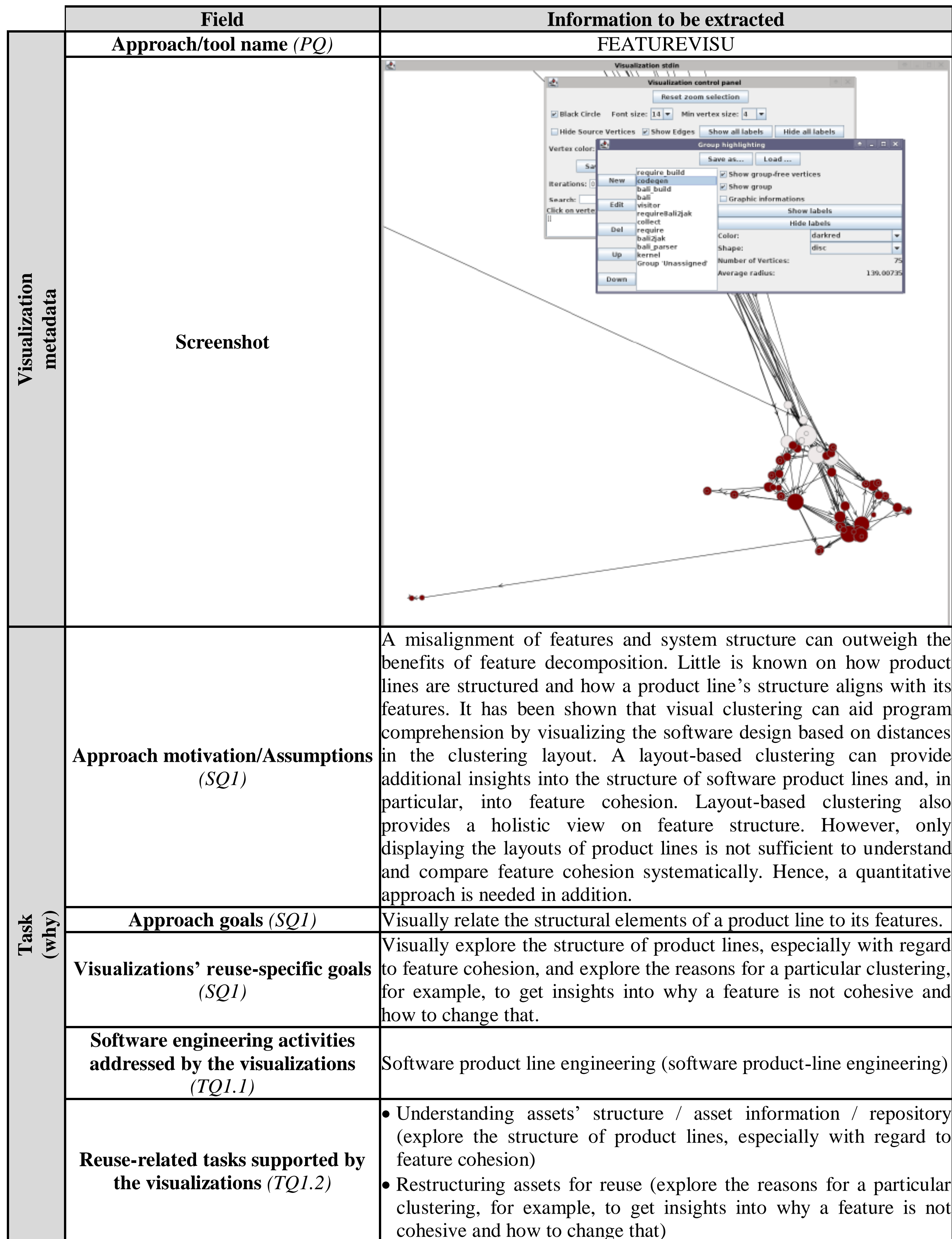
	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	The verification of interface descriptions and management of interface changes are factors essential to the success of product integration process. Project delay problems often occur in the component integration process. Many software applications encounter similar difficulties to effectively integrate the implemented component subsystems.
	Approach goals *(SQ1)*	Facilitate reviewing component interfaces for completeness and compatibility.
	Visualizations' reuse-specific goals *(SQ1)*	Verify interface compatibility and help clustering the components for ordering the sequences of integration plan.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (product integration)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Integrating reusable assets (verify interface compatibility and help clustering the components for ordering the sequences of integration plan)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	User (user)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Component / Asset and related information (component / communications between associated components)
	Source of visualized items/data *(TQ3.1)*	Input components.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The input components' interface descriptions and their compatibility with associated components are extracted automatically. The extracted interface descriptions can then be used in the static review process. Detailed format of UML component diagram is needed for the construction of the Component Compatibility Graph. The automation of the verification of component interface compatibility is then carried out by graph traversal. The next steps are: (1) Extraction of Interface Descriptions; (2) Construction of Component Dependency Graph; and (3) Verification of Interface Compatibility of Components Once the user has imported a component diagram in XML file format, the system read this file as input, and then XML tags and values are contained in the DOM tree table window.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graph)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Each node represents a component, and each edge represents communications between associated components.
	Visualization strategies and techniques *(TQ4.2)*	Clustering (cluster)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, it uses the UML component diagram, i.e. white box view, the DOMParser and Grappa.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (three case studies[39] were studied)
	Application scenarios of the visualizations *(TQ7.1)*	IDMS itself, a simple ordering product system, and Computed Tomography (CT) scan image visualization system. [commercial]
	Evaluated aspects *(TQ7.2)*	Subjective evaluation of the correctness and usefulness of IDMS
	Visualization evaluation results/outcomes *(TQ 7.3)*	The preliminary results were satisfactory to the architectural designer by examining the outputs of the system.

Table 53. FEATUREVISU [Apel2011421]

	Field	Information to be extracted
Publication metadata	**Title**	Feature cohesion in software product lines: An exploratory study
	Authors	Apel, S., Beyer, D.
	Publication date (year/month)	May, 2011
	Publication type	Conference
	Source	Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Honolulu, Hawaii
	Pages	pp. 421-430
	Link (if applicable)	http://dx.doi.org/10.1145/1985793.1985851
	Abstract	Software product lines gain momentum in research and industry. Many product-line approaches use features as a central abstraction mechanism. Feature-oriented software development aims at encapsulating features in cohesive units to support program comprehension, variability, and reuse. Surprisingly, not much is known about the characteristics of cohesion in feature-oriented product lines, although proper cohesion is of special interest in product-line engineering due to its focus on variability and reuse. To fill this gap, we conduct an exploratory study on forty software product lines of different sizes and domains. A distinguishing property of our approach is that we use both classic software measures and novel measures that are based on distances in clustering layouts, which can be used also for visual exploration of product-line architectures. This way, we can draw a holistic picture of feature cohesion. In our exploratory study, we found several interesting correlations (e.g., between development process and feature cohesion) and we discuss insights and perspectives of investigating feature cohesion (e.g., regarding feature interfaces and programming style).

