Software Visualization in 3D - Implementation, Evaluation, and Applicability

Von der Wirtschaftswissenschaftlichen Fakultät

der Universität Leipzig

genehmigte

DISSERTATION

zur Erlangung des akademischen Grades

Doctor rerum politicarum

Dr. rer. pol.

vorgelegt

von Dipl.-Wirtsch.-Inf. Richard Müller

geboren am 23. September 1983 in Dresden

Gutachter: Prof. Dr. U. W. Eisenecker
Prof. Dr. B. Franczyk

Tag der Verleihung: 15. April 2015

Bibliographic Description

Müller, Richard
Software Visualization in 3D – Implementation, Evaluation, and Applicability
Leipzig University, Dissertation
126 Pages, 104 References, 25 Figures, 7 Tables, 8 Listings, 2 Appendices

Abstract

The focus of this thesis is on the implementation, the evaluation and the useful application of the third dimension in software visualization. Software engineering is characterized by a complex interplay of different stakeholders that produce and use several artifacts. Software visualization is used as one mean to address this increasing complexity. It provides role- and task-specific views of artifacts that contain information about structure, behavior, and evolution of a software system in its entirety. The main potential of the third dimension is the possibility to provide multiple views in one software visualization for all three aspects. However, empirical findings concerning the role of the third dimension in software visualization are rare. Furthermore, there are only few 3D software visualizations that provide multiple views of a software system including all three aspects. Finally, the current tool support lacks of generating easy integrateable, scalable, and platform independent 2D, 2.5D, and 3D software visualizations automatically.

Hence, the objective is to develop a software visualization that represents all important structural entities and relations of a software system, that can display behavioral and evolutionary aspects of a software system as well, and that can be generated automatically.

In order to achieve this objective the following research methods are applied. A literature study is conducted, a software visualization generator is conceptualized and prototypically implemented, a structured approach to plan and design controlled experiments in software visualization is developed, and a controlled experiment is designed and performed to investigate the role of the third dimension in software visualization.

The main contributions are an overview of the state-of-the-art in 3D software visualization, a structured approach including a theoretical model to control influence factors during controlled experiments in software visualization, an Eclipse-based generator for producing automatically role- and task-specific 2D, 2.5D, and 3D software visualizations, the controlled experiment investigating the role of the third dimension in software visualization, and the recursive disk metaphor combining the findings with focus on the structure of software including useful applications of the third dimension regarding behavior and evolution.

Acknowledgment

I am writing these last lines with pleasure. A long way with ups and downs lies behind me. At this point I would like to take the opportunity to say thank you to all the people who have paved this way and who have accompanied me on this way.
First, I want to thank my parents Bettina and Bernd Müller as they have given me the chance to study and have made this possible in the first place. Moreover, I want to thank my brother Axel Müller who has been a constant role model and a valuable advisor. Another big thank you goes to Mikaela Mollenhauer. She has supported me in an extraordinary manner during the ups and downs.
I also wish to thank Ulrich Eisenecker for the outstanding supervision, the inspiring discussions, and the useful advices regarding science and life. I want to thank Bogdan Franczyk for acting as an assessor of this thesis and for his constructive feedback as well as André Ludwig for his support. Furthermore, I want to thank Dirk Zeckzer for the excellent cooperation. Thanks are also due to my friends and colleagues Johannes Müller, Pascal Kovacs, Jan Schilbach, Christopher Klinkmüller, Ábel Sinkovics, Christoph Jobst, Max Lillack, Christoph Augenstein, Martin Roth, Stefan Mutke, Hendrik Kerkhoff, Robert Kunkel, and David Baum.
Finally, I would like to thank all the students who have contributed directly and indirectly to this thesis, namely Elton Qinami, Christian Mählig, André Naumann, Philipp Günther, Lino Janke, Dan Häberlein, Denise Zilch, Andreas Gessner, and Christian Stein.

"An attempt at visualizing the fourth dimension: Take a point, stretch it into a line, curl it into a circle, twist it into a sphere, and punch through the sphere."

Albert Einstein

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AOPT
Avalon-Optimizer [10, 34, 57, 85]

API
Application Programming Interface [18, 22, 23]

AST
Abstract Syntax Tree [17–19]

BD
Big Data [88]

BI
Business Intelligence [88]

CAD
Computer-aided Design [24]

CASE
Computer-Aided Software Engineering [8]

CMM
Common Meta-Model [9]

CSS
Cascading Style Sheet [33]

CVS
Concurrent Versions System [2, 6]

DMM
Dagstuhl Middle Model [9]

DOM
Document Object Model [14, 30, 31]

DSL
Domain Specific Language [10–12, 17, 18, 20, 34, 56, 85, 88, *Glossary:* DSL]

EBNF
Extended Backus-Naur Form [18]

EMF
Eclipse Modeling Framework [14, 19]

EMP
Eclipse Modeling Project [5, 16, 34, 57, 86]

GP
Generative Programming [10, 13, 34, *Glossary:* GP]

GPU
Graphics Processing Unit [34]

HTML
Hypertext Markup Language [30–34]

HTTP
Hypertext Transfer Protocol [34]

IDE
Integrated Development Environment [2, 15, 16, 18, 23, 34, 86, 88]

ISO
International Organization for Standardization [24]

JAR
Java Archive [17]

JDT
Java Development Tools [5, 16, 17, 23, 34, 57, 86]

LOC
Lines of Code [2, 49, 87]

M2M
Model-to-Model Transformation [15, 24, 57, *Glossary:* M2M]

M2P
Model-to-Platform Transformation [15, 24, 57, *Glossary:* M2P]

MDA
Model-Driven Architecture [12]

MDSD
Model-Driven Software Development [10, 12, 13, 34, *Glossary:* MDSD]

MSE
File exchange format [9, 57]

MWE2
Modeling Workflow Engine 2 [18–21, 24, 57, *Glossary:* MWE2]

OMG
Object Management Group [12]

OSGi
Open Services Gateway initiative [17]

PDE
Plug-in Development Environment [5, 16, 34, 57, 86]

RCP
Rich Client Platform [16, 88]

SAI
Abstract Scene Access Interface [28, 30, 31]

SOA
Service-oriented Architecture [6]

SRC
Shape Resource Container [34]

SVN
Subversion [2, 6]

TMF
Textual Modeling Framework [5, 16, 34, 57, 86]

UA
User Agent [31]

URI
Uniform Resource Identifier [21, 31]

VRML
Virtual Reality Modeling Language [24, 25, 27]

WebGL
Web Graphics Library [31]

X3D
Extensible 3D [5, 10, 12, 24, 25, 27, 28, 30–34, 57, 85, 89, *Glossary:* X3D]

X3DOM
Extensible 3D Document Object Model [5, 10, 12, 24, 30–34, 57, 85, 88, 89, *Glossary:* X3DOM]

XHTML
Extensible Hypertext Markup Language [30, 31, 33, 34]

XML
Extensible Markup Language [13, 24, 27, 28, 30]

XSD
XML Schema Definition [24, 57]

1 Introduction

This chapter motivates the topic of the thesis and formulates the corresponding problem statement. Further, it describes the research design including objective, research questions, and the applied research methods. The introductory chapter closes with a summary of the main contributions and an outline of the thesis.

1.1 Motivation and Problem Statement

Typical software engineering processes are forward and reverse engineering [Chikofsky and Cross 1990] usually controlled by a cross-cutting management process [Frailey et al. 2004]. Forward engineering covers the classical software development process from requirements engineering to the design and the implementation of the software system. On the contrary, reverse engineering aims at analyzing an implemented software system to gain an understanding of its design, and if necessary, of its requirements. This supports the system's maintenance, enhancement, replacement, or reuse. Reverse engineering plays an important role in software engineering because of the rapid development of technology [Garcia et al. 2004]. Software engineering management covers planning, coordinating, measuring, monitoring, controlling, and reporting of these processes. As depicted in Figure 1.1, these software engineering processes comprise several phases[1] and involve stakeholders that use and produce artifacts. Today and in the future, one major challenge is the increasing complexity due to an increase of number, types, and relations of stakeholders on the one hand and of artifacts on the other hand.

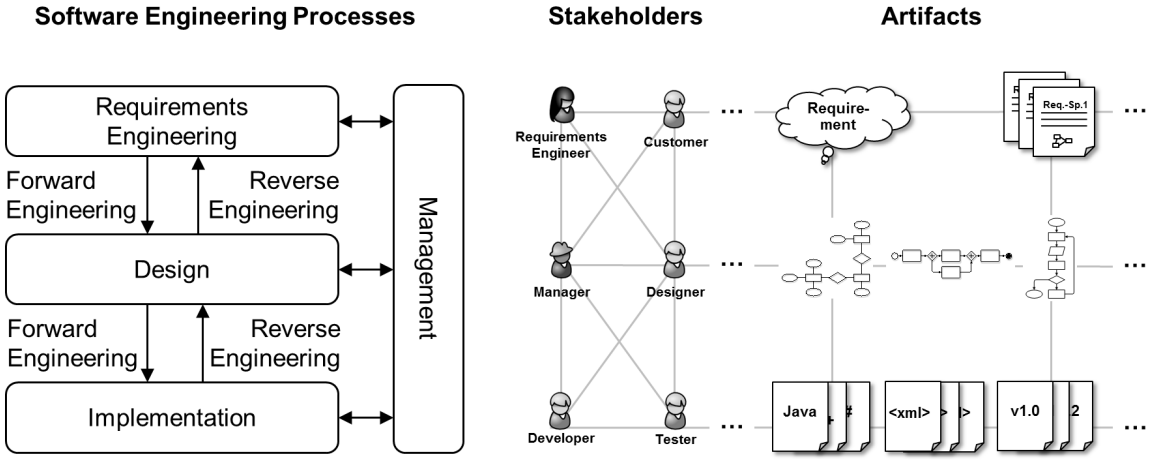


Figure 1.1: Software engineering processes with stakeholders and artifacts.

[1] According to the chosen software development process model, these phases may be processed iteratively.

Software visualization provides means to deal with this complexity and to support all three major software engineering processes [Bohnet and Döllner 2005]. According to Diehl [2007], software visualization is *"[...] the visualization of artifacts related to software and its development process."*. The major objective is to provide role- and task-specific views of these artifacts for stakeholders. The visualized aspects of a software system may be structural, behavioral, and evolutionary. While source code, data structures, or static call-graphs are sources of information regarding the structure of a software system, execution traces provide behavioral information about it. Additionally, information regarding the evolution can be obtained from version control systems, such as Concurrent Versions System (CVS), Subversion (SVN), or Git. For example, a developer can use a structural view of the system to detect design flaws during software quality assessment, a tester is provided with a behavioral view to detect bottlenecks during execution, and a manager is supported in planning and decision making by an aggregated evolutionary view. In general, software visualization tools support stakeholders in software comprehension, finding errors, improving the quality of the software, and managing complexity [Bassil and Keller 2001]. The necessity of software visualization is further confirmed by a survey conducted by Koschke [2003]. The results indicate that 82% of the participants see software visualization as important and absolutely necessary in software engineering. There are many useful 2D, 2.5D, and 3D software visualizations. Some comprehensive overviews are provided by Gračanin et al. [2005], Teyseyre and Campo [2009], and Caserta and Zendra [2011].
The focus of this thesis is on the implementation phase, especially on the combined visualization of artifacts containing information about the structure, behavior, and evolution of software systems. A useful application of the third dimension may be the possibility to provide multiple views with one software visualization for all three aspects, i.e., structure, behavior, and evolution. However, empirical findings concerning the role of the third dimension in software visualization are rare [Müller et al. 2014a, b; Müller and Zeckzer 2015a]. Furthermore, there are only few 3D software visualizations that provide multiple views of a software system including all three aspects [Müller and Zeckzer 2015a]. Finally, there is no general approach to automatically generate 2D, 2.5D, and 3D software visualizations that scale for large software systems (> 500K Lines of Code (LOC)), that are easy to integrate into an Integrated Development Environment (IDE), and that are platform independent [Müller et al. 2011].

1.2 Objective and Research Questions

Consequently, the main objective of this thesis is to develop a software visualization that represents all important structural entities and relations of a software system, that can display behavioral and evolutionary aspects of a software system as well, and that can be generated automatically.

The resulting research question is based on this objective and detailed by three sub-questions.
RQ: How should a software visualization be designed to visualize structural, behavioral, and evolutionary aspects of a software system, and how can it be generated automatically?

- **SQ1:** What is the state-of-the-art in 3D software visualization?
- **SQ2:** How can 2D, 2.5D, and 3D software visualizations be generated automatically?
- **SQ3:** What role does the factor *dimensionality* play in solving software engineering tasks?

1.3 Research Methodology

In order to answer the research questions, the research methodology shown in Figure 1.2 is chosen.

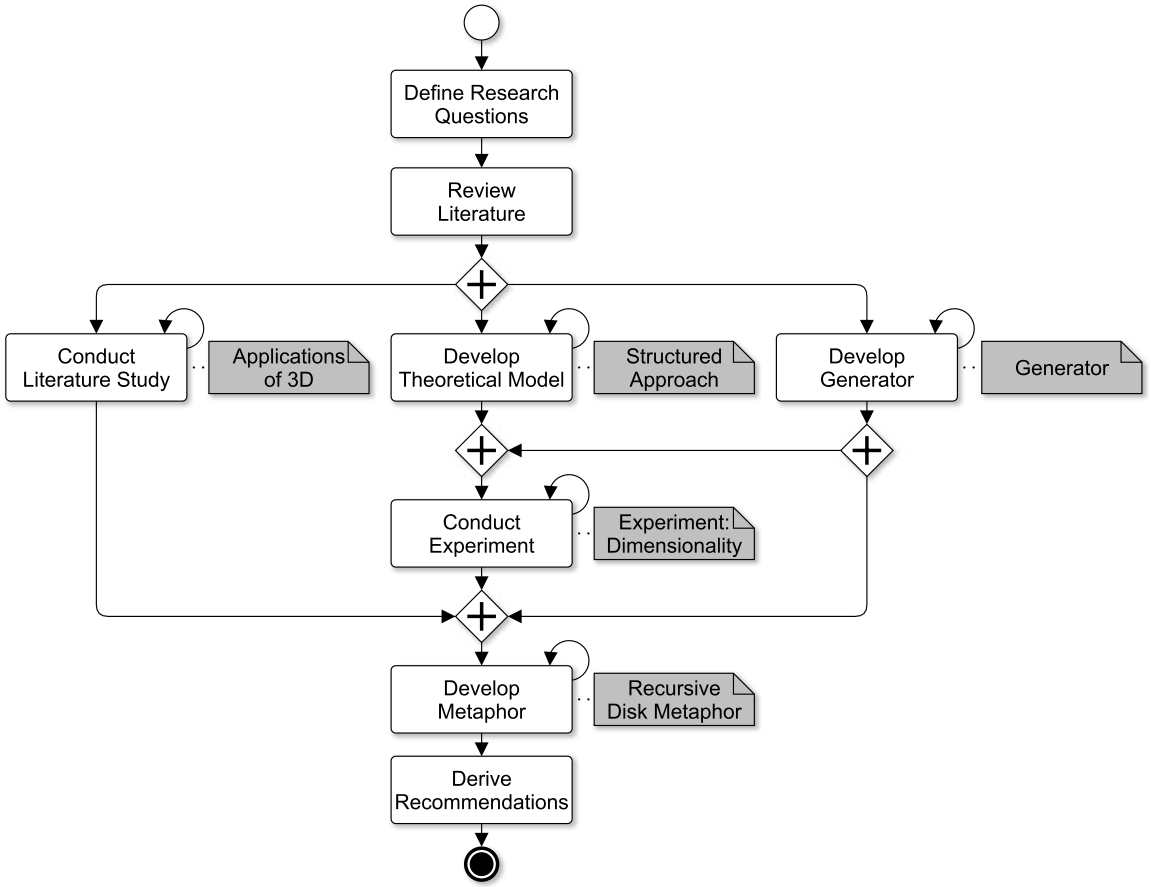


Figure 1.2: Research methodology.

First, the relevant literature is reviewed. This step lays the foundations providing the theoretical and the technical background. The four main research methods applied in this thesis are the literature study combining a systematic mapping study [Petersen et al. 2008] and a systematic literature review [vom Brocke et al. 2009], the development of a prototype [Wilde and Hess 2007], the development of a structured approach, and the controlled experiment [Sjøberg et al. 2007]. The literature study is conducted to examine the state-of-the-art in

3D software visualization (**SQ1**). The prototypical conception and implementation of the software visualization generator (**SQ2**) and the structured approach are two preconditions for the controlled experiment. The generator produces 2D, 2.5D, and 3D software visualizations. The approach helps to plan and to design controlled experiments and to control the influence factors. The experiment is designed and performed to empirically investigate the role of the third dimension in solving software engineering tasks (**SQ3**). The findings of the previous steps are combined and form the input for the development of the recursive disk metaphor (**RQ**). All major steps have an iterative character, as they are repeated during the whole research process. Finally, recommendations for 3D software visualizations are derived.

1.4 Contributions

Altogether, there are five main contributions which have all been published separately as peer-reviewed conference and workshop articles.

1. Literature study presenting an overview of state-of-the-art in 3D software visualization [Müller and Zeckzer 2015a].
2. Eclipse-based generator for generating 2D, 2.5D, and 3D software visualizations automatically [Müller et al. 2011].
3. Structured approach for conducting controlled experiments in software visualization [Müller et al. 2014a].
4. Controlled experiment investigating the role of the third dimension in software visualization [Müller et al. 2014b].
5. Recursive disk metaphor combining the findings with focus on the structure of software with useful applications of the third dimension [Müller and Zeckzer 2015b].

1.5 Outline

The structure of the thesis is based on the research process. In Chapter 2 the theoretical and technical foundations are explained. The literature study giving an overview of the state-of-the-art in 3D software visualization is described in Chapter 3. The generator to create 2D, 2.5D, and 3D software visualizations automatically as well as the structured approach for conducting controlled experiments in software visualization are presented in Chapters 4 and 5. The actual experiment investigating the role of the third dimension in solving software engineering tasks is subject of Chapter 6. The recursive disk metaphor brings all findings of the previous chapters together and is introduced in Chapter 7. Finally, the contributions are summarized, the recommendations for 3D software visualization are concluded, and an outlook to future work is provided in Chapter 8.

2 Background

This chapter explains the theoretical foundations of software visualization and of relevant software engineering paradigms. Further, it introduces the important technical concepts based on Eclipse, such as Java Development Tools (JDT), Plug-in Development Environment (PDE), Eclipse Modeling Project (EMP), and Textual Modeling Framework (TMF) with Xtext, and Xtend as well as Extensible 3D (X3D) and Extensible 3D Document Object Model (X3DOM) to create and render 3D scenes.

2.1 Software Visualization

Software visualization is a branch of information visualization as it offers techniques and methods to visualize abstract data [Diehl 2007, p. 3]. In addition, it has an interdisciplinary character that is affected by software engineering, human computer interaction, graphics, and cognitive psychology [Marcus et al. 2005]. There are two main fields of application for software visualization: educational visualization and software engineering [Hundhausen 1996]. The scope of this thesis is placed on 3D software visualization for software engineering. In the following, the working definition for software visualization is chosen and explained, the supported software engineering tasks are introduced, important taxonomies are presented, and the visualization process is described in detail.

2.1.1 Definitions

The field of software visualization has continually developed since its beginnings in the late 1980's. Table 2.1 summarizes the most influencing definitions.

Table 2.1: Definitions for software visualization.

Authors	Definition
Myers [1990]	*"Program visualisation uses graphics to illustrate some aspect of the program or it's run-time execution, where the program is specified in a conventional, textual manner."*
Price et al. [1993]	*"Software visualisation is the use of the crafts of typography, graphic design, animation and cinematography with modern human- computer interaction technology to facilitate both the human understanding and effective use of computer software."*
Roman and Cox [1993]	*"Program visualisation is a mapping, or transformation, of a program to a graphical representation."*

Continued on next page

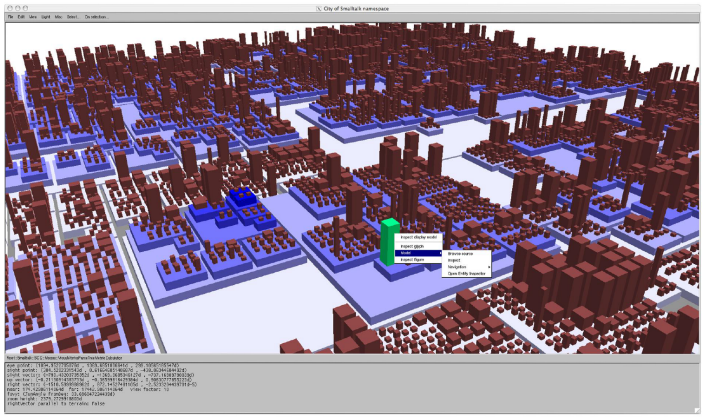
Table 2.1 – continued from previous page

Authors	Definition
Knight and Munro [1999]	*"Software visualisation is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration."*
Reiss [2005]	*"[...] the development and evaluation of methods for graphically representing different aspects of software, including its structure, its abstract and concrete execution, and its evolution."*
Diehl [2007, p. 3f]	*"[...] the visualization of artifacts related to software and its development process. [...] visualizing the structure, behavior, and evolution of software."*

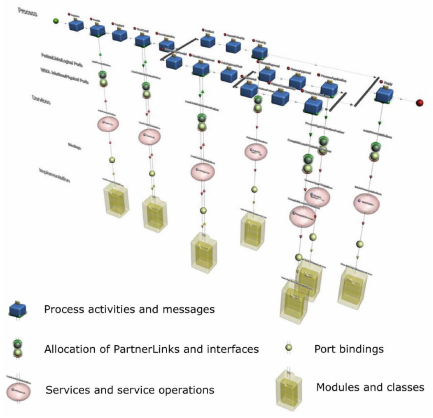
Diehl [2007, p. 3f] provides the most recent definition of software visualization, including all necessary aspects, i.e., structure, behavior, and evolution. For these reasons, this definition is used throughout the thesis.