[39] Section 4.8 discusses the adequacy of the “case study” denomination, based on the experimental software engineering literature.

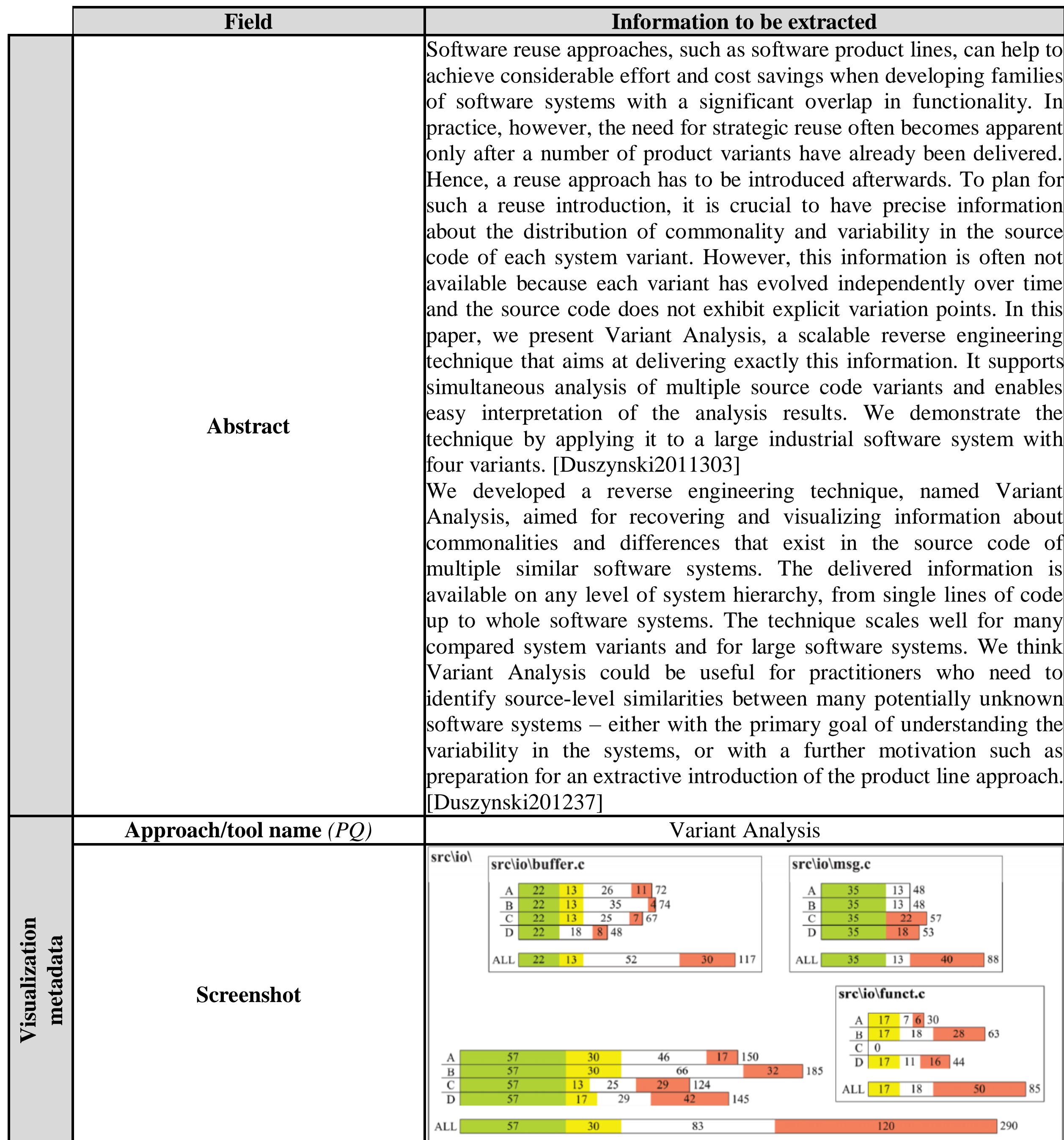
	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	FEATUREVISU
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	A misalignment of features and system structure can outweigh the benefits of feature decomposition. Little is known on how product lines are structured and how a product line's structure aligns with its features. It has been shown that visual clustering can aid program comprehension by visualizing the software design based on distances in the clustering layout. A layout-based clustering can provide additional insights into the structure of software product lines and, in particular, into feature cohesion. Layout-based clustering also provides a holistic view on feature structure. However, only displaying the layouts of product lines is not sufficient to understand and compare feature cohesion systematically. Hence, a quantitative approach is needed in addition.
	Approach goals *(SQ1)*	Visually relate the structural elements of a product line to its features.
	Visualizations' reuse-specific goals *(SQ1)*	Visually explore the structure of product lines, especially with regard to feature cohesion, and explore the reasons for a particular clustering, for example, to get insights into why a feature is not cohesive and how to change that.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software product line engineering (software product-line engineering)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' structure / asset information / repository (explore the structure of product lines, especially with regard to feature cohesion) • Restructuring assets for reuse (explore the reasons for a particular clustering, for example, to get insights into why a feature is not cohesive and how to change that)