- *Structure* includes program code, data structures, static call-graphs, relations, and the organization of a software system.
- *Behavior* covers the execution of a software system with real and abstract data.
- *Evolution* refers to the development process of a software system. This information is usually provided by version control systems, such as CVS, SVN, or Git.

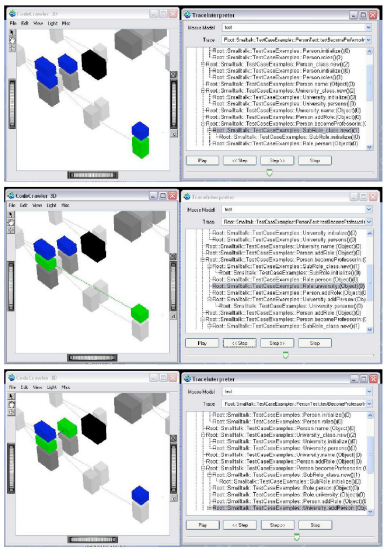
Figure 2.1 shows examples of software visualizations including structural (a-b), behavioral (c-d), and evolutionary (e-f) aspects. Wettel and Lanza [2007] visualize the structure of software systems with a city metaphor (a). Here, packages are mapped to districts and classes are mapped to buildings. The base area of the buildings is proportional to the number of attributes and their height is proportional to the number of methods of a class. Eicker et al. [2007] provide structural views of Service-oriented Architectures (SOAs) divided into business process layer, service interface layer, and application layer (b). Greevy [2007] visualizes execution traces in terms of object creations and interactions in the context of a static class hierarchy (c). Von Pilgrim and Duske [2008] represent execution traces with stacked views (d). Ripley et al. [2007] offer means to visualize and explore workspace activity as well as evolution on a project-wide basis. The workspaces are mapped to a stack of cylinders where each cylinder corresponds to an artifact. Stacks of cylinders with recent changes are placed in front of the view. Stacks with less activity move into the background (e). Steinbrückner and Lewerentz [2010] use the city metaphor to visualize the evolution of software systems. Their approach maps the development history to elevation levels. The levels correspond to the number of versions of an artifact. Thus, the higher an artifact is placed, the higher version it has. Besides system evolution, the approach includes modification history and authorship history (f).



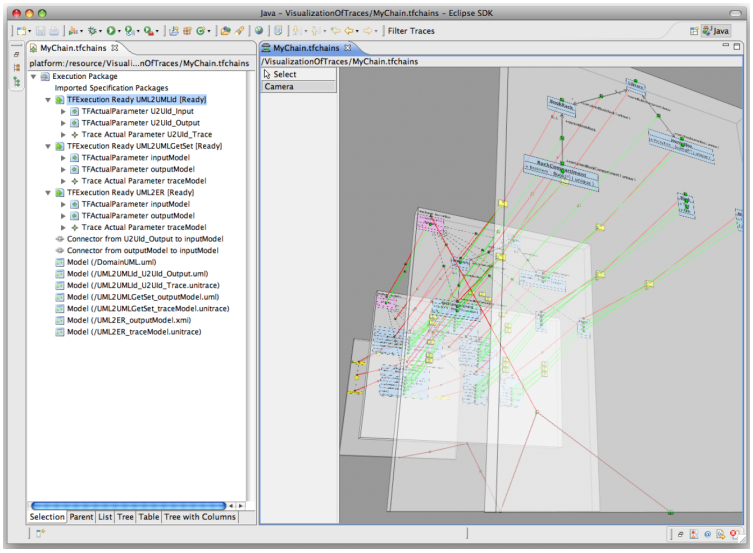
(a)



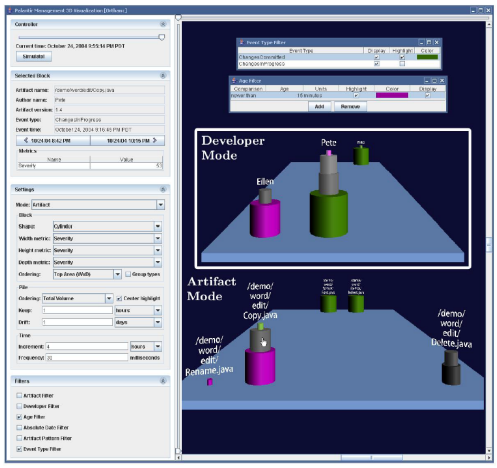
(b)



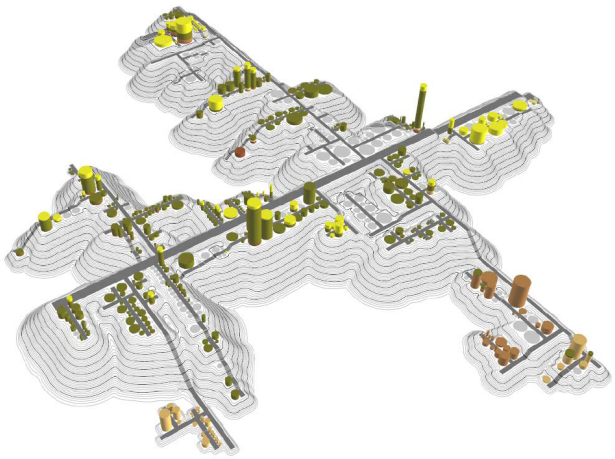
(c)



(d)



(e)



(f)

Figure 2.1: Software visualization examples showing structural, behavioral, and evolutionary aspects: structure with (a) CodeCity [Wettel and Lanza 2007] and (b) SOA views [Eicker et al. 2007], behavior with (c) CodeCrawler [Greevy et al. 2005] and (d) GEF3D [von Pilgrim and Duske 2008], evolution with (e) Palantír [Ripley et al. 2007] and (f) Evo-Streets [Steinbrückner and Lewerentz 2010].

2.1.2 Supported Software Engineering Tasks

Software visualization enhances comprehension of software systems in constructive and analytical tasks in software engineering. Actually, it supports three major software engineering processes with their corresponding tasks [Bohnet and Döllner 2005, p. 4]:

- *forward engineering*, i.e., design and implementation of new software systems,
- *reverse engineering*, i.e., maintenance, enhancement, and reuse of existing software systems, and
- *management*, i.e., planning, coordinating, measuring, monitoring, controlling, and reporting the software engineering process.

In general, software visualization tools support stakeholders in software comprehension, finding errors, improving the quality of the software, and managing complexity [Bassil and Keller 2001]. The forward engineering process covers the classical software development process from requirements engineering to the design and the implementation of the software system. This process is usually supported by Computer-Aided Software Engineering (CASE) tools. These tools provide structural views, as well as behavioral views of the software system under development. The reverse engineering process aims at gaining sufficient design-level understanding about an existing software system to help with its maintenance, enhancement, replacement, and reuse [Chikofsky and Cross 1990]. Structural, behavioral, and evolutionary views of the initially unknown system help to solve these tasks time- and cost-efficiently. Reverse engineering plays an important role in software engineering due to the rapid development in technology [Garcia et al. 2004]. For this reason, software visualization in the context of reverse engineering is an important use case. The management process covers the planning, coordinating, measuring, monitoring, controlling, and reporting of the software engineering process [Frailey et al. 2004]. In this context the main source for visualization are evolutionary aspects of software systems, such as system evolution, modification history, and authorship history. The aggregation and visualization of this information may support the management in planning and decision making.

2.1.3 Taxonomies

Taxonomies are a mean to structure a discipline in order to communicate on a common basis among researchers and to identify research gaps. Table 2.2 builds partly upon Hundhausen et al. [2002, p. 261] and summarizes the main taxonomies in the field of software visualization.

Table 2.2: Taxonomies for software visualization.

Taxonomy	Dimensions
Myers [1990]	Aspect (Code, Data, Algorithm) × Form (Static, Dynamic)

Continued on next page

Table 2.2 – continued from previous page

Taxonomy	Dimensions
Stasko and Patterson [1993]	Aspect × Abstractness × Animation × Automation
Roman and Cox [1993]	Scope × Abstraction Level × Specification Method × Interface × Presentation
Price et al. [1993]	Scope × Content × Form × Method × Interaction × Effectiveness
Maletic et al. [2002]	Task × Audience × Target × Representation × Medium
Storey et al. [2005]	Intent × Information × Presentation × Interaction × Effectiveness
Gallagher et al. [2005]	Static Representation × Dynamic Representation × Views × Navigation and Interaction × Task Support × Implementation × Visualization

These taxonomies play an important role in developing the theoretical model for conducting controlled experiments because they reveal all important aspects that may influence the processing of a software engineering task supported by a software visualization.

2.1.4 Metamodels

According to Diehl [2007, p. 3f], the information of software systems to be visualized may be structural, behavioral, or evolutionary. For each of these aspects exists a metamodel. Structural information is covered by Famix [Ducasse et al. 2011], behavioral information is covered by Dynamix [Greevy 2007], and evolutionary information is covered by Hismo [Ducasse et al. 2004]. These metamodels are independent of the actual programming language, execution trace tool, or version control system. The serialization format of Famix, Dynamix, and Hismo is MSE (meaning unknown) [Kuhn and Verwaest 2008]. There are alternative metamodels, such as the Dagstuhl Middle Model (DMM) [Lethbridge et al. 2004] and the Common Meta-Model (CMM) [Strein et al. 2007]. However, none of these alternative models covers all three aspects.

2.1.5 Visualization Pipeline

The main steps of a general *visualization pipeline* are extraction, analysis, filtering, mapping, and rendering [dos Santos and Brodlie 2004]. In this thesis, this general visualization process is adapted to software visualization and realized with a generator. The adapted visualization pipeline is depicted in Figure 2.2.

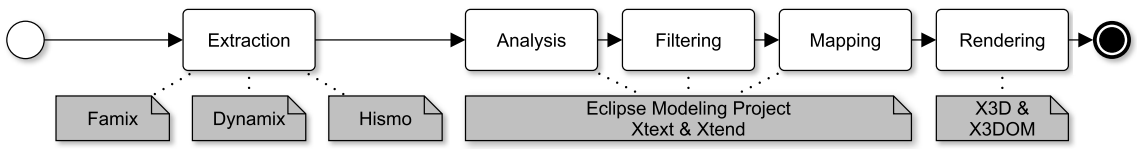


Figure 2.2: Adapted visualization pipeline for generating software visualizations based on dos Santos and Brodlie [2004].

The information needed for the visualization is *extracted* from software systems and stored in corresponding models, either conforming to Famix [Ducasse et al. 2011], to Dynamix [Greevy 2007], or to Hismo [Ducasse et al. 2004]. These metamodels are defined with Xtext that automatically generates the necessary language infrastructure, including parsers and validators, for each metamodel. During *analysis*, these models are checked for syntactic and semantic validity. They must conform to their metamodel and fulfill some predefined validation rules, e.g., each entity must have a unique identifier. In the next step, the user *filters* the desired entities. This step may occur at build time of the visualization or at runtime. The *mapping* is realized by model transformations and model modifications using Xtend. It is divided into two parts. First, the valid and filtered entities from the input model are mapped to a platform independent model. Then, the layout of these entities is computed providing sizes and positions for the visualization. Second, the platform independent model is mapped to a platform specific one, here, X3D. Finally, the X3D model is optimized for the web and converted with the Avalon-Optimizer (AOPT) to X3DOM [Behr et al. 2012]. The resulting visualization is *rendered* by a browser. As the rendering may be done on every platform, X3D and X3DOM are platform independent.

2.2 Adapted Software Engineering Paradigms

The generator to be developed adapts and combines concepts and techniques of Generative Programming (GP) and of Model Driven Software Development (MDSD) from the field of software engineering. For this reason, both paradigms are described in the following.

2.2.1 Generative Paradigm

A comprehensive overview of GP is for example provided by Czarnecki and Eisenecker [2000] and by Czarnecki [2005]. Especially Czarnecki and Eisenecker [2000] is used as a primary source to explain the terminology and concepts of the generative paradigm.

Definition

"Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge." [Czarnecki and Eisenecker 2000, p. 5]

According to this definition, the generative paradigm aims at the development of a set of software systems, a so called system family. A *system family* covers a set of of systems that are similar enough in terms of architecture to be assembled by a common set of components [Czarnecki and Eisenecker 2000, p. 31]. The requirements of the product to be created are described with a Domain Specific Language (DSL). A *domain* is a bounded field of knowledge

comprising professional knowledge on the one hand and technical knowledge on the other hand [Czarnecki and Eisenecker 2000, p. 34]. The professional knowledge includes concepts and terminologies understood by practitioners and the technical knowledge includes ways of how to build software systems. Further, a domain is always related to its stakeholders. A *DSL* is specialized, problem-oriented, and provides means to describe concrete members of a system family [Czarnecki and Eisenecker 2000, p. 137]. The resulting specification is handed over to a generator. The generator assembles the desired product by combining elementary and reusable components. *Components* are building blocks that are used to assemble different systems of a system family [Czarnecki and Eisenecker 2000, p. 9]. A *generator* is a piece of software that produces a system automatically according to the specification [Czarnecki and Eisenecker 2000, p. 333ff]. There are four essential tasks of a generator. It

- completes the specification with default values,
- verifies the specification and reports warning and error messages,
- performs optimizations, and
- generates artifacts.

The result of the generation process is a member of the system family and shares the common system family architecture, no matter if it is an intermediate or a final artifact.

Generative Domain Model

The *generative domain model* plays an important role in the context of this paradigm, as it summarizes its important terms and their relations. It is shown in Figure 2.3.

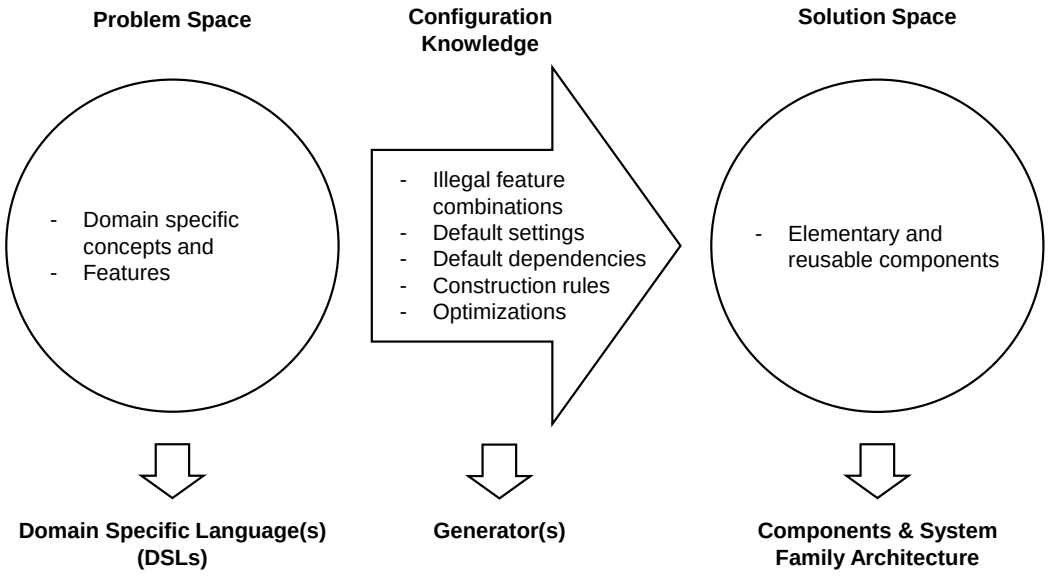


Figure 2.3: Elements of the generative domain model [Czarnecki and Eisenecker 2000, p. 132].

The generative domain model consists of the problem space, the solution space and the configuration knowledge [Czarnecki and Eisenecker 2000, p. 131f]. The *problem space* provides

domain specific concepts and features to specify members of a system family by means of a DSL. The *solution space* covers the elementary and reusable implementation components and the common system family architecture. The *configuration knowledge* maps the elements of the problem space to the elements of the solution space and is usually implemented as a generator. The mapping process considers information about illegal feature combinations, default settings, default dependencies, construction rules, and optimizations.
As depicted in Figure 2.4, there are different ways of mappings between the problem and the solution space [Czarnecki 2005]. The generative domain model can be processed recursively, i.e., the solution space of one model is simultaneously the problem space of another generative domain model. This results in a chained mapping (a). Moreover, multiple problem spaces, either defined by a composed DSL (b) or by alternative DSLs (d), can be mapped to one solution space. Finally, one problem space can be mapped to complementary (c) or alternative (e) solution spaces. In practice, there may be more complex combinations of these elementary mapping patterns.

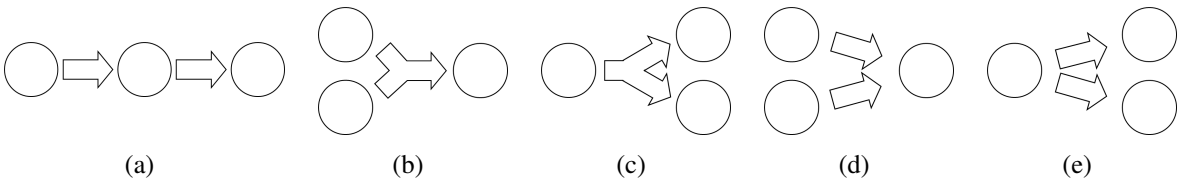


Figure 2.4: Mapping alternatives between problem and solution space [Czarnecki 2005]: (a) chained mapping, (b) multiple problem spaces, (c) multiple solution spaces, (d) alternative problem spaces, (e) alternative solution spaces.

Technology Projection

The elements of the generative domain model may be implemented with different techniques. The mapping of these elements to a paradigm, a programming language, or a platform is called *technology projection* [Czarnecki 2005]. An overview of several technology projections is provided by Czarnecki [2005]. In this thesis, Eclipse, especially Xtext and Xtend, as well as X3D and X3DOM are chosen as technology projections.

2.2.2 Model-Driven Paradigm

A comprehensive overview of MDSD is provided by for example Stahl et al. [2006, 2007]. MDSD can be seen as the practical realization, driven by the developer community, of the Model-Driven Architecture (MDA) standard [Miller and Mukerji 2003] initiated by the Object Management Group (OMG). Völter et al. [2013] have developed MDSD further and have brought the generative and model-driven paradigms closer together. Thus, both sources [Stahl et al. 2006, 2007; Völter et al. 2013] are combined to explain the terminology and concepts of the model-driven paradigm.

Definition

"MDSD is a generic term for techniques, that create runnable software from formal models automatically."[2] [translated into English from Stahl et al. 2007, p. 11]

According to this definition, in the model-driven paradigm models become central artifacts of software development, as they are at least as important as source code. Models are specified in a way allowing the automatic generation of parts of a software system or a complete software system by means of transformations.

Metamodels and Models

Similar to GP, in the context of MDSD, the domain plays an important role. In order to provide the basis for automation, the domain's structure has to be formalized. This formalization is realized by the metamodel. The relations between all relevant terms of the model-driven paradigm are outlined in Figure 2.5.

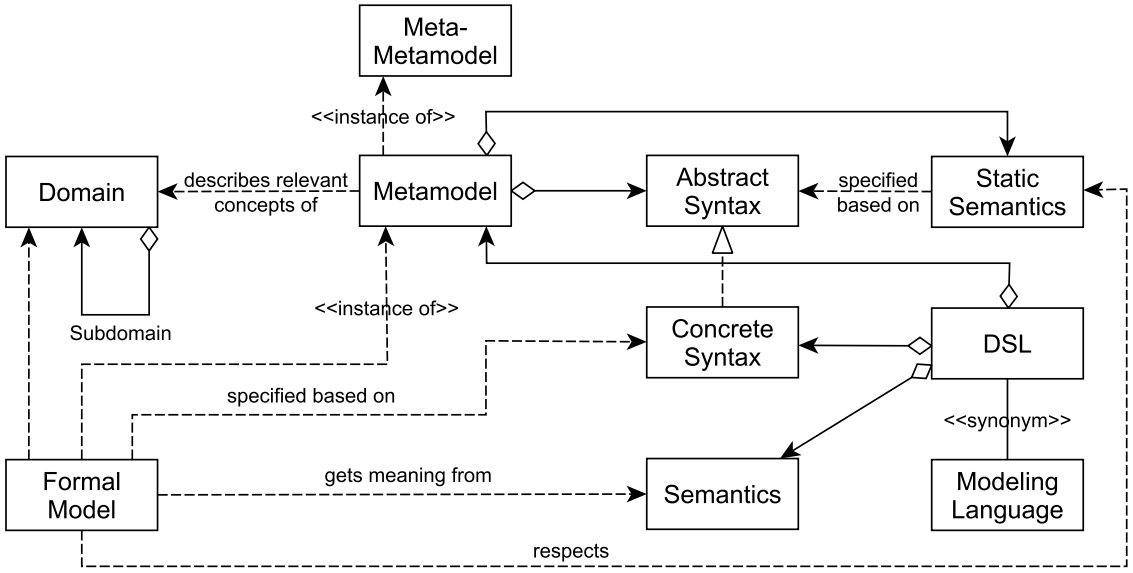


Figure 2.5: Relations between domain, DSL, formal model, and metamodel [Stahl et al. 2006, p. 56].