	Field	Information to be extracted
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Developer / programmer (developers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Feature model / Product line artifacts and related information (elements of features and their dependency relation)
	Source of visualized items/data *(TQ3.1)*	A clustering layout (which provides information to build the dependency graph of a software product line, which is also an input) and a mapping between element nodes and features.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Feature-cohesion-based measures consider the number of references between program elements and their distances in a clustering layout. Classic measures and distance-based measures. For assessing feature structure, the number of dependencies between elements inside a feature (internal dependencies) can be related to the number of dependencies to elements outside that feature (external dependencies). Another option is to relate the number of dependencies inside a feature to the overall number of elements of that feature. Information obtained from a clustering layout is used to explore and assess feature cohesion. The distances computed by a layout-based clustering algorithm are used to assess feature structure in software product lines, complementary to classic indicators for structure. The tool receives as input the dependency graph of a software product line and a mapping between element nodes and features. The tool optimizes the layout of the dependency graph iteratively by grouping element nodes that depend on each other. The dependency graph spans a nontrivial network, in which many forces take effect simultaneously.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Network / Graph (graph)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Calls, usage, inheritance, etc. are depicted as edges in the graph. The software system is decomposed according to the dependencies in the graph. Distances between elements are presented in a two-dimensional space, in which related elements have close positions and unrelated elements have distant positions. A feature has a higher cohesion than coupling if its elements are close to each other, because then they are connected by many internal edges. If the elements that a feature introduces are scattered across the entire layout, then the cohesion of the feature is lower than its coupling to other features. The discs (nodes of the graph) represent the fields and methods of the system. The area of a disc for a node is proportional to the node's edge degree. If the discs form clusters, then the corresponding fields/methods heavily depend on each other. Initially, the color of all nodes is light gray.
	Visualization strategies and techniques *(TQ4.2)*	• Details on demand / Labeling / Tooltip (tool tips) • Clustering (layout based clustering (a.k.a. visual clustering) / cohesive elements are drawn closely together in such layouts, long-distance references do not witness cohesion / highly connected nodes shall be in the same cluster) • Filtering / Tuning/Tweaking (node coloring and displaying edges) • Layout (force-directed graph drawing) • Zooming / Geometric (zooming) • Panning / Drag-and-drop (drag & drop)

	Field	Information to be extracted
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	CCVISU, a visual-clustering tool, was extended.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (exploratory study)
	Application scenarios of the visualizations *(TQ7.1)*	Forty sample software product lines of different sizes and domains, developed by refactoring or from scratch, all made available on the web. [academic] [open source]
	Evaluated aspects *(TQ7.2)*	Differences between individual features and individual product lines with regard to feature cohesion
	Visualization evaluation results/outcomes *(TQ 7.3)*	(1) Considerable differences were found between individual features and entire product lines (effectively, covering the entire spectrum of possible values of the measures). (2) There is room for refactoring features into smaller pieces. (3) Distance-based measures draw a similar picture as their classic counterparts (they correlate strongly). But in certain cases, they provide more information than the classic measures. (4) There are correlations between feature cohesion, and feature and system size. (5) The features of product lines developed by refactoring have significantly higher cohesion values (for all measures) than the features of product lines developed from scratch.

Table 54. Variant Analysis [Duszynski2011303 / Duszynski201237]

	Field	Information to be extracted
Publication metadata	**Title**	Analyzing the source code of multiple software variants for reuse potential [Duszynski2011303] Recovering variability information from the source code of similar software products [Duszynski201237]
	Authors	Duszynski, S., Knodel, J., Becker, M. [Duszynski2011303] Duszynski, S., Becker, M. [Duszynski201237]
	Publication date (year/month)	October, 2011 [Duszynski2011303] June, 2012 [Duszynski201237]
	Publication type	Conference [Duszynski2011303] Conference [Duszynski201237]
	Source	Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011) [Duszynski2011303] Proceedings of the 3rd International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2012) [Duszynski201237]
	Volume and Edition (for journals)	N/A [Duszynski2011303] N/A [Duszynski201237]
	Place (for conferences)	Limerick, Ireland [Duszynski2011303] Zürich, Switzerland [Duszynski201237]
	Pages	pp. 303-307 [Duszynski2011303] pp. 37-40 [Duszynski201237]
	Link (if applicable)	http://dx.doi.org/10.1109/WCRE.2011.44 [Duszynski2011303] http://dx.doi.org/10.1109/PLEASE.2012.6229768 [Duszynski201237]

	Field	Information to be extracted
	Abstract	Software reuse approaches, such as software product lines, can help to achieve considerable effort and cost savings when developing families of software systems with a significant overlap in functionality. In practice, however, the need for strategic reuse often becomes apparent only after a number of product variants have already been delivered. Hence, a reuse approach has to be introduced afterwards. To plan for such a reuse introduction, it is crucial to have precise information about the distribution of commonality and variability in the source code of each system variant. However, this information is often not available because each variant has evolved independently over time and the source code does not exhibit explicit variation points. In this paper, we present Variant Analysis, a scalable reverse engineering technique that aims at delivering exactly this information. It supports simultaneous analysis of multiple source code variants and enables easy interpretation of the analysis results. We demonstrate the technique by applying it to a large industrial software system with four variants. [Duszynski2011303] We developed a reverse engineering technique, named Variant Analysis, aimed for recovering and visualizing information about commonalities and differences that exist in the source code of multiple similar software systems. The delivered information is available on any level of system hierarchy, from single lines of code up to whole software systems. The technique scales well for many compared system variants and for large software systems. We think Variant Analysis could be useful for practitioners who need to identify source-level similarities between many potentially unknown software systems – either with the primary goal of understanding the variability in the systems, or with a further motivation such as preparation for an extractive introduction of the product line approach. [Duszynski201237]
Visualization metadata	**Approach/tool name** *(PQ)*	Variant Analysis
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	An introduction of a systematic reuse approach is an appealing idea, as improvements in maintenance effort and product quality can be expected. But the need for strategic reuse often becomes apparent only after a number of product variants have already been delivered, i.e., proactive planning of reuse happens rather seldom. Instead of developing reusable components with defined variation points, reuse opportunities are only explored after multiple variants have been developed and have to be evolved in parallel. Hence, a reuse approach has to be introduced afterwards. However, turning multiple similar but slightly different implementations into a single code base composed of reusable, generic components is not trivial – especially if detailed information on common and variable code parts has not been tracked during the parallel evolution of the products and is therefore lost. Thus, to plan for such a reuse introduction, it is crucial to have precise information about the distribution of commonality and variability in the source code of each system variant. However, this information is often not available because each variant has evolved independently over time and the source code does not exhibit explicit variation points. The comparison of many similar systems is usually performed pair-wise, with each of the variants compared to each other. However, presenting comparison results of three or more variants in a pair-wise way hides important information, such as the size of common parts shared by all analyzed variants.
	Approach goals *(SQ1)*	Recover and visualize information about commonalities and differences that exist in the source code of multiple similar software systems (delivering quantitative information about similarity across system variants) for identifying system parts suitable for transformation into reusable assets and planning necessary implementation steps (i.e., supporting the reuse potential assessment and the migration to systematic software reuse), besides providing an overview of commonality distribution in the whole analyzed system family, allowing for detailed goal-driven refinement of the analysis results.
	Visualizations' reuse-specific goals *(SQ1)*	Deliver precise quantitative information about the similarity across the analyzed system variants through an abstracted result presentation in order to assess reuse potential.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software maintenance (maintenance) • Software product line engineering (product line development / migration towards a product line / preparation for an extractive introduction of the product line approach)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Discovering and evaluating potentially reusable assets (deliver quantitative information about the similarity across the analyzed system variants in order to assess reuse potential)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	User (users)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	Source code and related information (lines of text in source code files / commonalities and variabilities in the source code of multiple software systems)
	Source of visualized items/data *(TQ3.1)*	Compilation Unit objects.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Occurrence matrices are used for organizing variability information. An Occurrence Matrix is created for each Compilation Unit object. N+1 bars are created, one for each occurrence matrix. They are constructed as follows: (1) Each variant is represented as a set of distinct atomic elements. (2) A matrix is created for each variant: the rows of the matrix represent the atomic elements of the variant, and the columns represent all the analyzed variants. (3) A union matrix is created for the union of all sets. Its rows represent all the elements existing in any of the sets. (4) Each matrix cell has a value of "1" if the element represented by the field's row belongs to the variant represented by the field's column, or a value of "0" if not. (5) Each matrix has an additional summary column, which counts the number of variants the given element belongs to.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Geometric forms (bar diagrams)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Subsystems represent directories and the Compilation Units represent code files. Code Elements correspond to text lines. The length of each bar equals the number of rows in the respective matrix. Each bar is divided into three parts, representing core, shared, and unique Code Elements in each matrix, with lengths equal to the number of respective elements. The calculation result can be visualized as a colored bar in the bar diagram and by coloring the single elements in the detailed result visualization.
	Visualization strategies and techniques *(TQ4.2)*	Clustering (hierarchical aggregation)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment (assumed)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: Eclipse IDE. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, based on Eclipse.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	1) Practical use [probably by the authors themselves] 2) Practical use [probably by the authors themselves] ([in both:] we applied the Variant Analysis technique)
	Application scenarios of the visualizations *(TQ7.1)*	1) Applied to four variants of a C++ software system developed by an industrial customer. Performance and scalability results were measured on the following configuration: Intel Core2 Duo 2.53 GHz, 3 GB RAM, Win XP 32bit. 2) Applied to the source code folder (/usr/src/) of four large software systems from the BSD Unix family. To estimate the relative similarity of NetBSD 1.0 to the other analyzed variants, a subset calculation was performed. [commercial] [open source]
	Evaluated aspects *(TQ7.2)*	1) Performance, response times, and scalability. 2) Relative similarity of NetBSD 1.0 to the other analyzed variants.
	Visualization evaluation results/outcomes *(TQ 7.3)*	1) Performance, response times, and scalability of the solution are good. 2) OpenBSD 2.0 is the most similar, and BSD 4.4 lite is the least similar to NetBSD 1.0. The calculation result was computed in 1.1 second.

Table 55. API-Dependence Visualization [Bauer2012435]

	Field	Information to be extracted
Publication metadata	**Title**	Understanding API usage to support informed decision making in software maintenance
	Authors	Bauer, V., Heinemann, L.
	Publication date (year/month)	March, 2012
	Publication type	Conference
	Source	Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Szeged, Hungary
	Pages	pp. 435-440
	Link (if applicable)	http://dx.doi.org/10.1109/CSMR.2012.55
	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.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	API-Dependence Visualization[40]
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Reuse of third-party libraries promises significant productivity improvements in software development. Informed decision making therefore requires a thorough understanding of the extent and nature of dependencies on external APIs. 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.
	Approach goals *(SQ1)*	Analyze the dependencies of software projects on external APIs, enabling quick insight into how external libraries are used by a project and how complex the dependencies are, besides aiding in decision making regarding library migration scenarios and determining the degree of dependence to its included libraries.
	Visualizations' reuse-specific goals *(SQ1)*	Gain a quick overview of the library dependencies and understand to which extent a package is dependent on APIs and also how the dependencies of a certain API span over the system architecture.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software maintenance (software maintenance)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' structure / asset information / repository (gain a quick overview of the library dependencies / understand to which extent a package is dependent on APIs and how the dependencies of a certain API span over the system architecture)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	Maintainer (maintainers)

[40] This is not an official name, but one of the screenshots refers to the tool by using this name.

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (information about library usage) • Software project and related information (hierarchical composition of the software project)
	Source of visualized items/data *(TQ3.1)*	From the source code of a project.
	Collection procedure/method of visualized items/data *(TQ3.2)*	The source code of a project is statically analyzed to determine the dependencies and use the extracted information to produce a visualization. To determine the degree of API dependence and complexity, the total number of all method calls to external APIs is determined. Secondly, the number of distinct method calls is extracted for each external API. Thirdly, the visualization encodes the proportion of each distinct method call with respect to all method calls. The abstract syntax tree (AST) for Java code is obtained and traversed to extract the API references of the source code of software projects. The degree of dependence to an API approximated with the number of API method calls. For each class, the total number of API calls, the number of distinct API methods called for each included library as well as their proportion are determined, and the data hierarchically along the package structure are aggregated. Every Java Archive File (JAR) contained in the project is considered as included library.
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Hierarchy (tree table) • Matrix / Matrix-like (interactive table) • Geometric forms (colored bars)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Characteristics of source code are mapped into colored bars. The columns [of the output table with the analysis results] 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 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.
	Visualization strategies and techniques *(TQ4.2)*	• Details on demand / Drill-down (expanding the tree reveals how packages of the system use the APIs / drill-down) • Filtering / Collapse/Expand (expanding the tree)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, in a web environment (HTML output)
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse (assumed)
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	Implemented in Java, it uses the Eclipse Java Compiler and ConQAT, an open source software quality assessment toolkit.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [probably by the authors themselves] (qualitatively evaluate by answering questions typically raised in the use case scenarios)
	Application scenarios of the visualizations *(TQ7.1)*	Three open source Java projects, which use external libraries. [open source]
	Evaluated aspects *(TQ7.2)*	Not specified
	Visualization evaluation results/outcomes *(TQ 7.3)*	The evaluation shows that the central questions raised during the identified usage scenarios can be answered by the approach.