The *metamodel* defines the abstract syntax and the static semantics of a language [Stahl et al. 2006, p. 55ff]. In other words, it describes concepts that can be used for creating the formal model. The *abstract syntax* of a language specifies the language's structure, typically as a tree or a graph. The realization of an abstract syntax is the *concrete syntax* that is accepted by a parser and used as notation to specify formal models. It may be textual, graphical, tabular, or a combination of these [Völter et al. 2013, p. 27]. For example, in Extensible Markup Language (XML) the XML document is formulated in the concrete syntax of XML. A parser

[2] *"Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen." [Stahl et al. 2007, p. 11]*

instantiates a representation in memory, where the resulting Document Object Model (DOM) is the abstract syntax of XML. The syntax of a language is not sufficient to check semantic rules. For this reason, the criteria for well-formedness of a language are defined by its *static semantics*. These are a set of constraints and/or type system rules to which formal models have to conform. Additionally, the *dynamic semantics* gives a meaning to the metamodel's constructs. Finally, a *formal model* is formulated in the concrete syntax and follows its semantic, i.e., it is an instance of the metamodel. Subsequently, the term model is used as a synonym for a formal model. In order to finish the theoretically infinite instance hierarchy of models and their metamodels, at a certain point the metamodel is defined in itself.
An important meta-metamodel in the Eclipse ecosystem, especially in the Eclipse Modeling Framework (EMF), is Ecore. It provides the language including syntax and semantics for defining custom metamodels for a domain. A subset of the important elements of Ecore and their relations are shown in Figure 2.6.

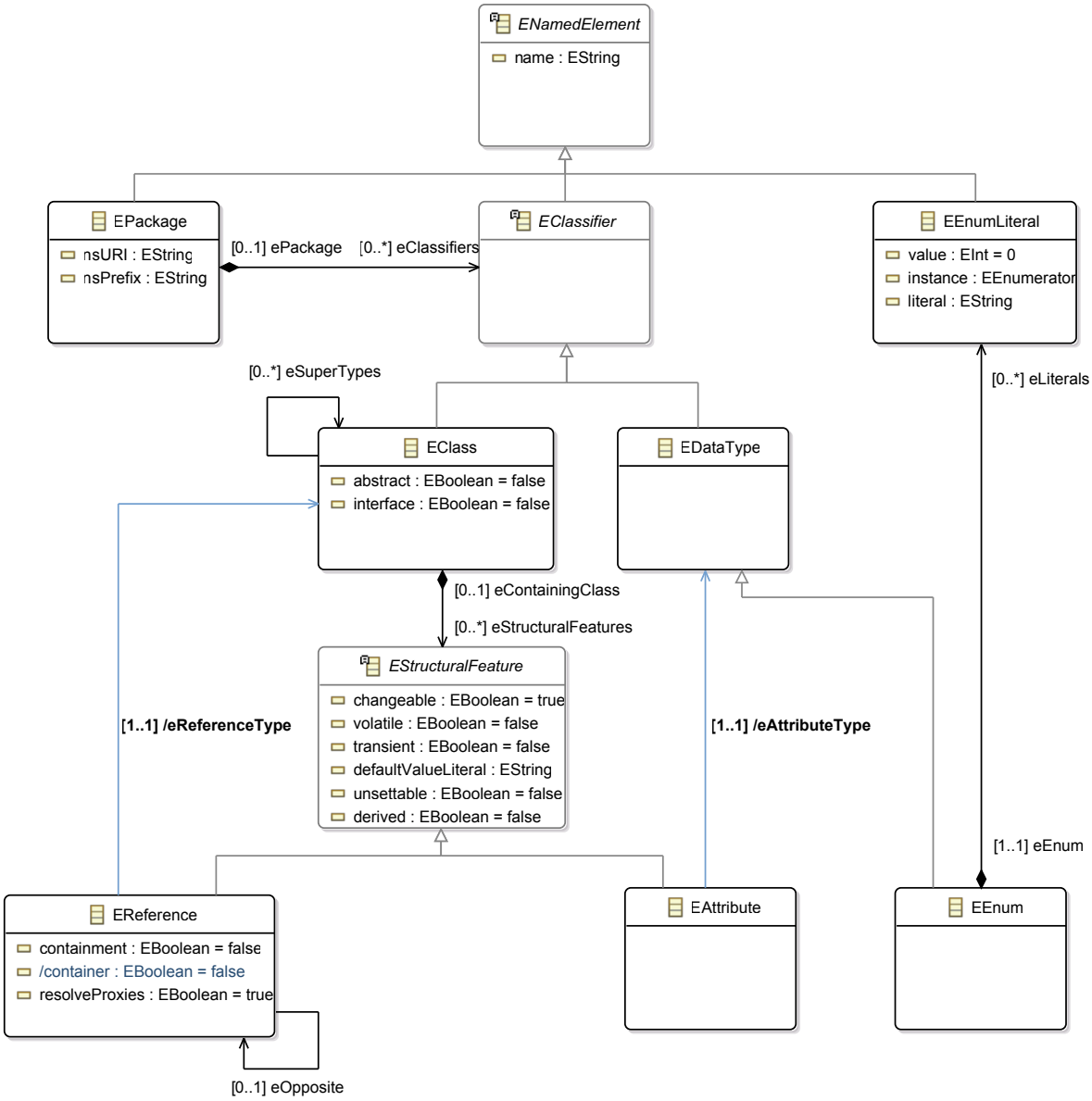


Figure 2.6: Subset of Ecore.

The class `EPackage` represents packages and may contain `EClasses`. This relation is modeled by the reference `eClassifiers` to the abstract class `EClassifier`. `EClass` defines classes and holds references to other classes as well as attributes. These relations are modeled by the reference `eStructuralFeatures` to the abstract class `EStructuralFeature`. The references between classes, represented by `EReference`, may be inheritance, composition, or aggregation. `EAttribute` stands for attributes and has a reference to its type. Primitive and complex datatypes are specified by `EDataType`. A special datatype is the enumeration represented by `EEnum` containing literals of type `EEnumLiteral`.

Model Transformations

A *platform* supports the realization of a domain and can be founded on existing building blocks, such as middleware, libraries, frameworks, or components [Stahl et al. 2006, p. 59]. Transformations are necessary to connect the domain concepts of the problem space with the platform in the solution space. Transformation rules are based on the metamodel's constructs. There are two types of model transformations: Model-to-Platform Transformation (M2P) and Model-to-Model Transformation (M2M) [Stahl et al. 2006, p. 60]. A *M2P* generates artifacts that are based on the platform. This process is also called generation. An example is generated source code that fits into an existing framework. For this purpose, templates describe how source code should be generated from model elements. A *M2M* maps one or more source models to a target model. There are three different categories: Model transformation, model modification, and model weaving [Stahl et al. 2007, p. 199f]. A *model transformation* maps a source model to a target model, where both models conform to a different metamodel. That means, there is a metamodel change and the source model remains unchanged. In contrary, a *model modification* changes or extends source model elements, i.e., the target model is the modified source model and still conforms to the same metamodel. In some cases, information is distributed in different models and needs to be joined. Here, a *model weaving* combines at least two source models and creates one target model.

Tool Support

A suitable tool support is necessary to realize a model-driven approach. Czarnecki and Helsen [2006] provide a comprehensive overview of such tools. One of the tools mentioned in the survey is openArchitectureWare, the predecessor of Xtext and Xtend explained in the next section.

2.3 Eclipse

Eclipse is an open-source and generic IDE for a myriad of purposes. In other words: *"The Eclipse Project provides a kind of universal tool platform - an open extensible IDE for anything and yet nothing in particular."* [Eclipse Website 2014]. Eclipse can be used as an IDE for a certain programming language, as an IDE framework combining different programming

languages, as a tool framework, as an application framework, and as a runtime environment. The reason of this variety lies in the IDE's plug-in architecture. The main task of the small Eclipse core is to load plug-ins at runtime. These plug-ins provide the actual functionalities of Eclipse. The Eclipse Project addresses two areas. On the one hand, there is the development of tool integration platforms including core frameworks and technologies that are used as a base to build software development tools of all kinds. On the other hand, there are tools required to build and to extend these platforms. These tools are used to build the integration platforms, to extend, and to adopt the platforms.
For the generator, the following Eclipse projects are important: JDT and PDE as parts of the Eclipse Project, as well as Xtext and Xtend as parts of the TMF respectively the EMP. At the moment, the generator is implemented with version 4.3 (Kepler) of Eclipse and version 2.6 of Xtext and Xtend.

2.3.1 Java Development Tools

The JDT project provides tools to develop, test, debug, build, and deploy Java applications, including Eclipse plug-ins [JDT Project Website 2014]. The Java IDE adds a Java project nature and a Java perspective to the Eclipse workbench enhanced by corresponding views, editors, wizards, builders as well as code merging and refactoring tools.

2.3.2 Plug-in Development Environment

The PDE project provides tools to develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and Rich Client Platform (RCP) products [PDE Project Website 2014]. The PDE also adds perspectives, views, editors, wizards, and builders to the Eclipse workbench. Further, it is highly integrated with the JDT. The modular and extensible plug-in architecture of the generator is realized with three concepts that are described next, namely plug-ins, fragments, and features.

Plug-in

A *plug-in* is the smallest functional unit in Eclipse [Eclipse Documentation 2014]. It can be an extension and it can offer extension points as placeholders for other plug-ins. The extension concept is explained in Figure 2.7.

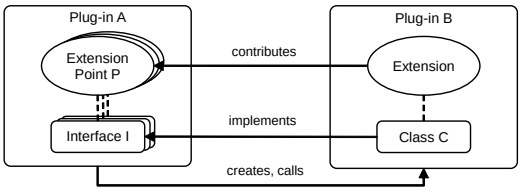


Figure 2.7: Extension points and extensions in Eclipse.

Plug-in A defines the extension point P and its interface I. Plug-in B contributes an extension to plug-in A as it implements this interface in class C. At runtime, plug-in A instantiates this class by calling the methods of the interface.
Every plug-in consists at least of a manifest (`META-INF/MANIFEST.MF`). The manifest contains all necessary information about its execution including name, version, and dependencies to other plug-ins. The runtime management of a plug-in is controlled by an implementation of the Open Services Gateway initiative (OSGi) framework. The extended manifest (`plugin.xml`) specifies the extensions of the plug-in and the extension points for other plug-ins. Further optional parts of a plug-in are the Java byte-code, usually shipped as a Java Archive (JAR) file, as well as resources, such as icons, help pages, or internationalized strings.

Fragment

A *fragment* is always part of a plug-in [Eclipse Documentation 2014]. It extends a plug-in non-invasively with further contents or functionality. For example, language packages are often implemented as fragments and added after the development of the plug-in has been finished. Its manifest (`fragment.xml`) controls the coupling of the fragment and the plug-in. Apart from this, the structure of a fragment is similar to plug-ins.

Feature

A *feature* combines related plug-ins and their fragments to a product [Eclipse Documentation 2014]. For example, the JDT is a feature consisting of plug-ins, such as a Java source code editor, a debugger, and a console. The relations between plug-ins, fragments, and features are illustrated in Figure 2.8. Here, feature x bundles the plug-ins A, B, C and fragment 1. Feature y introduces fragment 2 that extends plug-in C.

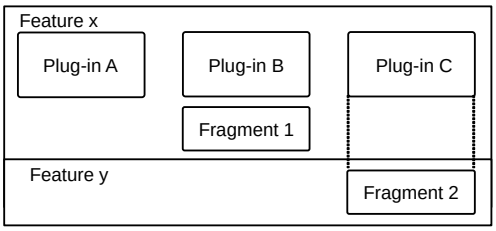


Figure 2.8: Relations between plug-ins, fragments, and features [Daum 2008, p. 499].

2.3.3 Xtext

Xtext is a language development framework for programming languages and DSLs [Xtext Documentation 2014]. It covers all aspects of a complete language infrastructure by providing runtime components, such as a parser, a type-safe Abstract Syntax Tree (AST), a serializer, a code formatter, a linker, compiler checks, a factory, a validation component for static analysis, and a code generator. Moreover, Xtext is completely integrated into the Eclipse

IDE. It uses the dependency injection framework Google Guice to wire up the whole language as well as the IDE infrastructure. All subsequent statements refer to version 2.7.2 of Xtext.
In the context of the generator, Xtext's grammar language is used to define the metamodels, i.e., Famix, Dynamix, and Hismo. The desired components are described in a Modeling Workflow Engine 2 (MWE2) configuration that is the input for the language generator. The language generator creates important components for the visualization process such as parsers, validators, and the Application Programming Interfaces (APIs).

The Grammar Language

The grammar language is a DSL for the description of textual languages [Xtext Documentation 2014]. It is used to describe the concrete syntax and how the syntax is mapped to an in-memory representation—the AST. The AST will be created by the parser when it reads an input file written in this language. Every Xtext grammar starts with a header that defines some properties of the language. Additional grammars can be included with the keyword `with`. This mechanism is called *grammar mixin*. An existing Ecore model can be referenced with the keyword `import` and aliased to avoid name collisions with the keyword `as` followed by a name.
There are four different types of rules available to define the grammar: terminal rules, parser rules, data type rules, and enum rules. A *terminal rule* is described using Extended Backus-Naur Form (EBNF)-like expressions. Return types are atomic values of type `EDataType`. Examples for terminal rules are in Listing 2.1 in lines 4 to 18. A hidden terminal symbol defines a sequence of patterns that are ignored by the parser, such as white space, and comments. Nevertheless, they are woven into the model but do not play any role for the semantic model. They are shipped as default in `Terminals.xtext`.

```
1  grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)
2  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3
4  terminal ID:
5    '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
6  terminal INT returns ecore::EInt:
7    ('0'..'9')+;
8  terminal STRING :
9    '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|'"') )* '"' |
10   "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */ | !('\\'|"'") )* "'";
11 terminal ML_COMMENT:
12   '/*' -> '*/';
13 terminal SL_COMMENT:
14   '//' !('\n'|'\r')* ('\r'? '\n')?;
15 terminal WS:
16     (' '|'\t'|'\r'|'\n')+;
17 terminal ANY_OTHER:
18   .;
```

Listing 2.1: Xtext grammar of important terminal rules [Xtext Documentation 2014].

A *parser rule* uses terminal rules and other parser rules. The parser rules lead to the parse tree. This is the blueprint for the AST. In this context, actions and assignments are used to derive types and initialize the elements of the AST. If not explicitly specified the return type of a parser rule is the rule's name. *Assignments* (=, +=,?=) are used to bind the consumed information to a feature of the currently produced object. The type of this object is specified by the return type of the parser rule. It is possible to declare cross-references in the grammar. This information is used by the linker. *Actions* make the creation of a return type explicit. There are simple actions and assigned actions. A *data type rule* creates instances of `EDataType` instead of `EClass`. They are similar to terminal rules but they are context sensitive and allow the use of hidden tokens. An *enum rule* returns enumeration literals from strings. It is a kind of data type rule with a specific value converter and creates an instance of `EEnum`. Xtext infers Ecore models from a grammar. Xtext parsers create an in-memory object graph that is an instance of Ecore models. Such a model consists of an `EPackage` containing `EClasses` with `EAttributes` and `EReferences`, `EDataTypes` and `EEnums` according to the different parser rules. Xtext generates or infers an Ecore model from every grammar as follows [Xtext Documentation 2014]:

- An `EPackage` for each `generate` declaration,
- an `EClass` for each return type of a parser rule and for each type defined in an action or a cross-reference,
- an `EEnum` for each return type of an enum rule,
- an `EDataType` for each return type of a terminal rule or a data type rule,
- an `EAttribute` in each current return type, and
- an `EReference` in each current return type for each assignment and for each assigned action.

MWE2

The MWE2 is a declarative, externally configurable generator engine [Xtext Documentation 2014]. It provides means to describe object compositions and to declare object instances, attribute values, and references. Its main purpose is the definition of workflows. A *workflow* summarizes components that interact with each other. Among others, there are components to read and write EMF models, or to transform them. However, it is also possible to write custom components and to integrate them into MWE2 workflows.

The Language Generator

The language generator takes an Xtext grammar as input and generates necessary components of the language infrastructure according to the MWE2 configuration [Xtext Documentation 2014]. This includes a parser, a serializer, an inferred Ecore model, and a couple of base classes for validation, formatting, and testing. The generator also contributes to shared project resources such as the `plugin.xml`, `MANIFEST.MF`, and the Google Guice modules.

Famix Example

As mentioned in Section 2.1, Famix is a metamodel describing structural aspects of software [Ducasse et al. 2011]. In the following example, this metamodel is defined by means of an Xtext grammar. Then, the language generator is used to generate the language infrastructure and necessary components. The whole process is configured with MWE2.
Listing 2.2 shows a subset of the Xtext grammar definition of Famix. In other words, this is a DSL for describing structural aspects of software systems. The complete grammar definition of Famix can be found in Listing A.2.

```
1  grammar org.svis.xtext.Famix with org.eclipse.xtext.common.Terminals
2  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3  generate famix "http://www.svis.org/famix"
4
5  Root:
6   document=Document?;
7
8  Document:
9   {Document}
10  '(' elements+=FAMIXElement* ')';
11
12 FAMIXElement:
13  FAMIXNamespace | FAMIXClass | FAMIXAttribute | FAMIXMethod | FAMIXInheritance;
14
15 FAMIXNamespace:
16  '(FAMIX.Namespace'
17  '(' 'id: ' name=INT_ID ')'
18  '(' 'name' value=MSESTRING ')'
19  ('(' 'isStub' isStub=Boolean ')')?
20  ('(' 'parentScope' parentScope=IntegerReference ')')?
21  ')';
22
23 FAMIXClass:
24  '(FAMIX.Class'
25  '(' 'id: ' name=INT_ID ')'
26  '(' 'name' value=MSESTRING ')'
27  '(' 'container' container=IntegerReference ')'
28  ('(' 'isInterface' isInterface=Boolean ')')?
29  ('(' 'isStub' isStub=Boolean ')')?
30  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
31  ('(' 'sourceAnchor' type=IntegerReference ')')?
32  ')';
33
34 FAMIXAttribute:
35  '(FAMIX.Attribute'
36  '(' 'id: ' name=INT_ID ')'
37  '(' 'name' value=MSESTRING ')'
38  '(' 'declaredType' declaredType=IntegerReference ')'
39  ('(' 'hasClassScope' hasClassScope=Boolean ')')?
40  ('(' 'isStub' isStub=Boolean ')')?
41  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
42  '(' 'parentType' parentType=IntegerReference ')'
43  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
44  ')';
45
46 FAMIXMethod:
```

```
47   '(FAMIX.Method'
48   '(' 'id: ' name=INT_ID ')'
49   '(' 'name' value=MSESTRING ')'
50   ('(' 'cyclomaticComplexity' cyclomaticComplexity=INT ')')?
51   ('(' 'declaredType' declaredType=IntegerReference ')')?
52   ('(' 'hasClassScope' hasClassScope=Boolean ')')?
53   ('(' 'isStub' isStub=Boolean ')')?
54   ('(' 'kind' kind=MSESTRING ')')?
55   ('(' 'modifiers' modifiers+=MSESTRING* ')')?
56   ('(' 'numberOfStatements' numberOfStatements=INT ')')?
57   '(' 'parentType' parentType=IntegerReference ')'
58   '(' 'signature' signature=MSESTRING ')'
59   ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
60   ')';
61
62 FAMIXInheritance:
63   '(FAMIX.Inheritance'
64   '(' 'id: ' name=INT_ID ')'
65   ('(' 'previous' previous=IntegerReference ')')?
66   '(' 'subclass' subclass=IntegerReference ')'
67   '(' 'superclass' superclass=IntegerReference ')'
68   ')';
69
70 IntegerReference:
71   '(' 'ref: ' ref=[FAMIXElement|INT_ID] ')';
```

Listing 2.2: A subset of the Xtext grammar of Famix.

In Line 1, the name of the language is declared. This declaration requires that the grammar file is named `Famix.xtext` and placed in the package `org.svis.xtext`. The keyword `with` references an existing language, `Terminals.xtext`, that is also required for this language. This grammar mixin provides definitions for `INT`, `ID`, and `STRING`. Further, Ecore is `imported` in Line 2 and used for primitive type definitions. The `generate` statement in line 3 creates an `EPackage` named `famix` with the namespace Uniform Resource Identifier (URI) `http://www.svis.org/xtext/famix`. This empty `EPackage` will be extended by `EClasses` with `EAttributes` and `EReferences` defined by the subsequent parser rules. In this listing, there are nine parser rules in lines 5 to 71.

According to this grammar or metamodel, every Famix model starts with a `Root` element containing one or none `Document`. The `Document` holds a list of `FAMIXElements`. Each `FAMIXElement` has a name. `FAMIXElements` may be of type `FAMIXNamespace` for packages, `FAMIXClass` for classes, `FAMIXAttribute` for attributes, `FAMIXMethod` for methods, and `FAMIXInheritance` for inheritance relations. All relations between these elements are modeled using the element `IntegerReference`. For example, parent packages are resolved by the references `parentScope` and `ref`.