Table 56. FeatureCommander [Feigenspan20121]

	Field	Information to be extracted
Publication metadata	**Title**	Do background colors improve program comprehension in the #ifdef hell?
	Authors	Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G.
	Publication date (year/month)	??, 2013[41]
	Publication type	Article (Journal)
	Source	Empirical Software Engineering
	Volume and Edition (for journals)	v. 18, n. 4
	Place (for conferences)	N/A
	Pages	pp. 699-745
	Link (if applicable)	http://dx.doi.org/10.1007/s10664-012-9208-x
	Abstract	Software-product-line engineering aims at the development of variable and reusable software systems. In practice, software product lines are often implemented with preprocessors. Preprocessor directives are easy to use, and many mature tools are available for practitioners. However, preprocessor directives have been heavily criticized in academia and even referred to as “#ifdef hell”, because they introduce threats to program comprehension and correctness. There are many voices that suggest to use other implementation techniques instead, but these voices ignore the fact that a transition from preprocessors to other languages and tools is tedious, erroneous, and expensive in practice. Instead, we and others propose to increase the readability of preprocessor directives by using background colors to highlight source code annotated with ifdef directives. In three controlled experiments with over 70 subjects in total, we evaluate whether and how background colors improve program comprehension in preprocessor-based implementations. Our results demonstrate that background colors have the potential to improve program comprehension, independently of size and programming language of the underlying product. Additionally, we found that subjects generally favor background colors. We integrate these and other findings in a tool called FeatureCommander, which facilitates program comprehension in practice and which can serve as a basis for further research.

[41] Although it is cited as 2013, it was retrieved from the search engine in 2012 when it was accepted for publication.

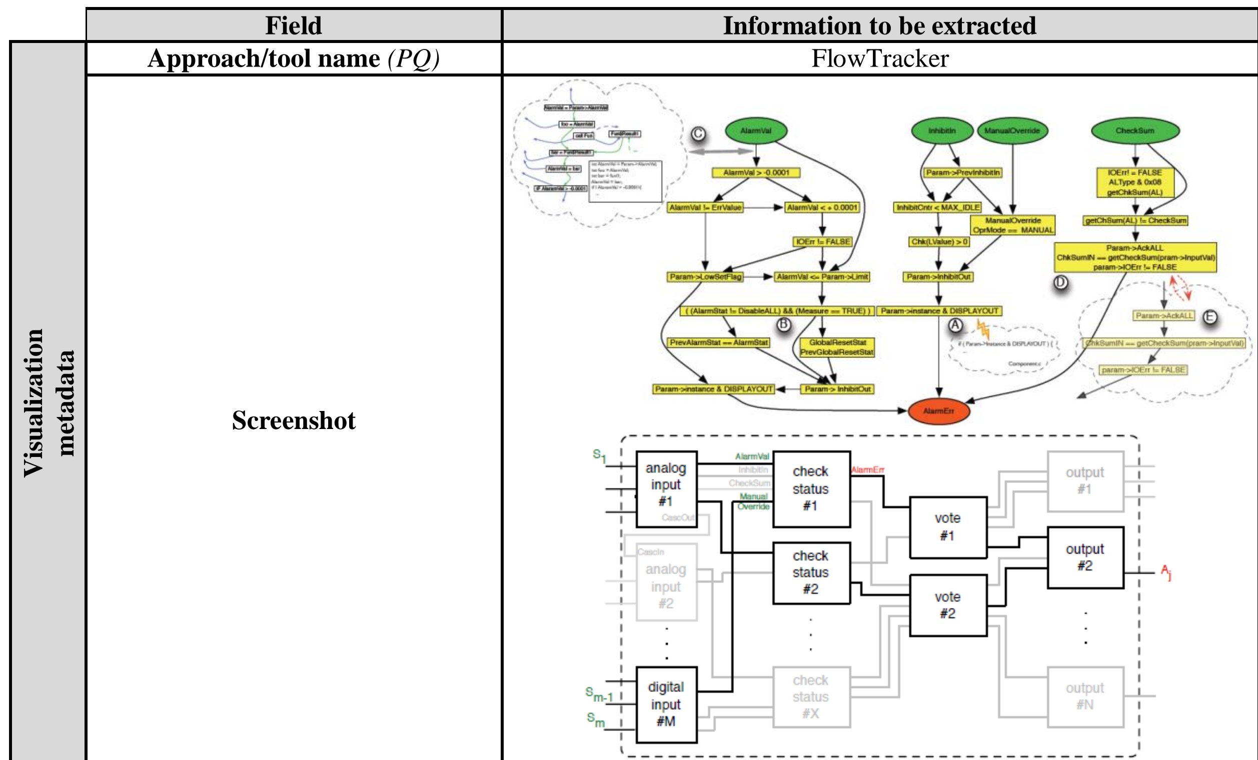
	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	FeatureCommander
	Screenshot	
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	In practice, companies implement SPLs mostly with conditional compilation using preprocessor directives, which are used to annotate feature code and are removed before compilation. However, the flexibility and expressiveness can lead to complex and obfuscated code that is inherently difficult to understand and can lead to high maintenance costs. Hence, preprocessor usage potentially threatens program comprehension. It is imperative to consider comprehensibility of source code, because understanding is a crucial part in maintenance: Maintenance programmers spend most of their time with understanding code. By ensuring easy-to-understand source code, software development costs can be reduced. So far, little is known about the influence of background colors on program comprehension used in source-code editors.
	Approach goals *(SQ1)*	Allow a programmer to identify feature code at first sight and distinguish code of different features.
	Visualizations' reuse-specific goals *(SQ1)*	Help distinguish feature code from base code.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	• Software maintenance (software maintenance) • Software product line engineering (software-product-line engineering)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	Understanding assets' structure / asset information / repository (help distinguish feature code from base code)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer with reuse (programmer / developer) • Maintainer (maintenance programmers)
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Source code and related information (feature code, percentage of each feature that occurs in the represented file or folder) • Feature model / Product line artifacts and related information (feature model)
	Source of visualized items/data *(TQ3.1)*	Structures are extracted from the source code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	N/A