This grammar is then handed over to the language generator with a MWE2 configuration presented in extracts in Listing 2.3. The complete MWE2 configuration can be found in Listing A.1.

```
1  module org.svis.xtext.GenerateFamix
2
3  import org.eclipse.emf.mwe.utils.*
4  import org.eclipse.xtext.generator.*
5  import org.eclipse.xtext.ui.generator.*
6
7  var grammarURI = "classpath:/org/svis/xtext/Famix.xtext"
8  var fileExtensions = "famix"
9  var projectName = "org.svis.xtext.famix"
10 var runtimeProject = "../${projectName}"
11 var generateXtendStub = true
12
13 Workflow {
14    bean = StandaloneSetup {
15     scanClassPath = true
16     platformUri = "${runtimeProject}/.."
17    }
18    component = DirectoryCleaner {
19     directory = "${runtimeProject}/src-gen"
20    }
21    //[...]
22    component = Generator {
23     pathRtProject = runtimeProject
24     pathUiProject = "${runtimeProject}.ui"
25     pathTestProject = "${runtimeProject}.tests"
26     projectNameRt = projectName
27     projectNameUi = "${projectName}.ui"
28     language = auto-inject {
29      uri = grammarURI
30      // Java API to access grammar elements (required by several other fragments)
31      fragment = grammarAccess.GrammarAccessFragment auto-inject {}
32      // generates Java API for the generated EPackages
33      fragment = ecore.EMFGeneratorFragment auto-inject {}
34      // serializer 2.0
35      fragment = serializer.SerializerFragment auto-inject {
36       generateStub = false
37      }
38      // The antlr parser generator fragment.
39      fragment = parser.antlr.XtextAntlrGeneratorFragment auto-inject {}
40      // Xtend-based API for validation
41      fragment = validation.ValidatorFragment auto-inject {}
42      // [...]
43     }
44    }
45 }
```

Listing 2.3: Language generator for Famix.

The keyword `module` in Line 1 declares the fully qualified name for the language generator's class `GenerateFamix`. Lines 3 to 5 `import` required Java classes. Lines 7 to 11 declare attribute values with the keyword `var`. The main workflow is defined in lines 13 to 45. It consists of a bean to initialize the workflow (lines 14–17), directory cleaning components (lines 18–21), and a generator component (lines 22–44). The generator component includes some properties and predefined fragments, e.g. for the Famix API (Line 31), model inference (Line 33), serialization (lines 35–37), parsing (Line 39), and validation (Line 41).

After processing this grammar by the language generator from Listing 2.3, one of the outputs is the inferred Ecore model, respectively the class diagram of the Famix metamodel in Figure 2.9. The result of this process is a complete language infrastructure for Famix models including an API and components to parse, serialize, and validate these models.

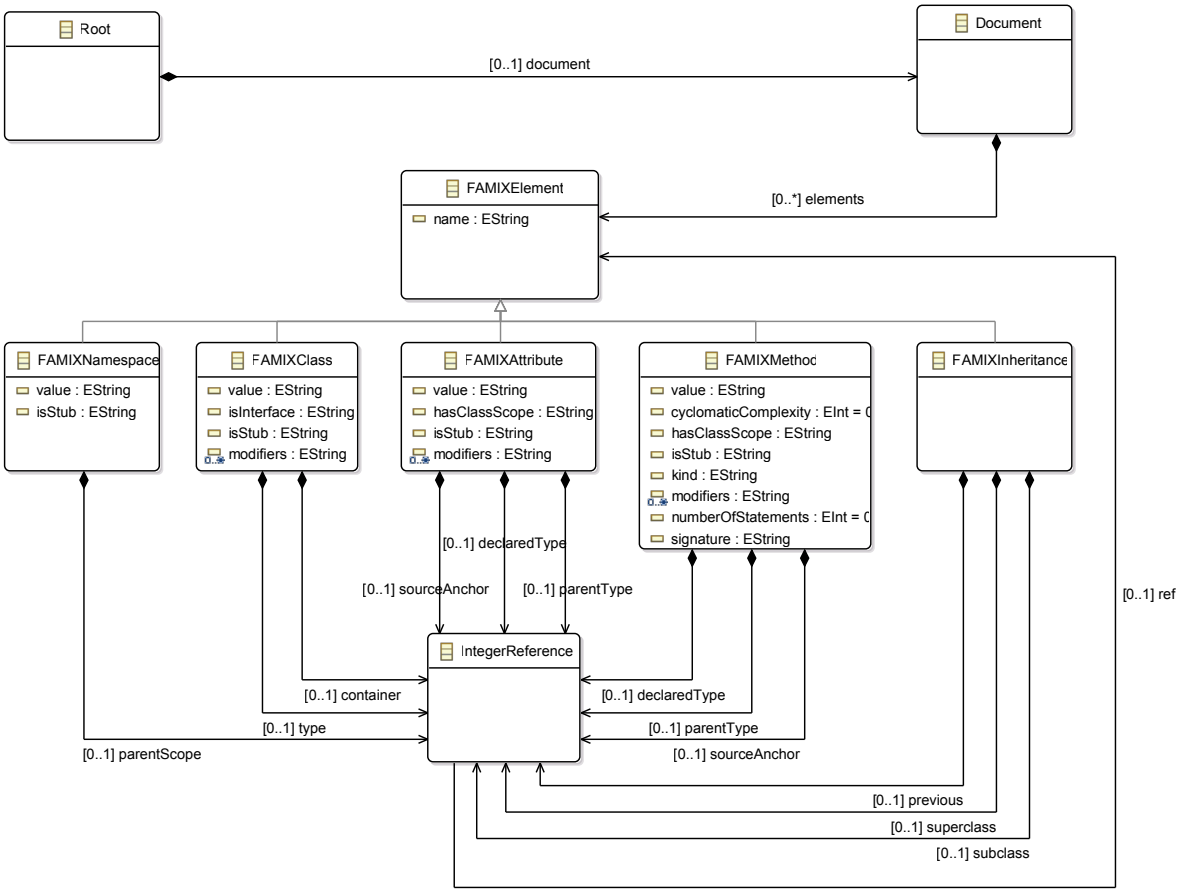


Figure 2.9: Subset of the Famix metamodel.

2.3.4 Xtend 2

Xtend is a statically-typed programming language that translates to comprehensible Java source code [Xtend Documentation 2014]. It is closely integrated with the Eclipse JDT providing IDE features like call-hierarchies, refactoring, and debugging. Concerning syntax and semantics Xtend is quite similar to Java. However, there are some improvements over Java, such as:

- extension methods,
- lambda expressions,
- active annotations,
- operator overloading,
- switch expressions,
- multiple dispatch, i.e., polymorphic method invocation,
- template expressions with intelligent white space handling,

- no statements as everything is an expression,
- properties for accessing and defining getters and setter,
- type inference,
- full support for Java generics, and
- it translates to Java.

In the context of the generator, Xtend's extension methods and the template expressions are used to describe M2Ms and M2Ps. Models conforming to an Xtext grammar are read by a generated parser component and modified or transformed by Xtend components. In the case of a M2M the models are written by a generated serializer component. In the case of a M2P the models are written directly according to a template definition. Similar to Xtext, the complete workflow is configured with MWE2.

2.4 Extensible 3D

X3D and X3DOM are two complementary approaches to create and render 2D and 3D scenes. X3DOM is build upon the X3D standard with the objective to render these scenes in a web browser without requiring additional plug-ins. In the following, both approaches are introduced. Comprehensive descriptions of X3D are given by Brutzman and Daly [2007] and descriptions of X3DOM are given by Behr et al. [2009, 2010, 2011, 2012].

2.4.1 X3D

X3D is an XML-based file format and runtime architecture to represent 3D scenes [X3D Website 2014]. It evolved from its predecessor Virtual Reality Modeling Language (VRML). X3D is royalty-free and since 2004 an International Organization for Standardization (ISO) ratified standard [X3D Standard 2014]. The development is managed by the Web3D Consortium. All subsequent statements refer to version 3.3 of the XML Schema Definition (XSD) for X3D [X3D Schema 2014].

Features

X3D provides a set of componentized features that can be tailored for use in several areas of applications, such as in engineering, in information as well as scientific visualization, in education, and in entertainment [X3D Website 2014]. The following features are supported: 3D graphics, 2D graphics, animation, spatialized audio and video, user interaction, navigation, user-defined objects, scripting, networking, physical simulation, geospatial positioning, Computer-aided Design (CAD) geometry, layering, support for programmable shaders, and particle systems.

Profiles, Components, and Levels

The underlying structure of X3D is modular. It is organized in nested profiles varying in functionality and complexity. A *profile* is composed by a predefined set of components [Brutzman and Daly 2007, p. 13ff]. Each component is divided into levels describing increasing capability. Every X3D node belongs to a component and varies in features depending on the component level. The specification of a profile is mandatory and the specifications of components and levels are optional.
There are five main profiles, namely `Core`, `Interchange`, `Interactive`, `Immersive`, and `Full` as well as three special profiles namely, `CADInterchange`, `MedicalInterchange`, and `MPEG-4 interactive`. The latter three profiles are not relevant in this thesis and thus excluded from further description. Figure 2.10 puts the five main profiles in a hierarchical relationship.

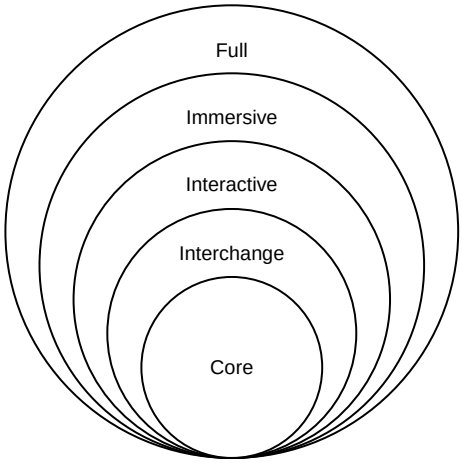


Figure 2.10: Profiles of the X3D standard [Brutzman and Daly 2007, p. 13].

All profiles build upon each other, i.e., every profile that is higher in the hierarchy includes the capabilities of its lower profile. For example, the `Interactive` profile offers all capabilities from the `Core` and the `Interchange` profile. The `Core` profile provides minimal definitions required by an X3D browser, such as the routing mechanism and meta-data. It acts as a base level so that an author can build minimally defined scenes specifying the required components and levels explicitly. The `Interchange` profile is the base for the representation and exchange of geometric models providing material, texture, lightening, and animation functionality. The `Interactive` profile adds the nodes necessary for users to interact with the scene, such as sensors. The `Immersive` profile adds audio, scripting, 2D geometry, environmental effects, and event utilities. This profile most closely matches the VRML 97 standard. The `Full` profile includes all nodes available in the X3D standard and covers advanced areas, as for example humanoid animation and geospatial components. Table 2.3 gives an overview of the supported components with corresponding levels in each profile.

Table 2.3: Overview of profiles, components, and levels in X3D version 3.3 [X3D Standard 2014].

Component	Interchange Profile Supported Levels	Interactive Profile Supported Levels	Immersive Profile Supported Levels	Full Profile Supported Levels
Core	1	1	2	2
Time	1	1	1	2
Networking	1	2	3	3
Grouping	1	2	2	3
Rendering	3	3	3	5
Shape	1	1	2	4
Geometry3D	2	3	4	4
Geometry2D			1	2
Text			1	1
Sound			1	1
Lighting	1	2	2	3
Texturing	2	2	3	3
Interpolation	2	2	2	5
Pointing device sensor		1	1	1
Key device sensor		1	2	2
Environmental sensor		1	2	3
Navigation	1	1	2	3
Environmental effects	1	1	2	4
Geospatial				2
Humanoid animation				1
Non-uniform Rational B-Spline (NURBS)				4
Distributed interactive simulation (DIS)				2
Scripting			1	1
Event utilities		1	1	1
Programmable shaders				1
CAD geometry				2
Texturing 3D				2
Cube map environmental texturing				3
Layering component				1

Continued on next page

Table 2.3 – continued from previous page

Component	Interchange Profile Supported Levels	Interactive Profile Supported Levels	Immersive Profile Supported Levels	Full Profile Supported Levels
Layout component				2
Rigid body physics component				2
Picking sensor component				3
Followers component				1
Particle systems component				3
Volume rendering component				4

Encoding

X3D supports three file formats. First, the XML-based format with the suffix `*.x3d`. Second, the VRML syntax with the suffix `*.x3dv`. Finally, the binary format with the suffix `*.x3db`.

Conventions

There are two important conventions in X3D concerning coordinate system and units of measurement [X3D Standard 2014]. Every scene has a three-dimensional, Cartesian, right-handed coordinate system. For this reason, rotations perform in a mathematical positive sense, i.e., counter-clockwise. The units of measurement for length, angle, time and color are listed in Table 2.4.

Table 2.4: Units of measurement in X3D.

Category	Unit of Measurement
Length	Meter
Angle	Rad
Time	Seconds
Color	RGB(0.0–1.0, 0.0–1.0, 0.0–1.0)

Scene Graph

A *scene graph* is the basic unit of the X3D runtime environment. It is a directed, acyclic graph or a tree, respectively [Brutzman and Daly 2007, p. 1]. The nodes of this tree correspond to objects of the scene. All objects are positioned in the virtual world according to the transformation hierarchy. This hierarchy describes the spatial relationships of the objects.

Further, the behavior graph describes the connections between fields of nodes and the flow of events through the system.

Runtime Architecture

The interpretation, execution, and representation of a scene is realized by the X3D browser. The browser reads a scene description and parses its content. The created nodes are transferred to the scene graph manager which renders the nodes with corresponding geometry, appearance, position, and orientation. Further, the manager is able to receive events from animation or script nodes manipulating the scene. The Abstract Scene Access Interface (SAI) provides means to access nodes of a scene at runtime. Figure 2.11 outlines the architecture of X3D.

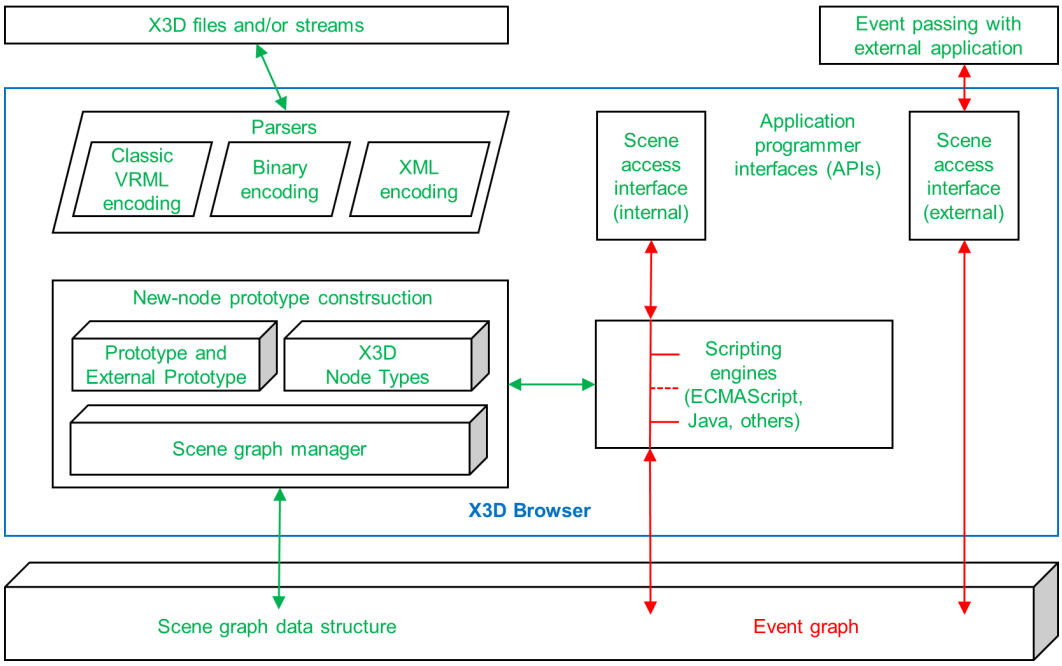


Figure 2.11: X3D architecture [X3D Standard 2014].

Viewer

The resulting scene can be viewed in a web browser with corresponding plug-ins or standalone with a player. A list of available plug-ins and players can be found at [X3D Website 2014]. The InstantPlayer was used for testing during development. This viewer is part of the InstantReality platform developed by the Fraunhofer Institute for Computer Graphics [InstantReality Website 2014].

Example with X3D

To illustrate the usage of X3D, a simple example is used. Listing 2.4 shows the XML-based scene description with meta-data to create the slightly rotated red box in Figure 2.12 rendered by the InstantPlayer.

```
1  <X3D version='3.3' profile='Interactive'>
2    <head>
3      <!--<component name='Geometry2D' level='2'/>-->
4      <meta name='title' content='x3d_box.x3d'/>
5      <meta name='author' content='Richard Mueller'/>
6      <meta name='created' content='2014-10-09'/>
7    </head>
8    <Scene>
9      <Transform translation='0 0 0' rotation='1 1 0 0.78'>
10       <Shape>
11         <Appearance>
12           <Material diffuseColor='1 0 0'/>
13         </Appearance>
14         <Box/>
15       </Shape>
16     </Transform>
17   </Scene>
18 </X3D>
```

Listing 2.4: X3D example.

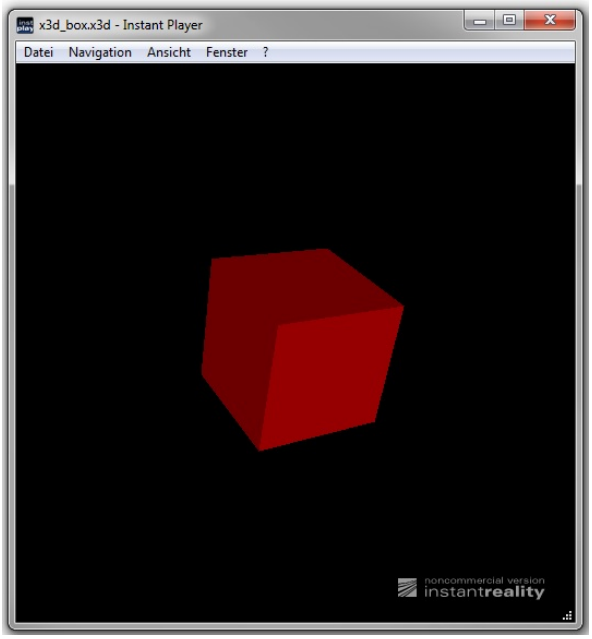


Figure 2.12: X3D example.

In Line 1, the `X3D` root node is defined. The fields of this root node indicate that the `Interactive` profile and version 3.3 of the standard are required. Lines 2 to 7 cover further metadata of the scene in the `head` node. Here, additional components and levels, not supported by the current profile, can be specified with the `component` node. Additionally, `meta` nodes are used to add information about title, author, and date of creation. The `Scene` node in lines 8 to 17 introduces the actual scene graph definition. The `Transform` node places the `Shape` node at the origin of the coordinate system (x=0, y=0, and z=0) and rotates it by approximately 45 degree (0.78 rad) around the x- and y-axes. The `Material` node sets the color of the shape to red. Finally, the form of the shape is defined in Line 14, a box.

2.4.2 X3DOM

X3DOM is an open-source framework and runtime architecture for 3D graphics on the web [X3DOM Website 2014]. The main objective of X3DOM is the integration of declarative X3D content in the Hypertext Markup Language (HTML) DOM tree [Behr et al. 2009, 2010]. However, it is more an intermediate solution until X3D becomes an integral part of HTML. Figure 2.13 shows the ideas and main differences behind both concepts. Whereas X3D is only runnable in a web browser with the help of additional plug-ins, X3DOM seamlessly integrates the X3D content in the web browser's DOM tree. Consequently, no additional plug-ins are necessary to run a 3D scene in a web browser. All subsequent statements refer to version 1.6.2 of X3DOM.

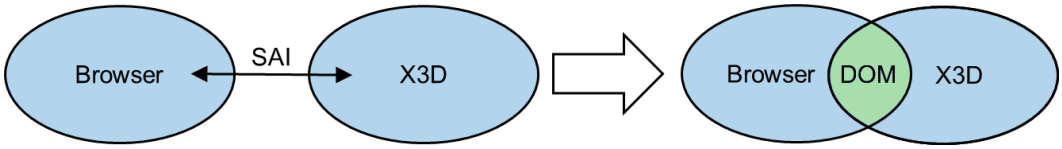


Figure 2.13: Moving from X3D to X3DOM [Behr et al. 2009].

Profile: HTML

X3DOM introduces a new profile, called `HTML`, to provide a subset of X3D that matches the needs of modern HTML applications [Behr et al. 2010]. This profile extends the `Interchange` profile with a level increase in the components `Networking`, `Grouping`, and `Navigation`. Thus, the nodes `Inline`, `Switch`, `StaticGroup`, `Billboard`, and `LOD` are supported. The `HTML` profile does neither include `Script` nodes nor does it provide support for prototypes as developers are supposed to script and partition the content from the DOM/HTML side [X3DOM HTML Profile 2014].

Encoding

X3DOM only supports the XML-based encoding directly [Behr et al. 2010]. However, some backends, for instance the X3D/SAI plug-in, support classical and binary encodings indirectly using `Inline` nodes.
If an Extensible Hypertext Markup Language (XHTML) encoded document is the base for the 3D scene, the original X3D content can be embedded with no changes. That means, upper-case names and self-closing tags can be used. On the contrary, if an HTML encoded document is the base, lower-case names have to be used and no self-closing tags are supported.

Runtime Architecture

The main objective of X3DOM is to render an X3D scene in the HTML DOM allowing the developer to add, to remove, or to change DOM elements [Behr et al. 2010]. This is possible

without additional plug-ins or plug-in interfaces, such as the SAI. Further, HTML events, e.g., `onclick`, `onload`, or `onmouseover`, are supported. As depicted in Figure 2.14, the three main building blocks are the User Agent (UA), the X3D runtime, and the X3DOM connector.

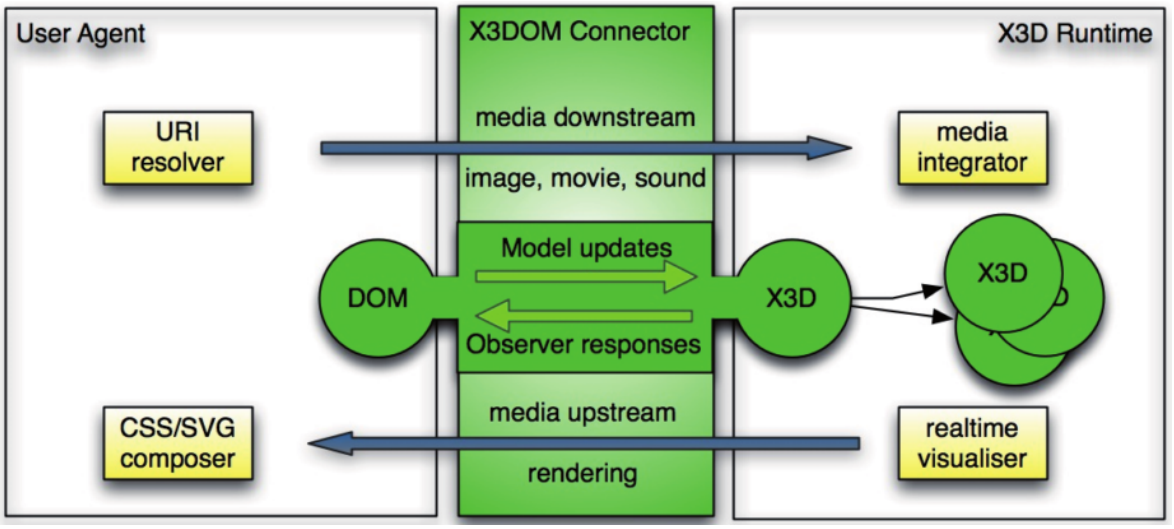


Figure 2.14: X3DOM architecture [Behr et al. 2010].

The UA, i.e., the web browser, holds the DOM tree, integrates and composes the final rendering. Furthermore, it provides an URI resolver to download images, movies, and sounds of the scene. The X3D runtime offers services to create and update the scene graph. It is responsible for scene rendering and for user input handling regarding navigation and picking. The X3DOM connector is the main unit of the architecture. It connects the DOM tree with the X3D runtime. In this role, it distributes relevant changes in both directions, i.e. model updates from the DOM tree to the scene graph and user inputs in the other direction. Further, it handles any media up- and downstream.
The general architecture is implemented as a JavaScript layer (`x3dom.js`). The integration of this layer allows the developer to insert X3D sections in the HTML or XHTML document. This layer acts as a connector and synchronizer that monitors DOM changes and thus updates the X3D structures and vice versa. For this purpose, the system does not use a single X3D runtime but a fallback model to pick the best environment for the given circumstances.

Fallback Model

The fallback model, illustrated in Figure 2.15, realizes the instantiation of different backend technologies for the X3D runtime. Every backend has specific feature and performance criteria that decrease with every alternative. The model supports native implementations, X3D/SAI plug-ins, Web Graphics Library (WebGL), and Flash. With this model it is possible to run X3D scenes in a browser without additional plug-ins.

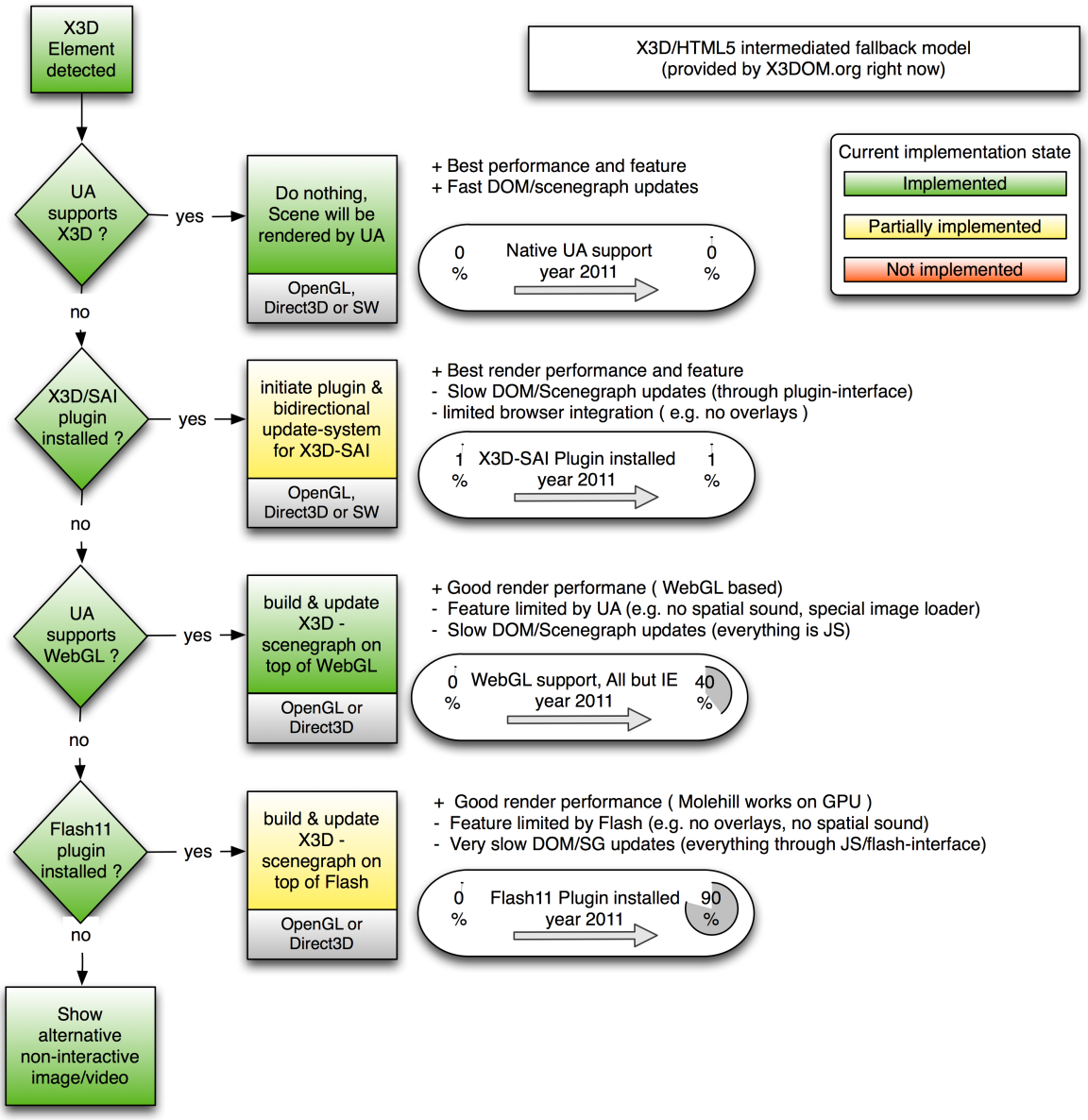


Figure 2.15: X3DOM fallback model [X3DOM Fallback Model 2014].

Example with X3DOM

Listing 2.5 uses the same X3D scene from Listing 2.4 embedded in an HTML document and loaded by a web browser using X3DOM. The resulting red box is shown in Figure 2.16.

```
1  <html>
2    <head>
3      <title>X3DOM example</title>
4      <script type='text/javascript' src='http://www.x3dom.org/download/x3dom.js'> </script>
5      <!--<script src='http://www.x3dom.org/download/1.6.2/components/Geospatial.js'></script>-->
6      <!--<script src='http://www.x3dom.org/download/x3dom.swf'></script>-->
7      <link rel='stylesheet' type='text/css' href='http://www.x3dom.org/download/x3dom.css'></link>
8    </head>
9    <body>
10     <h1>X3DOM example</h1>
11     <p>This is a html page with a slightly rotated red cube.</p>
12     <x3d width='600px' height='400px'>
```

```
13      <scene>
14        <transform translation='0 0 0' rotation='1 1 0 0.78'>
15          <shape>
16            <appearance>
17              <material diffuseColor='1 0 0'></material>
18            </appearance>
19            <box></box>
20          </shape>
21        </transform>
22      </scene>
23    </x3d>
24  </body>
25 </html>
```

Listing 2.5: X3DOM example.

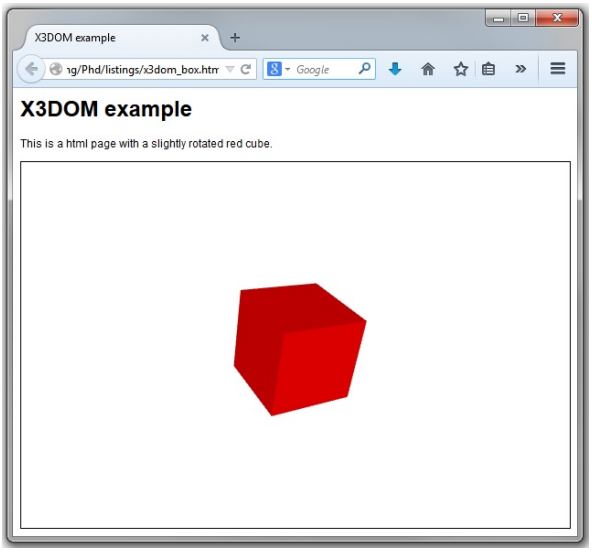


Figure 2.16: X3DOM example.

As in every HTML or XHTML document, the root element `html` consists of a `head` and a `body`. The X3DOM-specific elements in the `head` are in lines 4 and 7. Here, the last stable X3DOM release (`x3dom.js`) and the corresponding Cascading Style Sheet (CSS) file (`x3dom.css`) are included using `script` and `link` elements. If an additional component is required, it can be referenced explicitly by specifying the X3DOM version, as shown in Line 5. In order to provide Flash support, the reference to `x3dom.swf` is necessary, as depicted in Line 6. The actual scene graph is embedded within the `body` in lines 12 to 23. The differences to Listing 2.4 are due to the encoding. In this example the document is HTML encoded. Thus, the node names are lower-case and no self-closing tags are used. Additionally, the `width` and the `height` of the scene can be defined in the `x3d` root node. Apart from that, it is the same scene as in the previous X3D example.

AOPT

As real 3D scenes can increase in size very fast, an optimization is necessary to guarantee acceptable runtime and interaction performance. X3DOM provides a non-standardized, op-

timized `BinaryContainer` node [Behr et al. 2012]. Instead of a text-based description of the scene graph, it uses more compact, external binary files. The command line tool AOPT is part of the InstantReality platform [InstantReality Website 2014] and converts between the different file encodings, transforms any input into an X3DOM HTML/ XHTML document, analyzes 3D scenes, and creates optimized binary files [AOPT 2014].
However, the binary container is only a temporary solution with prototypical character. The X3DOM developers intend to replace it with the `ExternalGeometry` node from the Shape Resource Container (SRC) [Limper et al. 2014]. The main objective of SRC is to design a solution that scales for very large 3D scenes by enabling a progressive transmission of mesh data, by eliminating decode time through direct Graphics Processing Unit (GPU) uploads, and by minimizing the number of Hypertext Transfer Protocol (HTTP) requests.

2.5 Summary

In this chapter, the theoretical and technical foundations of the thesis are introduced, including software visualization, GP and MDSD, Eclipse as well as X3D and X3DOM.
The view of software visualization represented here follows the definition of Diehl [2007, p. 3f]. Thus, software visualization is the visualization of artifacts related to software and its development process. The main objective is to provide role- and task-specific views on these artifacts. Additionally, the artifacts may represent structural, behavioral, and/or evolutionary aspects of a software system.
The generative [Czarnecki and Eisenecker 2000] and the model-driven [Stahl et al. 2006] paradigm from the field of software engineering are combined and used for the theoretical concept of the software visualization generator. The generator should be able to accept structural, behavioral, and evolutionary models of a software system. These models have to be transformed into role- and task-specific software visualizations. The desired requirements of these visualizations are specified by a means of a DSL.
The generator is developed in the Eclipse IDE using JDT, PDE, TMF, and EMP. Thus, it has a modular, extensible plug-in architecture and is closely integrated with Eclipse. The technology projections of the elements of the generative domain model, are Xtext, Xtend, and X3D in combination with X3DOM. The problem space is defined by Famix [Ducasse et al. 2011], Dynamix [Greevy 2007], and Hismo [Ducasse et al. 2004], metamodels for describing structural, behavioral, and evolutionary aspects of software systems. The configuration knowledge, i.e., the generator, is realized with Xtend transformations. The solution space provides X3D components that are optimized for the web with AOPT and finally visualized with X3DOM. Hence, the generator accepts structural models of software systems and a DSL as input and transforms the models into 2D, 2.5D, or 3D scenes that can be interactively explored in a browser without plug-ins on any platform.

3 Literature Study

Müller, Richard, and Dirk Zeckzer. 2015. “Past, Present, and Future of 3D Software Visualization - A Systematic Literature Analysis.” In *Proceedings of the 6th International Conference on Visualization Theory and Applications*, Berlin, Germany.

3.1 Past, Present, and Future of 3D Software Visualization - A Systematic Literature Analysis

Past, Present, and Future of 3D Software Visualization
A Systematic Literature Analysis

Richard Müller[1], Dirk Zeckzer[2]
[1]*Information Systems Institute, Leipzig University, Leipzig, Germany*
[2]*Institute of Computer Science, Leipzig University, Leipzig, Germany*
rmueller@wifa.uni-leipzig.de, zeckzer@informatik.uni-leipzig.de

Keywords: 3D, Software Visualization, Systematic Mapping Study, Systematic Literature Review

Abstract: The ongoing 2D vs. 3D research debate from information visualization also affects software visualization. There are many 2D, 3D, and combinations of 2D and 3D visualizations for software representing its structure, behavior, or evolution. This study contributes findings to this debate and presents the results of analyzing the applications of 3D in software visualization with the objectives to outline the state-of-the-art, to reveal trends, and to identify research gaps. The analysis combined a systematic mapping study to get an overview and a systematic literature review to gain deeper insights. The relevant papers were identified by three different search strategies (manual browsing, keyword, and backward search). Starting with a set of 4386 publications from the fields of information and software visualization 155 relevant papers dealing with 2D & 3D or 3D software visualizations were identified. These papers were analyzed according to dimensionality, aspect, year, evaluation method, and application of the third dimension. In a nutshell, the majority of 3D software visualizations represents the structural aspect, is either evaluated using case studies showing working examples or not evaluated at all, and applies a 2D layout using the third dimension for displaying software metrics.

1 INTRODUCTION

As a branch of information visualization, software visualization provides tools and methods to create representations for structural, behavioral, and evolutionary aspects of software systems (Diehl, 2007). Comprehensive surveys with numerous visualizations ranging from 2D to 3D were performed by Gračanin et al. (2005), Teyseyre and Campo (2009), and Caserta and Zendra (2011). However, as in its parental discipline, there is an ongoing 2D vs. 3D debate. This study aims at investigating the use of 3D in software visualization and how its usefulness is evaluated.

A suitable approach for this investigation are systematic mapping studies and literature reviews. A systematic mapping study aims at building a classification scheme in order to structure a research field (Petersen et al., 2008). The scheme comprises facets detailed by categories. The different facets are combined to answer specific research questions. The results include frequencies of publications for each category within this scheme. The systematic literature review focuses on a deeper analysis of the publications and can have other goals (Brocke et al., 2009). Petersen et al. (2008) argue that both methods can be applied complementary. Thus, we used the mapping study to gain an overview of the field and investigated specific questions using detailed reviews.

The major contributions of this state-of-the-art report in 3D software visualization are answers to the following questions:

- *Venue:* Where were papers about 3D software visualization published?
- *Aspect:* Which aspects of software are visualized?
- *Evolution:* How did the topic evolve over the last 22 years?
- *Evaluation:* How was the usefulness of the 3D software visualizations evaluated?
- *Application:* How was the third dimension used?

On the basis of these answers trends were revealed and research gaps identified.

2 RELATED WORK

Important prior work ranges from meta-studies and surveys to literature reviews as well as a mapping study in the field of software visualization.

Hundhausen conducted two meta-studies, one about software visualization effectiveness (Hundhausen, 1996) and one about algorithm visualization effectiveness (Hundhausen et al., 2002). The classification of the evaluation methods into anecdotal, analytic, and empirical is taken from these studies.

Gračanin et al. (2005) provide a general overview over software visualization outlining several research directions, such as (distributed) virtual environments and visualization metaphors. Teyseyre and Campo (2009) give a comprehensive overview over 3D software visualization including visual representations, interaction issues, evaluation methods, and development tools. Caserta and Zendra (2011) focus on static aspects of software visualization in 2D and 3D. We used all three surveys as a starting point for the backward search to find relevant papers not covered by the selected workshops and conferences in our primary studies.

Kienle and Müller (2007) identified quality attributes and functional requirements for software visualization tools to support researchers using a literature review. Schots and Werner (2014) examined software visualizations with regard to reuse based on the task oriented taxonomy from Maletic et al. (2002). The complete review can be found here (Schots et al., 2014). Seriai et al. (2014) investigated the state-of-the-art in validation of software visualization tools with a mapping study. The primary categories of the evaluation method facet are taken from this study. The main difference to our study is the focus: we concentrate on 3D software visualizations including all aspects, such as structure, behavior, and evolution. In this context, we investigate publication locations, evaluation methods, the development of this specific field over time, as well as the application of the third dimension.

3 METHOD

For this study, a hybrid approach was applied combining a systematic mapping study (Petersen et al., 2008) with a systematic literature review (Brocke et al., 2009). First, a mapping study was performed to get an overview and to answer the first four research questions. Second, a detailed literature review was conducted to answer the fifth research question. Finally, the results of both processes are summarized in the findings. The complete process is depicted in Figure 1. Its steps will be described in the subsequent sections.

3.1 Define Scope & Research Questions

We describe the scope of this study according to Cooper's taxonomy of literature reviews (Cooper, 1988). The *focus* lies on applications of 3D software visualizations. Our *goal* is to integrate findings from publications of different workshops/conferences/journals to create a comprehensive view of this topic. The study is *organized* conceptually guided by a classification scheme. Further, we adopt a neutral *perspective*. The main *audience* are specialized scholars from the fields of information visualization and software visualization. The coverage is aimed to be representative as we combine manual browsing through relevant workshop and conference proceedings, a keyword search, and a backward search starting with state-of-the-art-papers.

With this study, we want to investigate the following research questions:

- **RQ1:** Which workshops/conferences/journals include papers on 3D software visualization?
- **RQ2:** Which aspects of software (structure, behavior, evolution) are visualized with 3D?
- **RQ3:** How did 3D software visualization evolve over the last 22 years and what are current trends?
- **RQ4:** How is the usefulness of the proposed 3D software visualizations evaluated?
- **RQ5:** How is the third dimension used?

3.2 Conceptualize Topic

For the classification scheme, a top-down and bottom-up approach were applied. We started with established definitions from literature for the classification of the relevant papers. If a paper introduces a new category, the corresponding facet in the classification scheme was extended. In this study, the following facets are important: *dimensionality*, *aspect*, *year*, *evaluation method*, and *application of the third dimension*. The categories for each facet are summarized in Table 1 and described next.

3.2.1 Dimensionality

We differentiate between *2D*, combined *2D and 3D*, and *3D* software visualizations. For this study, the last two categories are focused.

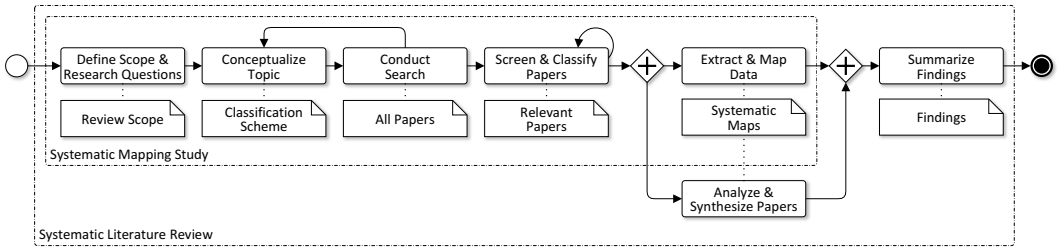


Figure 1: Process model for hybrid approach: Mapping study (Petersen et al., 2008) and literature review (Brocke et al., 2009).

3.2.2 Aspect

The different aspects of software that can be visualized are based on Diehl (2007). He defines software visualization as *"[...] the visualization of artifacts related to software and its development process."*. These artifacts can contain information about the *structure*, the *behavior*, or the *evolution* of the software system. Structure includes program code, data structures, the static call graph, relations, and the organization of the software system. Behavior covers its execution with real and abstract data. Evolution refers to its development process.

Table 1: Classification scheme for the study.