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	Geometric forms (bars)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Horizontal bars for each folder and file indicate whether and how much feature code a folder or file contains. Features are visualized as bars, ordered by the nesting hierarchy. There is a default setting, in which two shades of gray are assigned to features. Code of features located nearby in the source-code file has a different shade of gray, such that a developer can distinguish them. Colors can be then mapped to features.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (clicking them immediately shows the according code fragment) • Details on demand / Drill-down (clicking them immediately shows the according code fragment) • Filtering / Highlighting/Mitigation (highlighting technique that supports users in finding relevant information / feature code is displayed with a background color that distinguishes feature code from code of other features and base code / consistent usage of colors throughout all visualizations / the automatic color assignment chooses colors such that they are as different as possible in the hue value of the HSV color model / if a code fragment is assigned to multiple features, only the background color of the innermost feature is shown)) • Filtering / Tuning/Tweaking (users can assign colors to features / users can automatically assign a palette of colors to multiple features / users can adjust the opacity of the background color) • Filtering / Collapse/Expand (overview / features that are currently not of interest can be collapsed) • Overview + detail (overview / features that are currently not of interest can be collapsed) • Hierarchical visualization (hierarchy of features / tree representations) • Sorting (two tree representations of the project ordered according to the file structure, the other ordered by features) • Presentation / Simultaneous (multiple visualizations / two tree representations of the project)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: N/A • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The CIDE tool.

	Field	Information to be extracted
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	1) Experiment 2) Experiment 3) Experiment
	Application scenarios of the visualizations *(TQ7.1)*	1) 52 students from the University of Passau, with a medium-sized Java-based SPL with four optional features (computers with Linux, 19'' TFT screens). 2) Students at the University of Magdeburg (computers with Windows XP, 17'' TFT screens). 3) 9 master's and 5 PhD students from the University of Magdeburg, with a a large real-time extension for Linux implemented in C (computers with Windows XP, 17'' TFT screens). [academic] [open source]
	Evaluated aspects *(TQ7.2)*	1) The effect of background colors on program comprehension in preprocessor-based SPLs compared to ifdef directives, based on correctness of answers, response time and opinion of subjects. 2) Whether developers would switch between background colors and ifdef directives, based on performance (= how subjects switched between the annotation styles). 3) Scalability of colors, based on response times, correctness of tasks and opinion of subjects.
	Visualization evaluation results/outcomes *(TQ 7.3)*	It cannot be stated that background colors are always helpful in every situation in which preprocessors are used to implement variability. 1) For locating feature code, background colors significantly speed up the comprehension process (probably due to the preattentive color perception, compared to attentive text perception), but unsuitable background colors can slow down program comprehension. Subjects of the color group have to look only for a color, not read text to solve tasks. Colors can also negatively affect program comprehension if not chosen carefully (i.e., if they are too bright and saturated). 2) Subjects preferred background colors, even if they slow them down, and did not necessarily recognize the disturbing effect of the background color. 3) There was an improvement (in some static tasks) of program comprehension for locating feature code when using background colors. In large SPLs, background colors have a potentially positive impact on program comprehension in preprocessor-based SPLs in terms of locating feature code.

Table 57. FlowTracker [Yazdanshenas2012143]

	Field	Information to be extracted
Publication metadata	Title	Tracking and visualizing information flow in component-based systems
	Authors	Yazdanshenas, A. R., Moonen, L.
	Publication date (year/month)	June, 2012
	Publication type	Conference
	Source	Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC 2012)
	Volume and Edition (for journals)	N/A
	Place (for conferences)	Passau, Germany
	Pages	pp. 143-152
	Link (if applicable)	http://dx.doi.org/10.1109/ICPC.2012.6240482
	Abstract	Component-based software engineering is aimed at managing the complexity of large-scale software development by composing systems from reusable parts. In order to understand or validate the behavior of a given system, one needs to acquire understanding of the components involved in combination with understanding how these components are instantiated, initialized and interconnected in the particular system. In practice, this task is often hindered by the heterogeneous nature of source and configuration artifacts and there is little to no tool support to help software engineers with such a system-wide analysis. This paper contributes a method to track and visualize information flow in a component-based system at various levels of abstraction. We propose a hierarchy of 5 interconnected views to support the comprehension needs of both safety domain experts and developers from our industrial partner. We discuss the implementation of our approach in a prototype tool, and present an initial qualitative evaluation of the effectiveness and usability of the proposed views for software development and software certification. The prototype was already found to be very useful and a number of directions for further improvement were suggested. We conclude by discussing these improvements and lessons learned.

	Field	Information to be extracted
Visualization metadata	**Approach/tool name** *(PQ)*	FlowTracker
	Screenshot	

	Field	Information to be extracted
Task (why)	**Approach motivation/Assumptions** *(SQ1)*	Various studies have shown that program comprehension accounts for a significant part of the development and maintenance efforts and with today's rapid growth in system size and complexity, software engineers are faced with tremendous comprehension challenges. Even though component-based design supports comprehension by lowering coupling and increasing the cohesion of components, the overall comprehension of component-based systems can be prohibitively complicated. In order to understand a system's behavior, one needs to understand how control and data flow are interlaced through its combination of component and configuration artifacts. However, there is little support for system-wide analysis of component-based systems. There is extensive literature on the visualization of non-source artifacts to support domain experts, but there is considerably less information on the visualization of source code related information for non-developers. Safety domain experts need to see the system's source artifacts represented in a context that is relevant to them – not just what the code does, but what it means. Dynamic analysis (tracing) during real-life operation is not an option due to safety hazards. Thus, any reverse engineered views on the system need to be goal-driven, at a suitable level of abstraction, and based on relevant knowledge of the application domain. Dependence graphs, and slices through dependence graphs, are complex, often even more complex than the original source artifacts. These models reflect all relevant program points and dependencies from a compiler's perspective, which is an intrinsic characteristic that makes them well-suited for detailed program analysis, but it makes them less suited for directly supporting comprehension or visualization.
	Approach goals *(SQ1)*	Track and visualize information flow in a component-based system at various levels of abstraction, provide source-based evidence that signals from the system's sensors trigger the appropriate actuators, and provide source-based evidence to support software certification.
	Visualizations' reuse-specific goals *(SQ1)*	Improve the comprehensibility of configuration and composition of the components, by understanding how control and data flow are interlaced through its combination of component and configuration artifacts.
	Software engineering activities addressed by the visualizations *(TQ1.1)*	Software development with reuse (component-based software engineering)
	Reuse-related tasks supported by the visualizations *(TQ1.2)*	• Understanding assets' behavior (understand how control and data flow are interlaced through its combination of component and configuration artifacts / improve the comprehensibility) • Integrating reusable assets (configuration and composition of the components)
Audience (who)	**Visualizations' audience (stakeholders who can benefit from the visualizations)** *(SQ2)*	• Developer / programmer (developers) • Others / non-related to software development (non-developer safety domain experts)