Facet		Category
Dimensionality		2D and 3D 3D
Aspect		Structure Behavior Evolution
Year		1991—2013
Evaluation Method	Anecdotal	Case Study (Example)
	Empirical	Case Study (User) Controlled Experiment Questionnaire
	Analytic	Guideline Checking Heuristic Evaluation
Application		Extended 2D Full 3D 2D layout org. in 3D 3D as time Stacked views 3D for cognition Local fish-eye

3.2.3 Year

The years for the relevant papers range from 1991 until 2013. This period results from the search strategies described in Section 3.3.

3.2.4 Evaluation Method

Typical evaluation methods in software visualization are *case study*, *controlled experiment*, and *questionnaire* (Sjøberg et al., 2007; Seriai et al., 2014). However, the term case study is used in two different ways in software visualization. On the one hand, a case study is the demonstration of a working example as in (Wettel and Lanza, 2008). This type of case study is without representative users. On the other hand, a case study actually involves representative users as in (Denford et al., 2002). The second type also includes explorative user studies as in (Lanza et al., 2013). Thus, we differentiate between *case study (example)* and *case study (user)*. In addition, we found *guideline checking* and *heuristic evaluation* described in (Andrews, 2008). All methods can further be classified into anecdotal, empirical, and analytic evaluation methods (Hundhausen, 1996; Hundhausen et al., 2002). Anecdotal methods use compelling examples, empirical methods involve representative users, and analytic methods are performed by evaluation experts using guidelines or heuristics.

3.2.5 Application of the third dimension

Reiss (1995) identified six different categories for the application of the third dimension. As there were papers not fitting in any of these categories, we introduced another one resulting in the following seven categories.

1. *Extended 2D:* A 2D layout is extended to 3D resulting in an additional dimension. This dimension can be used to display further information, such as software metrics as done in sv3D (Mar-

cus et al., 2003) or CodeCity (Wettel and Lanza, 2007).

2. *Local fish-eye:* Another technique builds upon a 2D layout where the user is able to select a set of nodes and place them at the front. This technique uses perspective to make the selected nodes appear bigger and the other ones smaller. It results in local fish-eye views without changing the original graph such as in rubber sheet (Sarkar et al., 1993).
3. *2D layout organized in 3D:* The third technique takes a 2D layout and the information is organized in a 3D space, such as with cone and cam trees (Robertson et al., 1991) or with hyperbolic trees (Munzner, 1997). It is usually applied to get more space and to minimize edge-crossings. Two other examples for this category are the perspective wall (Mackinlay et al., 1991) and the 'code on the wall' metaphor (Jackson et al., 2002).
4. *Full 3D:* The next technique moves from 2D to 3D space and uses the full capabilities of three dimensions. Examples are Angle (Churcher and Tech, 2003), Metaballs (Rilling and Mudur, 2005), and the 3D scatter plot in ComVis (Bohner et al., 2007).
5. *3D as time:* Further, the third dimension is used to represent time, such as in VRCS (Koike and Chu, 1998), Vizz3D (Löwe and Panas, 2005), or Palantír (Ripley et al., 2007).
6. *Stacked views:* This technique uses the third dimension to display several 2D views simultaneously. Examples are 3D sequence diagram as in (Gil and Kent, 1998) and GEF3D (von Pilgrim and Duske, 2008).
7. *3D for cognition:* Finally, 3D shapes are applied to support the mental model and to optimize the cognition of the visualization. Examples for this category are Geons (Irani and Ware, 2003) and the use of social agents to visualize software scenarios (Alspaugh et al., 2006).

3.3 Conduct Search

We combined three search methods in order to make the sample more representative. First, we browsed manually through all publications from relevant workshops and conferences in the field of software visualization including SoftVis (2003, 2005, 2006, 2008, 2010), VisSoft (2002, 2003, 2005, 2007, 2009, 2011, 2013), IWPC/ICPC (1998-2013), Dagstuhl Seminar on Software Visualization (2001), OOPSLA Workshop on Software Visualization (2001), and ICSE Workshop on Software Visualization (2001). Second, we performed a keyword search on publications of relevant workshops and conferences in the field of information visualization including IEEE VIS (2000-2013), PacificVis (2008-2013), and EuroVis (2007-2013). The keyword was "*software visualization*". Third, we conducted a backward search using three state-of-the-art papers related to 3D software visualization: Gračanin et al. (2005), Teyseyre and Campo (2009), and Caserta and Zendra (2011).

3.4 Screen & Classify Papers

The screening process for each paper included title, abstract, conclusion, and—if necessary—further parts. We used the following inclusion and exclusion criteria to select the relevant papers. The publication is included, if

- it deals with single 2D and 3D or 3D software visualizations (this automatically excludes surveys),
- it is peer reviewed including full papers, short papers, and posters (this automatically excludes books, book chapters, technical or research reports, or white papers), and
- it is written in English[1].

The publication is excluded, if

- the third dimension only serves aesthetic purposes, i.e., augmented 2D visualizations (Stasko and Wehrli, 1993), and
- it does not deal with software visualization, e.g., network visualization (hardware) or security.

In addition, we classified all relevant papers according to the categories of the classification scheme. If the classification of a paper was not unique, it was marked, discussed by the authors, and finally included and classified or excluded. For this reason, this step has an iterative character. We used the reference management software Mendeley for screening and classifying the papers. The provided XML export was helpful for further data processing.

3.5 Extract & Map Data

Major results of systematic mapping studies are frequency/pie charts and bubble plots. Frequency/pie charts show the distribution of a variable in an absolute or relative manner. Bubble plots resemble x-y scatter plots but with bubbles in category intersections where the size of a bubble represents frequencies

[1] One paper was written in Italian, a language none of the authors is fluent in.

Table 2: Results for the three search strategies.

	Manual	Keyword	Backw.	Sum
Total	878	2998	510	4386
Dupl.	0	0	146	146
Other	405	2984	220	3609
2D	393	10	73	476
3D	80	4	71	155

of publications. We used Excel for data management and the creation of the frequency/pie charts and bubble plots or systematic maps respectively.

3.6 Analyze & Synthesize Papers

To get an overview of the application of the third dimension in software visualization, it was necessary to conduct a more detailed analysis of the relevant papers. This went beyond the screening process described above. We had to study further parts of the paper, especially sections explaining the concepts and their implementation and the provided figures. The results of this deeper review are also presented in a systematic map.

3.7 Summarize Findings

Based on the results including frequency/pie charts and the systematic maps, we deduced findings including trends and research gaps in 3D software visualization. Trends can be detected by analyzing the evolution of the topic over time. Small bubbles in the maps highlight research areas that might be underresearched.

4 RESULTS

Table 2 shows the amount of papers identified with each search strategy and Figure 2 details these results with a Venn diagram.

Overall, 4386 papers were found, manually (878), using a keyword search (2998), or using references in surveys (510). From these, 146 papers were duplicates in the backward search which yields a total of 4240 unique papers to be examined. From these, 631 papers deal with software visualization and 155 (24.6%) with 2D & 3D (41, 26.0%) or 3D only (114, 74.0%) software visualization. These were published as full papers (116, 74.8%), short papers (17, 11.0%), and posters (22, 14.2%). These 155 papers are the input to the steps 'extract & map data' as well as 'analyze & synthesize papers' and thus the basis for answering the research questions.

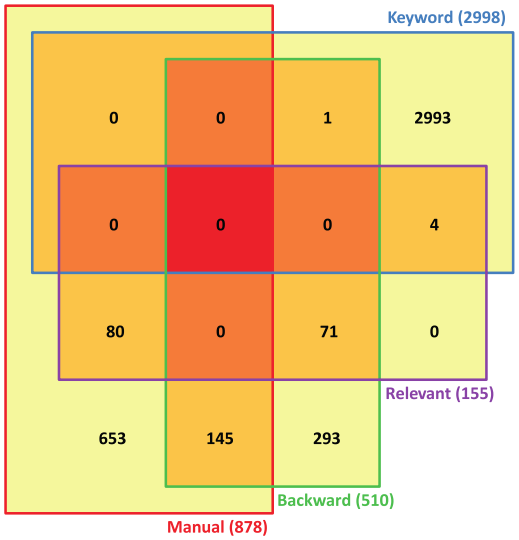


Figure 2: Results for the three search strategies as a Venn diagram.

4.1 RQ1: Which workshops/conferences/journals include papers on 3D software visualization?

Figure 3 shows all workshops, conferences, and journals including papers with 3D software visualization that were found using the method described in Section 3. We observed, that most of the 3D software visualization papers were published on VisSoft (6 events, 30 papers before 2012) and SoftVis (5 events, 24 papers).

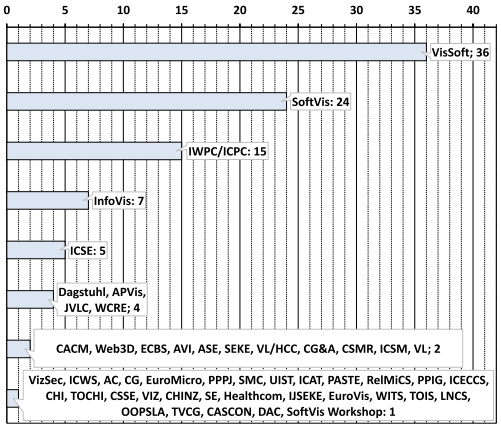


Figure 3: Workshops, conferences, and journals with 3D software visualizations.

After SoftVis and VisSoft merged in 2013, 6 papers were published on the new VisSoft 2013. Further, 4 papers emerged from the precursor of SoftVis, the Dagstuhl 2001 event. Altogether, 64 papers (41.29%) containing 3D software visualization were published on the main events.

An additional 15 papers were presented on IWPC/ICPC (16 events). On the main visualization conferences, a total of 9 papers are related to 3D software visualization (InfoVis: 7, EuroVis: 1, PacificVis: 0, related conferences/workshops: 1). For the software engineering related conferences, the count is 10 papers (ICSE: 5, WCRE: 4, SE: 1, OOPSLA: 1). Overall, these conferences contributed 34 papers (21.94%) to our study. 40 other venues added 57 papers (36.77%), with at most 4 additional papers per venue.

4.2 RQ2: Which aspects of software visualization (structure, behavior, evolution) are visualized with 3D?

Figure 4 shows the distribution of the different aspects displayed using 3D software visualization. From the 155 papers analyzed, 67 (43.2%) visualize structure alone, 54 (34.8%) structure and behavior, 18 (11.6%) structure and evolution, 6 (3.9%) structure and behavior and evolution, 5 (3.2%) behavior alone, and 5 (3.2%) evolution alone. That means, that 145 papers (93.5%) deal with structure alone or in combination with behavior and/or evolution. No 3D visualization was proposed for a combination of behavior and evolution without the aspect of structure.

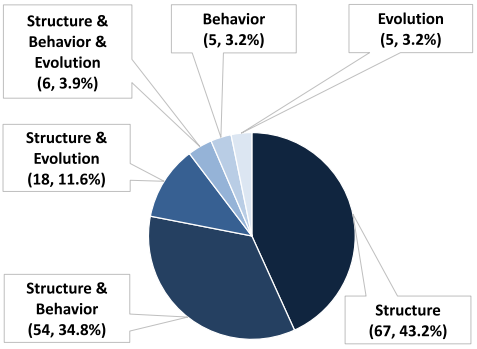


Figure 4: Aspect displayed using 3D software visualizations.

4.3 RQ3: How did 3D software visualization evolve over the last 22 years and what are current trends?

Figures 5 and 6 show the evolution of 3D software visualizations from 1991 until 2013. Few papers were found dealing explicitly with 3D software visualization before 2001. Overall, 30 papers were published between 1991 and 2000, between 1 and 6 papers per year. All papers include structural aspects.

Between 2001 and 2008, 10 papers or more were published each year on 3D software visualization, with an exception of 2006, when only 6 papers address this topic. Overall, from 2001 until 2008 two thirds (67.7%) of the found papers were published. Between 2009 and 2013, less papers were published on this topic per year—between no papers in 2012 and eight papers in 2013. Overall, between 2001 and 2013, 126 papers dealing with 3D software visualization were published. Most papers address structure (56, 36.1%) or a combination of structure and behavior (40, 25.8%), structure and evolution (13, 8.4%), and structure, behavior, and evolution (6, 3.9%). Behavior alone (5, 3.2%) and evolution alone (5, 3.2%) are rarely considered. It is remarkable, that the order of the different aspects or combinations of aspects regarding the amount of papers published stays the same, independently of the year or the amount of papers published with only few exceptions: in 1994, 1995, and 1996 only papers combining structure and behavior were published, structure and evolution (2) is ranked first in 1997 before structure alone (1) and structure and behavior (1), structure and behavior (7) is ranked first in 2001 before structure alone, structure and evolution (4) is ranked first together with structure alone (4) in 2008 before structure and behavior (3), and finally structure and behavior (3) is ranked first in 2013 before structure alone (2), structure, behavior, and evolution (2), and structure and evolution (1). However, the amount of papers including 3D software visualization is already small for each year.

4.4 RQ4: How is the usefulness of the proposed 3D software visualizations evaluated?

Figure 7 shows the different aspects of 3D software visualization and their evaluation methods. Some 3D software visualizations were evaluated using several different evaluation methods. Therefore, the total count is larger than the total number of papers analyzed.

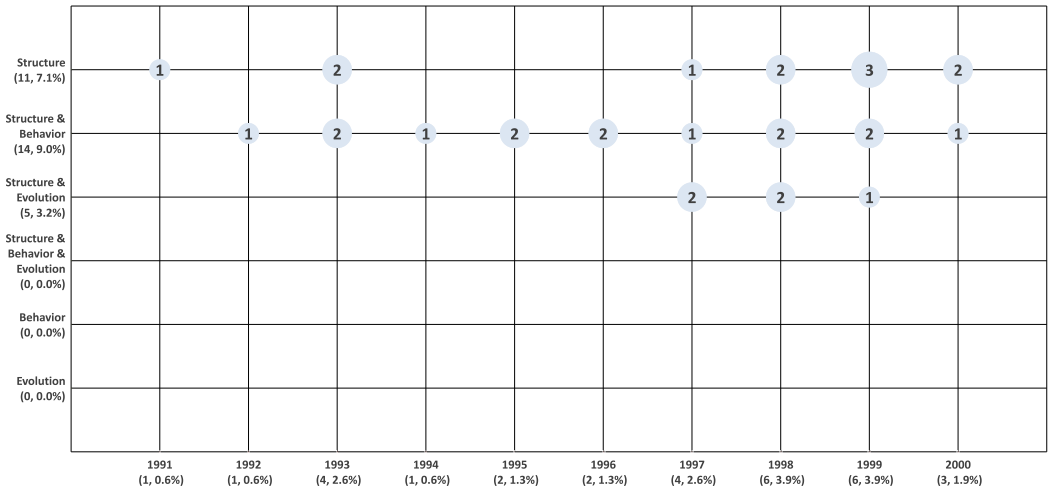


Figure 5: Time vs. aspect for 3D software visualizations (1991-2000).

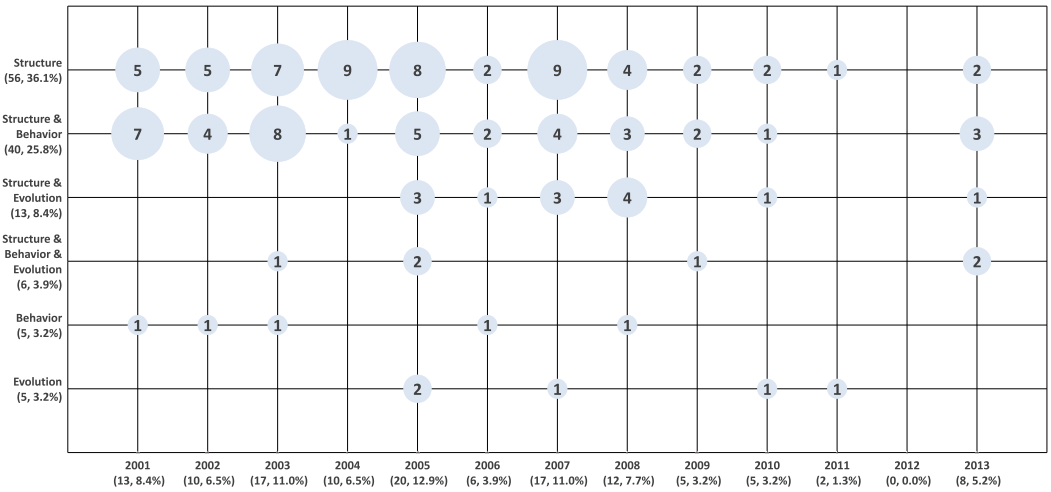


Figure 6: Time vs. aspect for 3D software visualizations (2001-2013).

The different aspects or combinations of aspects were mostly evaluated using case studies showing working examples (89, 53.3%) or not evaluated at all (27, 16.2%). Some 3D visualizations were evaluated using case studies that involve representative users (19, 11.4%). Few 3D visualizations were evaluated using controlled experiments (15, 9.0%). Other evaluation methods used were guideline checking (10, 6.0%), questionnaires (7, 4.2%), and heuristic evaluations (2, 1.2%).

With respect to the combination of aspect and evaluation method, the bubble chart does not exhibit any particularities. As most numbers are small, the difference in ratios does not provide evidence for relationships.

4.5 RQ5: How is the third dimension used?

Figure 8 shows the different aspects of 3D software visualization and their application of the third dimension. As a paper might contain multiple visualizations or a visualization might belong to different categories at the same time, the sum is larger than the number of papers.

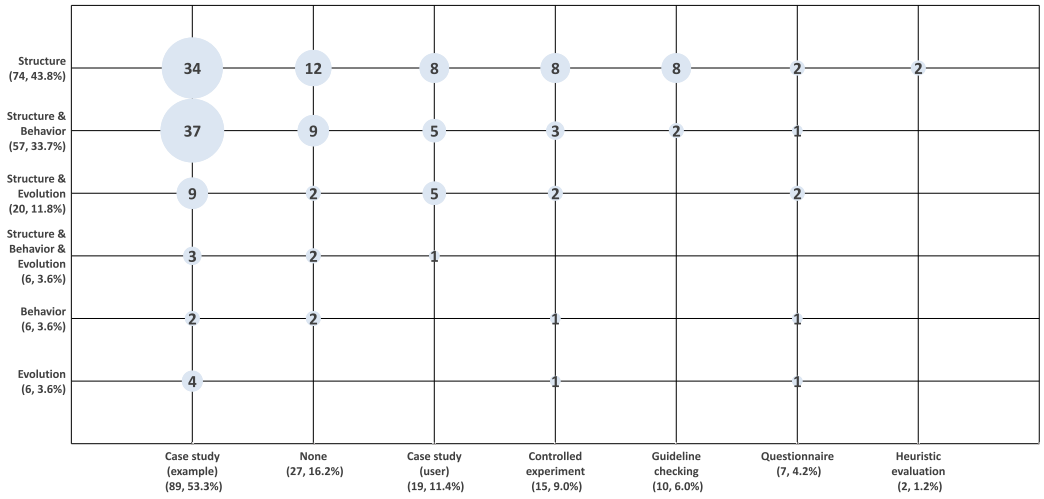


Figure 7: Evaluation methods vs. aspect for 3D software visualizations.

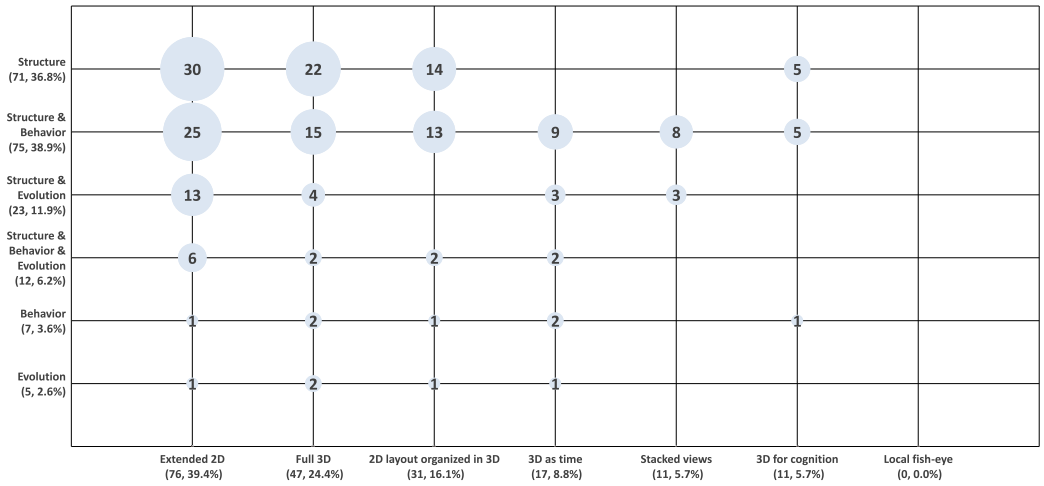


Figure 8: Application of the third dimension vs. aspect for 3D software visualizations.

Most papers extended 2D visualizations (76, 39.4%), followed by full 3D (47, 24.4%), 2D layout organized in 3D (31, 8.8%), 3D as time (17, 8.8%), and stacked views (11, 5.7%). Another 11 papers (5.7%) apply 3D for cognition only, while local fish-eye is not applied at all. The latter two will not be considered for the remaining analysis.

3D is applied for structure alone using extended 2D (30), full 3D (22), and 2D layout organized in 3D (14). Further, extended 2D and full 3D are used for all aspects and all combinations of aspects. 2D layout organized in 3D is used for all aspects except structure and evolution. In contrast, 3D as time is mostly used for structure and behavior (9), while only few paper use it for structure and evolution (3), structure, behavior, and evolution (2), behavior alone (2), and evolution alone (1). Finally, stacked views are only used for structure and behavior (8) and structure and evolution (3). Neither 3D as time nor stacked views are used for structure alone.

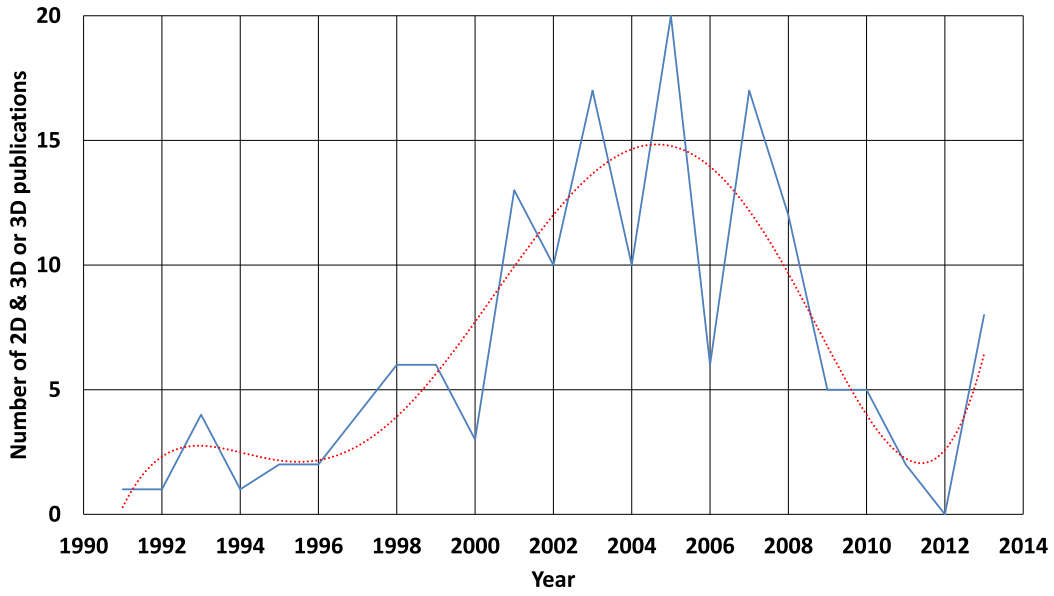


Figure 9: Number of 2D & 3D or 3D publications over time.

5 FINDINGS

Most papers dealing with 3D software visualization were published on the major software visualization conferences and workshops VisSoft (workshop until 2011), SoftVis (conference until 2010), and VisSoft conference (since 2013). A substantial amount of papers was also published at IWPC/ICPC and InfoVis. However, more than one third of the papers was published on 45 different venues.

An important functional requirement of a software visualization tool are multiple views (Kienle and Müller, 2007). These multiple views provide a holistic view of a software system combining structure, behavior, and/or evolution and thus facilitate program comprehension. The majority of 3D visualizations focus on structure, either alone (67, 43.2%) or in combination with behavior or evolution (72, 46.4%). Structure plays an important role in software visualization, as one main objective is to give the formerly intangible and invisible phenomenon software a meaningful shape (Gračanin et al., 2005). For the structural entities, such as namespaces/packages, classes, methods, as well as attributes, and their relations suitable representations are developed. The combination of these representations form the basic shape of a visualization that is usually enriched with behavioral or evolutionary information. However, the combination of all three aspects is rare in the analyzed 3D software visualizations (6, 3.9%). One reason for this might be the complexity such an approach requires. It is necessary to combine structural information with a large amount of data from execution traces and from version control systems. This might be interpreted as a serious deficit of prototype implementations and as a research gap as well.

The temporal analysis of the sample reveals that researchers started in 1991 to scrutinize the visualization of structure of software in 3D. One year later, behavioral aspects, and six years later evolutionary aspects, were also examined. Before 2001, there was no 3D software visualization covering all three aspects or behavior or evolution alone. In 2001, the field of software visualization started to establish with first tracks on software engineering conferences and the Dagstuhl seminar. Since 2002, first conferences exclusively on software visualization have been launched. These events have influenced the further evolution of this area. For example, the fluctuations of the number of 3D publications depend on the dates of the main conferences. In 2005, there was the highest number of publications (20) probably because VisSoft and SoftVis took place at the same time. In 2012, there was no publication and obviously none of these two events took place. Additionally, it was found that there was a trend to develop more 3D visualizations between 2001 and 2008 (67.7% of the papers found) with its peak around 2005 (Figure 9). The trend since

then has to be analyzed taking into account that the 3D survey of Teyseyre and Campo (2009) appeared in 2009. Thus, only the main conferences or workshops contribute to the amount of 3D software systems while other venues are not represented. Further, 2012 no event dedicated to software visualization took place. Further analysis will show, if there is a trend to continue developing 3D software visualizations.

The applied evaluation methods are distributed as follows: anecdotal ($\approx$ 53%), empirical ($\approx$ 24%), and analytical ($\approx$ 7%). Further, a large number of visualizations does not have any evaluation at all ($\approx$ 16%). This is not surprising as no evaluation at all means least effort, while anecdotal evidence can be provided with some effort. Empirical studies, on the other hand, imply a large effort for planning, execution, and analysis. At the same time, the target group—experienced software developers—are not readily available for experiments. Finally, most visualizations are already built taking guidelines into account. Therefore, guideline checking will rarely provide any benefits. In summary, the formerly stated need for more empirical evaluations of 3D software visualizations by Teyseyre and Campo (2009) still exists.

The most frequently used category for the application of the third dimension is extending a 2D visualization (76, 39.4%). The resulting additional dimension is used for example to represent software metrics, such as LOC (Boccuzzo and Gall, 2007; Alam and Dugerdil, 2007; Wettel and Lanza, 2007; Kuhn et al., 2010), complexity (Sharif and Jetty, 2013; Balogh and Beszedes, 2013), or the number of modifications (Steinbrückner and Lewerentz, 2010), for relations (Balzer et al., 2004; Caserta et al., 2011), as well as for instances (Greevy et al., 2005; Waller et al., 2013). In some cases, the use of this dimension is configurable by the user (Marcus et al., 2003; Löwe and Panas, 2005).

In the next two categories—full 3D (47, 24.4%) and 2D layout organized in 3D (31, 16.1%)—the advantage of 3D lies in the additional space that is available to represent solid 3D shapes or to optimize the layout, e.g., to avoid edge-crossings in graphs. The categories 3D as time (17, 8.8%) and stacked views (11, 5.7%) are exclusively used in visualizations containing behavioral and/or evolutionary information. Hence, these two categories are suitable for representing dynamics.

Finally, 3D is used for cognition (11, 5.7%). Irani and Ware (2003) compared 2D UML diagrams and 3D geon diagrams in several experiments. They found out that substructures can be identified more accurately with shaded components than with 2D outline equivalents and that they are remembered more reliably. Here, the third dimension does not convey additional information but it facilitates the perception of the human visual system.

To sum it up, it could be helpful to start with a basic 2D shape visualizing the structure of a software system. Further, this basic shape is extended with behavioral and evolutionary aspects using one or a combination of the identified applications of the third dimension. That is, a useful software visualization is not necessarily limited to 3D. Rather, the optimal interplay between 2D and 3D may be the clue to the successful integration of all three aspects.

6 THREATS TO VALIDITY

6.1 Reliability

We have described our method in detail and mentioned all sources in order to make this study repeatable.

6.2 Objectivity

The researcher bias mainly influences the selection and the classification of papers.

6.2.1 Selection of Papers

We increased the representative quality of the study by triangulating three different search methods. We started with manual browsing of the proceedings of the main software visualization events, continued with a keyword search of important information visualization conferences, and finished with a backward search using state-of-the-art surveys in the field of software visualization.

6.2.2 Classification of Papers

Each paper whose classification was not clear, was marked as ‘needs review’ and thoroughly discussed. Overall, there were three iterations in the ‘screen & classify’ step with altogether 60 discussable papers.

6.3 Internal and External Validity

We addressed the internal validity of our study by starting with a top-down approach to built the classification scheme. Thus, we used an established base for the categories. We have tried to increase the external validity by increasing the representative level of the sample as described in Section 6.2.1.

7 CONCLUSION

We performed a systematic literature analysis using a hybrid approach that combined a systematic mapping study followed by a systematic literature review. The research questions addressed where papers about 3D software visualization were published, which aspects were visualized, how the topic evolved over the last 22 years, how the usefulness of the 3D software visualizations was evaluated, and how the third dimension was used.

The results show that the aspect 'structure', the evaluation method 'case study (example)', and the application of the third dimension 'extended 2D' are dominant. The combination of 'structure' with 'behavior' or 'evolution' was also found relatively often.

Although, the combination of all three aspects in one software visualization tool providing a holistic view is complex and challenging to implement, we see therein a research gap for the future.

The need for more empirical evaluations of 3D software visualizations stated earlier still exists and should be addressed in future work.

Finally, the third dimension is mainly used to represent software metrics. Other successful applications are to use the additional space for solid 3D shapes and for an optimized layout, to represent time, and to amplify cognition. Probably, the optimal interplay between 2D and 3D views plays an important role in the future.

REFERENCES

Alam, S. and Dugerdil, P. (2007). EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In *14th Work. Conf. Reverse Eng.*, pages 269–270.

Alspaugh, T. A., Tomlinson, B., and Baumer, E. (2006). Using social agents to visualize software scenarios. In *Proc. 2006 ACM Symp. Softw. Vis.*, pages 87–94, New York, New York, USA. ACM Press.

Andrews, K. (2008). Evaluation comes in many guises. In *Proc. 2008 AVI Work. BEyond time errors Nov. Eval. methods Inf. Vis.*, pages 8–10.

Balogh, G. and Beszedes, A. (2013). CodeMetropolis - a Minecraft based collaboration tool for developers. In *1st IEEE Work. Conf. Softw. Vis.*, pages 1–4.

Balzer, M., Noack, A., Deussen, O., and Lewerentz, C. (2004). Software landscapes: Visualizing the structure of large software systems. In *Proc. Sixth Jt. Eurographics - IEEE TCVG Conf. Vis.*, pages 261–266. Eurographics Association.

Boccuzzo, S. and Gall, H. (2007). CocoViz: Towards Cognitive Software Visualizations. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 72–79. IEEE.

Bohner, S. A., Gracanin, D., Henry, T., and Matkovic, K. (2007). Evolutional Insights from UML and Source Code Versions using Information Visualization and Visual Analysis. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 145–148.

Brocke, J. V., Simons, A., and Niehaves, B. (2009). Reconstructing the giant: On the importance of rigour in documenting the literature search process. In *17th Eur. Conf. Inf. Syst.*, pages 1–13.

Caserta, P. and Zendra, O. (2011). Visualization of the Static Aspects of Software: A Survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933.

Caserta, P., Zendra, O., and Bodénes, D. (2011). 3D Hierarchical Edge bundles to visualize relations in a software city metaphor. In *6th Int. Work. Vis. Softw. Underst. Anal.*

Churcher, N. and Tech, V. (2003). Visualising Class Cohesion with Virtual Worlds. In *Proc. Asia-Pacific Symp. Informattion Vis.*

Cooper, H. M. (1988). Organizing knowledge syntheses: A taxonomy of literature reviews. *Knowl. Soc.*, 1(1):104–126.

Denford, M., O'Neill, T., and Leaney, J. (2002). Architecture-based Visualisation of Computer Based Systems. *9th Annu. IEEE Int. Conf. Work. Eng. Comput. Syst.*, pages 139–146.

Diehl, S. (2007). *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer.

Gil, J. and Kent, S. (1998). Three dimensional software modelling. In *20th IEEE Int. Conf. Softw. Eng.*, pages 105–114.

Gračanin, D., Matković, K., and Eltoweissy, M. (2005). Software Visualization. *Innov. Syst. Softw. Eng.*, 1(2):221–230.

Greevy, O., Lanza, M., and Wysseier, C. (2005). Visualizing Feature Interaction in 3-D. In *3rd Int. Work. Vis. Softw. Underst. Anal.*, pages 114–119. IEEE.

Hundhausen, C. D. (1996). A meta-study of software visualization effectiveness.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *J. Vis. Lang. Comput.*, 13(3):259–290.

Irani, P. and Ware, C. (2003). Diagramming information structures using 3D perceptual primitives. *ACM Trans. Comput. Interact.*, 10(1):1–19.

Jackson, S., Devanbu, P., and Ma, K.-l. (2002). Interactive Poster: Addressing Scale and Context in Source Code Visualization. In *InfoVis*.

Kienle, H. M. and Müller, H. A. (2007). Requirements of Software Visualization Tools: A Literature Survey. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 2–9. IEEE.

Koike, H. and Chu, H.-C. (1998). How does 3-D visualization work in software engineering?: empirical study of a 3-D version/module visualization system. In *Proc. 20th Int. Conf. Softw. Eng.*, pages 516–519. IEEE Computer Society.

Kuhn, A., Erni, D., and Nierstrasz, O. (2010). Embedding spatial software visualization in the IDE: an exploratory study. In *Proc. 5th Int. Symp. Softw. Vis.*, pages 113–122, New York, USA. ACM Press.

Lanza, M., D'Ambros, M., Bacchelli, A., Hattori, L., and Rigotti, F. (2013). Manhattan: Supporting real-time visual team activity awareness. In *21st Int. Conf. Progr. Compr.*, pages 207–210.

Löwe, W. and Panas, T. (2005). Rapid construction of software comprehension tools. *Int. J. Softw. Eng. Knowl. Eng.*, 15(6):905–1023.

Mackinlay, J., Robertson, G., and Card, S. (1991). The perspective wall: Detail and context smoothly integrated. In *ACM Conf. Hum. Factors Comput. Syst.*, pages 173–179.

Maletic, J., Marcus, A., and Collard, M. (2002). A task oriented view of software visualization. In *1st Int. Work. Vis. Softw. Underst. Anal.*, pages 32–40. IEEE Comput. Soc.

Marcus, A., Feng, L., and Maletic, J. (2003). Comprehension of software analysis data using 3D visualization. In *11th Int. Work. Progr. Compr.*, page 105. IEEE Computer Society.

Munzner, T. (1997). H3: laying out large directed graphs in 3D hyperbolic space. In *Vis. Conf. Inf. Vis. Symp. Parallel Render. Symp.*, pages 2–10. IEEE Comput. Soc.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Proc. 12th Int. Conf. Eval. Assess. Softw. Eng.*, pages 68–77. British Computer Society.

Reiss, S. P. (1995). An Engine for the 3D Visualization of Program Information. *J. Vis. Lang. Comput.*, 6(3):299–323.

Rilling, J. and Mudur, S. (2005). 3D visualization techniques to support slicing-based program comprehension. *Comput. Graph.*, 29(3):311–329.

Ripley, R. M., Sarma, A., and van der Hoek, A. (2007). A Visualization for Software Project Awareness and Evolution. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 137–144. IEEE.

Robertson, G., Mackinlay, J., and Card, S. (1991). Cone trees: animated 3D visualizations of hierarchical information. In *ACM SIGCHI Conf. Hum. Factors Comput. Syst.*, pages 189–194.

Sarkar, M., Snibbe, S. S., Tversky, O. J., and Reiss, S. P. (1993). Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *6th Annu. ACM Symp. User Interface Softw. Technol.*, UIST '93, pages 81–91, New York, NY, USA. ACM.

Schots, M., Vasconcelos, R., and Werner, C. (2014). A Quasi-Systematic Review on Software Visualization Approaches for Software Reuse. Technical report, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.

Schots, M. and Werner, C. (2014). Using a Task-Oriented Framework for the Characterization of Visualization Approaches. In *2nd IEEE Work. Conf. Softw. Vis.*

Seriai, A., Benomar, O., Cerat, B., and Sahraoui, H. (2014). Validation of Software Visualization Tools : A Systematic Mapping Study. In *2nd IEEE Work. Conf. Softw. Vis.*

Sharif, B. and Jetty, G. (2013). An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension. In *1st IEEE Work. Conf. Softw. Vis.*

Sjøberg, D. I. K., Dybå, T., and Jørgensen, M. (2007). The Future of Empirical Methods in Software Engineering Research. In *Futur. Softw. Eng.*, pages 358–378. IEEE.

Stasko, J. and Wehrli, J. (1993). Three-dimensional computation visualization. *Proc. 1993 IEEE Symp. Vis. Lang.*, pages 100–107.

Steinbrückner, F. and Lewerentz, C. (2010). Representing development history in software cities. In *Proc. 5th Int. Symp. Softw. Vis.*, pages 193–202, New York, USA. ACM Press.

Teyseyre, A. R. and Campo, M. R. (2009). An overview of 3D software visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(1):87–105.

von Pilgrim, J. and Duske, K. (2008). Gef3D: a framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *Proc. 4th ACM Symp. Softw. Vis.*, pages 95–104, New York, New York, USA. ACM Press.

Waller, J., Wulf, C., Fittkau, F., Döhring, P., and Hasselbring, W. (2013). SynchroVis : 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *1st IEEE Work. Conf. Softw. Vis.*, pages 7–10.

Wettel, R. and Lanza, M. (2007). Visualizing Software Systems as Cities. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 92–99. IEEE.

Wettel, R. and Lanza, M. (2008). Visually localizing design problems with disharmony maps. In *Proc. 4th ACM Symp. Softw. Vis.*, pages 155–164, New York, New York, USA. ACM Press.

3.2 Summary

The literature study presents the results of analyzing the applications of 3D in software visualization with the objectives to outline the state-of-the-art, to reveal trends, and to identify research gaps. It combines a systematic mapping study [Petersen et al. 2008] and a literature review [vom Brocke et al. 2009].
The study refers to ***SQ1:*** *What is the state-of-the-art in 3D software visualization?* (see Section 1.2) and details it with the following research questions:

- *Venue:* Which workshops/conferences/journals include papers on 3D software visualization?
- *Aspect:* Which aspects of software (structure, behavior, evolution) are visualized with 3D?
- *Evolution:* How did 3D software visualization evolve over the last 22 years and what are current trends?
- *Evaluation:* How is the usefulness of the proposed 3D software visualizations evaluated?
- *Application:* How is the third dimension used?

The results of the study concerning venue indicate that most papers dealing with 3D software visualization were published on the major software visualization conferences and workshops including VisSoft (workshop until 2011), SoftVis (conference until 2010), and VisSoft conference (since 2013). A substantial amount of papers was also published at IWPC/ICPC and InfoVis. However, more than one third of the papers was published on 45 different venues.
The analysis of the 3D software visualizations shows that the aspect *structure* is dominant. The combination of *structure* with *behavior* or *evolution* was also found relatively often. However, the combination of all three aspects in one software visualization tool providing a holistic view was rarely found. Kienle and Müller [2007] argue that a visualization tool should provide multiple views. Thus, this lack is considered to be a research gap.
The temporal analysis of the sample reveals that two thirds of the 3D software visualization papers were published between 2001 and 2008. The 3D hype with its peak around 2005 is receding but it begins to slope again.
The applied evaluation methods are distributed as follows: anecdotal ($\approx 53\%$), empirical ($\approx 24\%$), and analytical ($\approx 7\%$). Further, a large number of visualizations does not have any evaluation at all ($\approx 16\%$). Consequently, the need for more empirical evaluations of 3D software visualizations stated earlier by Teyseyre and Campo [2009] still exists and should be addressed in future work.
The categories of the application of the third dimension are based on Reiss [1995] and extended by a further category: *3D for cognition*. Thus, the analysis of the application covers six categories[3]: *extended 2D*, *full 3D*, *2D layout organized in 3D*, *3D as time*, *stacked views*, and

[3] The original category *local fish-eye* was not found in the sample and for this reason omitted.

3D for cognition. The most frequently used category for the application of the third dimension is *extending a 2D* visualization. The resulting additional dimension is used for example to represent software metrics, such as LOC [Boccuzzo and Gall 2007; Alam and Dugerdil 2007; Wettel and Lanza 2007; Kuhn et al. 2010] (see Figure 2.1 (a)), complexity [Sharif and Jetty 2013; Balogh and Beszedes 2013], or the number of modifications [Steinbrückner and Lewerentz 2010] (see Figure 2.1 (f)), for relations [Balzer et al. 2004; Caserta et al. 2011], as well as for instances [Greevy et al. 2005; Waller et al. 2013] (see Figure 2.1 (c)). In some cases, the use of this dimension is configurable by the user [Marcus et al. 2003; Löwe and Panas 2005]. In the categories *full 3D* and *2D layout organized in 3D* the advantage of 3D lies in the additional space that is available to represent solid 3D shapes or to optimize the layout, e.g., to avoid edge-crossings in graphs. An example for these categories provide Eicker et al. [2007] (see Figure 2.1 (b)). The categories *3D as time* and *stacked views* are exclusively used in visualizations containing behavioral or evolutionary information. Examples for the category *3D as time* for software evolution provide Ripley et al. [2007] (see Figure 2.1 (e)) and the category *stacked views* for software behavior use von Pilgrim and Duske [2008] (see Figure 2.1 (d)). Hence, these two categories are suitable for representing dynamics. Finally, there is the category *3D for cognition*. Irani and Ware [2003] compared 2D UML diagrams and 3D geon diagrams in several experiments. They found out that substructures can be identified more accurately with shaded components than with 2D outline equivalents and that they are remembered more reliably. Here, the third dimension does not convey additional information but it facilitates the perception of the human visual system.
To sum it up, it could be helpful to start with a basic 2D shape visualizing the structure of a software system. Further, this basic shape is extended with behavioral and evolutionary aspects using one or a combination of the identified application categories of the third dimension. That is, a useful software visualization is not necessarily limited to 3D. Rather, the optimal interplay between 2D and 3D may be the clue to the successful integration of all three aspects.