	Field	Information to be extracted
Target (what)	**Visualized items/data (what is visualized)** *(SQ3)*	• Component / Asset and related information (implementation artifacts of component-based systems / actuator and all related sensors, component instances, and inter-component connections, dependencies between a component's input and output ports, dependencies between all system-level inputs (sensors) and outputs (actuators), intercomponent information flow from all sensors that can affect a given actuator, intra-component information flow from all input ports that can affect that output port, all conditions that control the information flow towards the selected output port)[42] • Source code and related information (pretty-printed source code)
	Source of visualized items/data *(TQ3.1)*	Component source code.
	Collection procedure/method of visualized items/data *(TQ3.2)*	Program slicing is used to leave out all parts of the program that are not relevant to a given point of interest. For each component in the system, a component dependence graph (CDG) is built by following the method for constructing inter-procedural dependence graphs and taking the component source code as "system source". The system's configuration artifacts are analyzed to build an inter-component dependence graph (ICDG). This graph captures the externally visible interfaces and interconnections of the component instances. The system-wide dependence graph (SDG) is constructed by integrating the system's ICDG with the CDGs for the individual components. The ICDG is taken and each "component instance node" is substituted with a sub-graph formed by the CDG for the given component. Views are constructed from the system-wide dependence model via a combination of slicing, transformation and visualization.

[42] The components are implemented in MISRA C [Yazdanshenas2012143].

	Field	Information to be extracted
Representation (how)	**Visualization metaphors used (how it is visualized)** *(SQ4)*	• Network / Graph (graph) • Diagrams ([box and line / data flow] diagram) • Matrix / Matrix-like (matrix)
	Data-to-visualization mapping (input/output) *(TQ4.1)*	Dependencies between all system-level inputs (sensors) and outputs (actuators) are shown in one single matrix, with sensors and actuators are represented as rows and columns, respectively. A filled cell of such matrix indicates that there is at least one path along which information can flow from that sensor to that actuator. When analyzing component dependencies, there is one dependency matrix for each component. Input and output ports are represented as rows and columns, respectively, and the dependencies between a component's input and output ports are represented by using filled cells. For analyzing the information flow of a given component, there is a diagram for each output port of the component.
	Visualization strategies and techniques *(TQ4.2)*	• Selection (a user can click on a component instance to zoom in on a single component, or click outside the diagram to return to a higher level of abstraction) • Browsing / Navigation (hyperlinks to enable easy navigation / hypertext navigation facilities, e.g. cross-referencing of program entities with their definition) • Details on demand / Drill-down (embedding hyperlinks to corresponding views on the next abstraction level / a user can click on a component instance to zoom in on a single component, or click outside the diagram to return to a higher level of abstraction / higher level views provide links into the source code) • Details on demand / Labeling (aggregate node is labelled based on the conditions it represents) • Clustering (shows which input ports can affect which output ports but hides all details on how the information flow is realized / combine sequences of conditions into aggregated conditions wherever possible to reduce cognitive overhead) • Filtering / Highlighting/Mitigation (highlights) • Filtering / Collapse/Expand (collapsable subgraphs to represent conditional clusters and their aggregate representation) • Hierarchical visualization (hierarchical views / hierarchy of views that represent system-wide information flows at various levels of abstraction) • Presentation (hierarchy of views that represent system-wide information flows at various levels of abstraction)
Medium (where)	**Device and/or environment used for displaying the visualizations (where it is visualized)** *(SQ5)*	Computer screen, standalone or in the own environment
	Resources used for interacting with the visualizations *(TQ5.1)*	Mouse

	Field	Information to be extracted
Requirements (which)	**Hardware and software requirements/dependencies** *(SQ6)*	• SW: The work builds on an earlier tool for reverse engineering a fine-grained system-wide model of the control and data dependencies in the system from source artifacts, creating system-wide dependence graphs (SDGs). The aiSee graph layout software is also necessary. • HW: N/A
	Programming languages, APIs and frameworks used for building the visualization *(TQ6.1)*	The following frameworks are used: CodeSurfer and its API, the OMG Knowledge Discovery Metamodel (KDM) and its API, a Java Native Interface (to drive KDM constructors in the Eclipse Modeling Framework), Xalan-J, a simple slicing tool in Java (created as part of earlier work), HTML and Doxygen.
Evidence (worthwhile)	**Visualization evaluation methods** *(SQ7)*	Practical use [by others] (preliminary qualitative study, exploratory, followed by a structured interview which was guided by a questionnaire)
	Application scenarios of the visualizations *(TQ7.1)*	Performed by a group of six subjects (before the tool can be adopted by an industry partner), being three senior engineers in Kongsberg Maritime (KM) and three colleagues in the final stages of their PhD studies at Simula Research Laboratory. [commercial] [academic]
	Evaluated aspects *(TQ7.2)*	Effectiveness and usability of the proposed views for software development and software certification, their fitness for the needs of an industrial partner, and other tasks where FlowTracker could be helpful.
	Visualization evaluation results/outcomes *(TQ 7.3)*	(1) System Dependence Survey: Subjects indicated that they found its presentation of information to be intuitive, and that the goal of summarizing system-wide information flow was adequately achieved. (2) System Information Flow: Subjects were generally satisfied with its functionality; two subjects had some reservations with respect to the amount of information shown; the way information is presented was received as intuitive. (3) Component Dependence Survey: the subjects agreed that it adequately summarizes the dependencies between input and output terminals. (4) Component Information Flow: Five of the subjects agreed that conditions can have a significant effect on the intra-component information flows and should be highlighted and put in perspective to improve comprehension. A subject stated that one might need to see the assignment statements in the diagram as well to understand the information flows, and would like to see the outgoing edges of condition nodes labelled to indicate which edge would be used if the condition would be evaluated during actual execution. There were concerns about the intuitiveness of the diagrams when they grow in size. The subjects would like to see more interactive facilities, especially some measures to better deal with the larger diagrams. (5) Implementation View: Subjects reported that it helped them to relate more easily to higher level views since it "helps to remove the gap between visualizations and the source code". They considered the hyperlinks from conditions in the Component Information Flow diagram to the respective locations in the source code beneficial for comprehension and traceability. (6) Overall Experience: Subjects were positive about the intuitiveness of the tool, but would like to see it closer integrated into their IDEs. (7) Other tasks where FlowTracker could be helpful include source code maintenance, track ripple effects of modified source code, track ripple effects of modified configuration files, configuring a new system, debug individual modules, auditing projects, and training new project members.

Appendix C – Visualization strategies and techniques

Table 58 relates the visualization strategies and techniques to the publications/approaches in which they were identified (*TQ4.2*). The strategies and techniques are organized as a Visualization Feature Model [Vasconcelos et al. 2014].

Table 58. Visualization strategies and techniques by publication/approach (*TQ4.2*)

Visualization strategy/technique	# of approaches	Approaches
Selection	16	[Constantopoulos19951] [Lange1995342] [Alonso1998483] [Biddle199992 / Marshall2001 / Marshall2001103] [Ye2000266] [Charters2002765] [Wahid2004414] [Gonçalves2007872 / Oliveira2007461] [Holmes2007100] [Stollberg2007236] [Dietrich200891] [Ali200950] [DeBoer200951] [Anquetil2010427] [Feigenspan20121] [Yazdanshenas2012143]
Browsing / Navigation	14	[Mancoridis199374] [Constantopoulos19951] [Lange1995342] [Alonso1998483] [Ye2000266] [Marshall2001103 / Anslow2004] [Charters2002765] [Wahid2004414] [McGavin2006153] [Gonçalves2007872 / Oliveira2007461] [Holmes2007100] [Stollberg2007236] [Anquetil2010427] [Yazdanshenas2012143]
Browsing / Querying	3	[Constantopoulos19951] [Ye2000266] [Kelleher200550]

Visualization strategy/technique	# of approaches	Approaches
Details on demand / Drill-down	13	[Mancoridis199374] [Alonso1998483] [Ye2000266] [Mittermeir200195] [Charters2002765] [Wahid2004414] [McGavin2006153] [Holmes2007100] [Stollberg2007236] [Anquetil2010427] [Bauer2012435] [Feigenspan20121] [Yazdanshenas2012143]
Details on demand / Labeling	9	[Mancoridis199374] [Constantopoulos19951] [Ye2000266] [Holmes2007100] [Stollberg2007236] [Dietrich200891] [López20091198] [Anquetil2010427] [Yazdanshenas2012143]
Details on demand / Labeling / Tooltip	2	[DeBoer200951] [Apel2011421]
Clustering	12	[Helfman199631] [Ye2000266] [Charters2002765] [Kelleher200550] [Tangsripairoj2006283] [Gonçalves2007872 / Oliveira2007461] [Dietrich200891] [Damaeviius2009507] [Areeprayolkij2010208] [Apel2011421] [Duszynski2011303 / Duszynski201237] [Yazdanshenas2012143]
Filtering	7	[Constantopoulos19951] [Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Wahid2004414] [Kelleher200550] [Holmes2007100] [Ali200950]

Visualization strategy/technique	# of approaches	Approaches
Filtering / Highlighting/Mitigation	13	[Mancoridis199374] [Alonso1998483] [Ye2000266] [Marshall2001103 / Anslow2004] [Mittermeir200195] [Gonçalves2007872 / Oliveira2007461] [Dietrich200891] [Ali200950] [DeBoer200951] [López20091198] [Anquetil2010427] [Feigenspan20121] [Yazdanshenas2012143]
Filtering / Tuning/Tweaking	8	[Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [Marshall2001103 / Anslow2004] [McGavin2006153] [Gonçalves2007872 / Oliveira2007461] [Apel2011421] [Feigenspan20121]
Filtering / Inclusion/Removal	7	[Mancoridis199374] [Marshall2001103] [Anslow2004 / Marshall200435] [McGavin2006153] [Gonçalves2007872 / Oliveira2007461] [Stollberg2007236] [Dietrich200891]
Filtering / Collapse/Expand	7	[Lange1995342] [Biddle199992] [Marshall2001103 / Anslow2004] [Holmes2007100] [Bauer2012435] [Feigenspan20121] [Yazdanshenas2012143]

Visualization strategy/technique	# of approaches	Approaches
Overview + detail	9	[Mancoridis199374] [Helfman199631] [Ye2000266] [Charters2002765] [McGavin2006153] [Tangsripairoj2006283] [Stollberg2007236] [Ali200950] [Feigenspan20121]
Layout	8	[Biddle199992] [Mittermeir200195] [Charters2002765] [Stollberg2007236] [Dietrich200891] [Ali200950] [Anquetil2010427] [Apel2011421]
Layout / 3D	2	[Washizaki20061222] [Ali200950]
Zooming / Geometric	6	[Marshall2001103 / Anslow2004] [Washizaki20061222] [Stollberg2007236] [Dietrich200891] [Ali200950] [Apel2011421]
Zooming / Semantic	1	[McGavin2006153]
Panning	1	[Dietrich200891]
Panning / Drag-and-drop	6	[Alonso1998483] [Marshall2001103] [Stollberg2007236] [Dietrich200891] [DeBoer200951] [Apel2011421]
Hierarchical visualization	4	[Tangsripairoj2006283] [Stollberg2007236] [Feigenspan20121] [Yazdanshenas2012143]
Animation	5	[Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [Mittermeir200195] [Dietrich200891]

Visualization strategy/technique	# of approaches	Approaches
Sorting	4	[Helfman199631] [Biddle199992] [Kelleher200550] [Feigenspan20121]
Rotating	3	[Washizaki20061222] [Stollberg2007236] [Ali200950]
Presentation	3	[Mittermeir200195] [Ali200950] [Yazdanshenas2012143]
Presentation / Simultaneous	8	[Lange1995342] [Biddle199992] [Biddle199992 / Marshall2001 / Marshall2001103] [Marshall2001103] [McGavin2006153] [Holmes2007100] [DeBoer200951] [Feigenspan20121]
Overlap / Flipping	3	[Alonso1998483] [McGavin2006153] [Ali200950]
Overlap / Transparency	1	[Biddle199992]
Linking	3	[Biddle199992 / Marshall2001 / Marshall2001103] [Ali200950] [DeBoer200951]
Focus + context	2	[Alonso1998483] [Holmes2007100]
N/A	1	[Marshall200381]