4 Generator for 2D, 2.5D, and 3D Software Visualizations

Müller, Richard, Pascal Kovacs, Jan Schilbach, and Ulrich Eisenecker. 2011. "Generative Software Visualization: Automatic Generation of User-Specific Visualizations." In *Proceedings of the International Workshop on Digital Engineering*, Magdeburg, Germany.[4]

4.1 Generative Software Visualization: Automatic Generation of User-Specific Visualizations

[4] In order to align the wording of the paper with the phd thesis, the abbreviations 2d, 2,5d, and 3d are changed to 2D, 2,5D, and 3D in the attached version of the paper.

Generative Software Visualization: Automatic Generation of User-Specific Visualizations

Richard Müller
rmueller@wifa.uni-leipzig.de

Pascal Kovacs
kovacs@wifa.uni-leipzig.de

Jan Schilbach
schilbach@wifa.uni-leipzig.de

Ulrich W. Eisenecker
eisenecker@wifa.uni-leipzig.de

Information Systems Institute
University of Leipzig
Leipzig, Germany

ABSTRACT

Software visualization provides tools and methods to create role- and task-specific views on software systems to enhance the development and maintenance process. However, the effort to produce customized and optimized visualizations is still high. Hence, we present our approach of combining the generative and the model driven paradigm and applying it to the field of software visualization. Based on this approach we want to implement a generator that allows to automatically generate software visualizations in 2D, 2.5D, 3D, or for virtual reality environments according to user-specific requirements.

Keywords

software visualization, model driven visualization, software visualization families, automation

1. INTRODUCTION

The preconditions for software visualization in 3D and virtual reality (VR) have improved dramatically, because of increased computing power available at low price and new presentation and interaction techniques. Our research tries to explore the resulting potential for software engineering, especially with respect to software development as well as maintenance.

Software visualization has the potential to considerably enhance understanding of software through providing structural, behavioral, evolutionary, or combined views [7]. This understanding is necessary for nearly all stakeholders involved in software development and maintenance, such as developers, project managers, and customers. All of these stakeholders have different tasks in different parts the of software lifecycle and therefore need different information about the software system they are involved in. Software visualizations have to support these users and their tasks, otherwise they are not useful and therefore will not be used. This task-oriented view was first proposed by Maletic et al. [12].

Reiss [16] identified some important issues of software visualization. Of these we address the lack of simplicity to use a visualization technique and the lack of adoption to real user problems. One reason for the lack of simplicity is, that visualization users need to supply exactly the data the visualization tool demands, because many tools require a special input format which the user has to provide. With our approach we are able to handle multiple software artifacts in multiple input formats, as long as they are in a well-structured format. The second issue, the lack of adoption to real user problems, comes with the mostly general scope of actual visualization tools. Reiss [16] takes the resource usage in the time around a specific event as an example for a real world use case, which is not covered by the available visualization tools. We believe that the adoption of principles, methodology, and techniques of software system families is the basis for developing a generator that adresses these problems. This generator takes one or more software artifacts and an easy-to-create configuration of the desired visualization as input. Furthermore, it provides ready-to-use visualizations optimized to the users requirements without the necessity of additional user intervention. These visualizations optimally support the different user needs and therefore the specific tasks in the process of the software development and maintenance. Moreover, the visualizations produced by the generator support any form of visualization technique be it two-, two-and-a-half- or three-dimensional (2D, 2.5D or 3D), printed on paper, displayed on a monitor, or presented in VR.

In this paper, we will describe the theoretical concepts of our approach and the design of the generator. In Section 2, we summarize theories and publications our work is based on. In Section 3, we explain the generative visualization process. In this context, we will first introduce the basics of Generative Programming, especially the generative domain model, and derive a generative software visualization domain model. Afterwards, we outline a technology projection to show how this model can be instantiated. To explain the characteristics of this approach, an example scenario demonstrates the process of generative software visualization in Section 4. The conclusion gives a brief evaluation of the work described in this article and provides an outlook to future research.

2. RELATED WORK

To respect the task-specific needs of different users, many tools for software visualization, e. g. Mondrian [14], CodeCity [17], or sv3d [13], allow to configure some aspects of the

visualization. However, these tools are limited with respect to their configuration options, e. g. the number of metaphors and layouts they offer. Furthermore, the configuration requires a substantial amount of additional manual work, or the tools are restricted to a single type of software artifact. Vizz3D [15] and Model Driven Visualization (MDV) [4] try to overcome these deficits with a more general approach.

Bull [4] describes MDV as an architecture for adapting the concepts of Model Driven Engineering (MDE). The software models used as input have to correspond to a platform independent metamodel, e. g. Dagstuhl Middle Model (DMM) [11]. Such a model can be retrieved by parsing sourcecode. To generate the visualizations, platform independent models called views are used, e. g. tree views or nested views. The definition of the necessary transformations between the input models and the view models have to be programmed by the user in a model transformation language, e. g. Atlas or Xtend[1]. After the transformation, a platform specific visualization will be automatically generated for a certain tool, e. g. Zest[2].

As a weakness of this approach, we identified the necessary creation of platform independent view models from scratch. On the one hand, this creates a high level of freedom. On the other hand, many possible benefits are prevented, such as using a common layout algorithm for different view models. Another drawback is the use of complex multi-purpose model transformation languages which are not easy to understand for non-experts. We show that using a formalized domain specific language (DSL) to describe the mapping from source to view elements will be easier to use while still preserving the automatic generation of a visualization.

Panas et al. [15] describe Vizz3D as a framework for configuring a visualization by using models and transformations. Beginning with a formalized model of software corresponding to a metamodel defined by Vizz3D, the user first configures the mapping to an abstract view. This view has a graph structure with nodes and edges including properties such as color or shape. In a second step, the user configures the mapping of the view to a concrete scene rendered by a tool, including the configuration of a metaphor and an optional layout.

A limitation of this approach is the required transformation of the users data into the Vizz3D source format, which causes additional effort. The user has to define two mappings to configure a visual representation instead of only one mapping. Finally this results in a tight coupling of platform independent metaphors to platform specific visualization tools.

3. GENERATIVE SOFTWARE VISUALIZATION

3.1 Generative Paradigm

Generative Programming aims at the automized production of software systems based on software system families:

> *"Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."* [6]

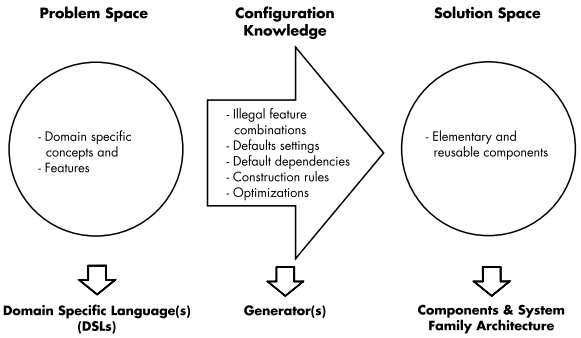


Figure 1: Generative Domain Model (GDM) [6]

Such a family covers a set of systems being similar enough from an architectural point of view to be built from a common set of assets. The requirements of the resulting system are described with a DSL. In this context, a domain is an area of knowledge comprising expert knowledge from stakeholders and technical knowledge of how to build software systems. A DSL is a specialized and problem-oriented language for domain experts to specify concrete members of a system family. It abstracts from technical knowledge and implementation details. This specification is processed by a generator, which automatically assembles the system by combining elementary and reusable components according to configuration knowledge and a flexible system family architecture as well. Components are building blocks for assembling different systems of a family.

The basic terms of the generative paradigm and their relationships are summarized by the generative domain model (GDM, see Fig. 1). It comprises the problem space, the solution space, as well as configuration knowledge for mapping the problem space to the solution space. The problem space covers domain specific concepts as well as their features used by domain experts to specify their requirements. The requirements are expressed in terms of one or more DSLs. The solution space includes elementary and reusable implementation components which can be assembled as defined through the system family architecture. The configuration knowledge encapsulates illegal feature combinations, default settings, construction rules, and optimizations as well as related information.

The parts of a GDM and two or more GDMs can be connected in different ways [5]. One possibility is that the solution space of one GDM is the problem space of another GDM. This is called a chaining of mappings. Furthermore, specifications in a DSL can be processed by different generators which map them to different solution spaces. In this case, there are several alternative solution spaces instead of only one.

3.2 Generative Software Visualization Domain Model

Comparing the generative paradigm with the field of software visualization, especially its visualization process, yields

[1]http://www.eclipse.org/modeling/emf/
[2]http://www.eclipse.org/gef/zest/

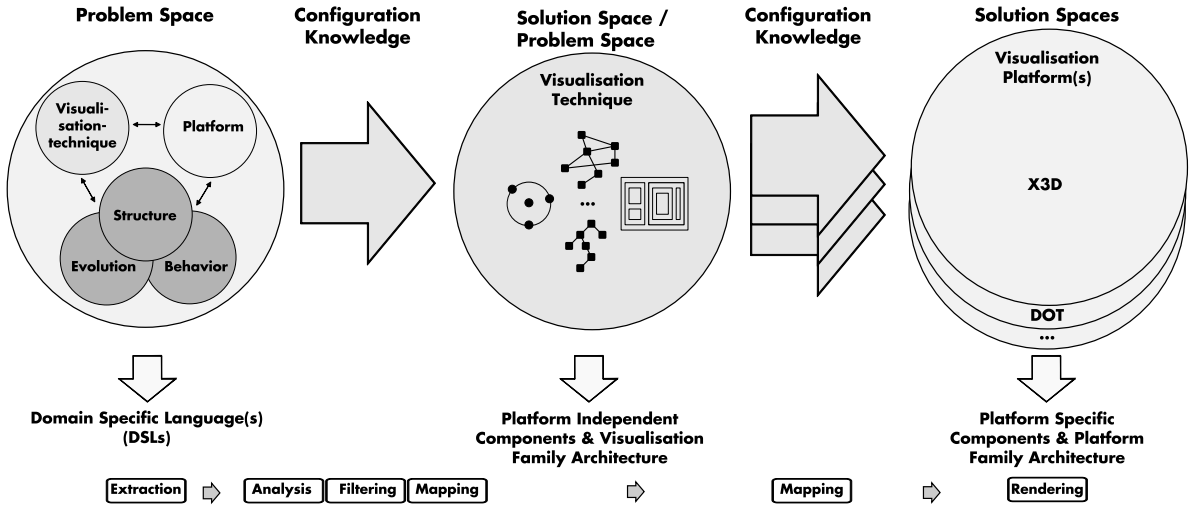


Figure 2: Generative Software Visualization Domain Model (GSVDM)

many remarkable similarities. A visualization should be automatically generated according to user-specific requirements by mapping information of software artifacts to a visual representation. For this reason, we adapt the definition of GP as follows:

> *"The visualization process should be arranged such that, given a particular requirements specification, a highly customized and optimized visualization can be automatically generated on demand from elementary, reusable implementation components belonging to a visualization family by means of configuration knowledge."*

The difference to the original definition is that the result of the generation process is not a software system but an optimized and ready-to-use visualization representing the structure, the behavior, and/or the evolution of a software system. Instead of assembling each visualization manually, it is created automatically from implementation components on the basis of a visualization family according to a specification in a DSL provided by the user.

This concept can be described in terms of the generative domain model. The chaining of mappings and the alternative solution spaces are used to realize the high variability of platforms for different visualization techniques. Besides that, the separation of the platform independent and the platform specific solution spaces makes it possible to reuse the implementation components. The resulting concept is called generative software visualization domain model (GSVDM, see Fig. 2).

The problem space offers means to specify concrete members of a visualization and a platform family for representing information from software artifacts. Using a DSL, the user can specify on which platform which information of a software artifact has to be visualized with which visualization technique. The DSL corresponds to the requirements specification in the above mentioned definition and abstracts from concrete implementations.

The solution space includes implementation components that can be assembled according to a family architecture. Due to the chaining of mappings, there are at least two solution spaces. The platform independent space contains abstract visualization techniques, such as trees, graphs, tables, and abstract or real world metaphors. The platform specific spaces provide concrete platforms for these techniques, such as Graphviz [10], Tulip [1], Gephi [2], or X3D [3].

The configuration knowledge, which is implemented as a generator, defines the mapping of problem space to solution space. In this case, the generation process is also the visualization process. This means that the process corresponds to a fully automated visualization process comprising all necessary parts of a visualization pipeline [8]. Thus, the knowledge about illegal feature combinations, default settings, default dependencies, construction rules, and optimizations is augmented with knowledge about extraction, analysis, filtering, mapping, and rendering from the DSL.

3.3 Model Driven Technology Projection

In order to implement this theoretical concept, it is necessary to identify concrete techniques for the elements of the software visualization domain model. Consequently, all spaces are described with structured models, and the configuration knowledge provides mappings between these spaces using model-to-model transformations. The DSL can be implemented as text-only or as a dialogue-based wizard controlling the different steps of the visualization pipeline. By using the XText-Framework[3] for implementing the DSL, we will be able to utilize existing functionality, to provide code completion, syntax highlighting and other useful features.

The starting point of the automatic visualization process are structured software artifacts containing information about structure, behavior, or evolution of software systems. Some of those structures – also known as metamodels – are Ecore, UML, XML or the DMM.

The common architecture for the visualization family is provided by a visualization technique meta-metamodel. It consists of a graph with nodes and edges where each element can have additional properties. The basic assumption be-

[3] http://www.eclipse.org/Xtext/

hind this is that all visualization techniques can be reduced to this abstract structure, so it is sufficient to have only one model that can be flexibly instantiated. Another advantage of using a graph structure is the ability to employ existing 2D and 3D layout algorithms rather than implementing them. The different platform independent visualization techniques are the implementation components.

It is obvious that the common architecture for the visualization platform depends on the used platform. For this reason, there is one model for each platform. Imagine the visualization should be an X3D-scene. Then, the X3D-file is the model, the X3D-schema definition is the metamodel, and the XML schema definition is the meta-metamodel. Further examples for visualization platforms are DOT[4] from Graphviz, TLP[5] from Tulip, or GEXF[6] from Gephi.

The configuration knowledge maps the elements of software artifacts to the elements of visualization techniques and finally to elements of a visualization platform corresponding to the DSL. The steps of the visualization pipeline are implemented by means of the Eclipse Modeling Framework. In this way, formal models can be analyzed and checked with predefined validation rules. For the mapping, i. e. model-to-model transformations, the transformation rules are defined on the meta-level and they are applied for each model conforming to the corresponding metamodel. In order to handle more than one source model the so called model-weaving is used. The rendering is done by the concrete visualization platforms.

4. EXAMPLE SCENARIO

In order to illustrate our approach we want to use a simple fictional scenario. Imagine a project manager who is preparing a meeting. In order to make the development team pay attention to current problems, information of the software system's structure enhanced with metrics is required. The system under development is a banking system implemented in Java and the relevant metrics are McCabe Complexity and LOC. To make it more understandable for all stakeholders, the manager waives source code and complex tables. Instead, the visualization should be 3D and represented by a nested visualization technique, which can be explored interactively in the company's virtual reality environment.

To do this in a generative way, the following steps have to be carried out. As a precondition, the necessary information has to be available in formal models, e.g. an Ecore model for the structure and an XML file for the metrics. Initially, the models are loaded by the generator (*extraction*). If necessary, the user can apply predefined rules to check the models for consistent semantics (*analysis*). Then, the relevant information from the software artifacts is selected (*filtering*), and a visual representation for each piece of information is defined by the user (*mapping*). This mapping comprises two stages: In the first stage, the visualization technique is selected, and in the second stage, the visualization platform is chosen. During these stages, the user sets the mapping rules for packages, classes, methods, attributes, and references to clusters, nodes, and edges as well as the visual appearance including shape, size, and colour of the different types of clusters, nodes, and edges. Clusters are special nodes, that can contain further clusters or nodes. The mappings are either completely controlled by the user or default mappings are applied. The simplified mapping rules in the banking example are as follows:

- package ↦ cluster ↦ brown cube
- class ↦ cluster ↦ blue sphere
- method ↦ node ↦ green to red cylinder
- attribute ↦ node ↦ blue cone
- reference ↦ edge ↦ blue line
- McCabe ↦ node ↦ green(low) - red(high)
- LOC ↦ cluster/node ↦ size

At this time, only the size and positions of the elements in the 3D space are missing. Here the graph structure from the meta-metamodel comes into play. By applying established layout algorithms the missing information is computed. For the banking example a force directed layout algorithm for clustered graphs is used [9]. Now, the generator has all necessary information to produce the visualization. As a result, the X3D-model in Fig. 3 and 4 is automatically generated

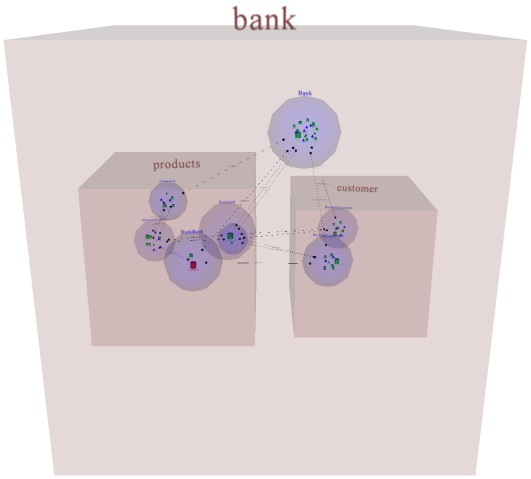


Figure 3: X3D-model of the banking example including structure and metrics (Overview)

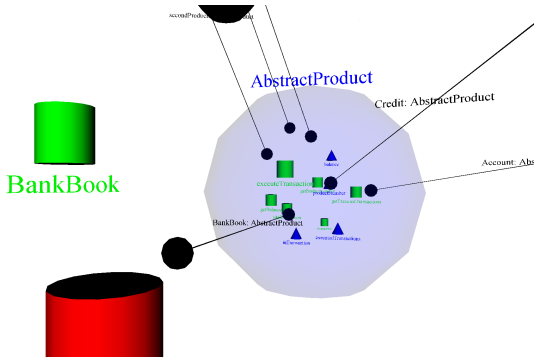


Figure 4: X3D-model of the banking example including structure and metrics (Detail)

[4]http://www.graphviz.org/content/dot-language
[5]http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format
[6]http://gexf.net/

and can be interactively explored in a suitable browser, such as an Eclipse view, a standalone X3D-Browser, or a virtual reality environment. These visualizations have been generated using a predecessor of the planned generator. In this predecessor a box- and solar-system-metaphor as well as X3D as target platform are hard-wired. By this means, it was possible to visualize the structure of a real-world example with several hundred classes.

5. CONCLUSION AND FUTURE WORK

It was explained how the generative paradigm and the model driven paradigm can be adopted to meet the requirements of generating highly customized and ready-to-use software visualizations by the user without writing any glue code by hand. Hence, this promising concept makes it possible to integrate different kinds of software artifacts with different visualization techniques and well-approved visualization tools, not being limited to a specific platform and configured by an easy-to-use DSL.

Our future work will continue the implementation of the generator architecture and infrastructure based on the Eclipse platform, the specification of the grammar of the DSL, the iterative development of the meta-metamodel for visualization techniques including a representative amount of metamodels of visualization techniques as well as the integration of some established visualization tools. Eventually, we plan to use the generator to evaluate different visualization aspects, like three-dimensionality, animation, and interaction for their suitability in different tasks and different stages of the software life cycle. With the resulting findings we want to improve the spread of task- and role-specific software visualization in industrial software development and maintenance.

6. REFERENCES

[1] D. Auber. Tulip : A huge graph visualisation framework. In P. Mutzel and M. Jünger, editors, *Graph Drawing Softwares*, Mathematics and Visualization, pages 105–126. Springer-Verlag, 2003.

[2] M. Bastian, S. Heymann, and M. Jacomy. Gephi: an open source software for exploring and manipulating networks. 2009.

[3] D. Brutzman and L. Daly. *X3D: Extensible 3D Graphics for Web Authors*. Elsevier, 2007.

[4] R. I. Bull. *Model Driven Visualization: Towards a Model Driven Engineering Approach for Information Visualization*. PhD thesis, University of Victoria, 2008.

[5] K. Czarnecki. Overview of generative software development. In *Unconventional Programming Paradigms*, number 3566 in Lecture Notes in Computer Science, pages 326–341. Springer, Berlin Heidelberg, 2005.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications: Methods, Techniques and Applications*. Addison-Wesley Longman, Amsterdam, June 2000.

[7] S. Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[8] S. dos Santos and K. Brodlie. Gaining understanding of multivariate and multidimensional data through visualization. *Computers & Graphics*, 28(3):311–325, June 2004.

[9] T. Dwyer. Extending the WilmaScope 3D graph visualisation system: software demonstration. In *APVis '05: proceedings of the 2005 Asia-Pacific symposium on Information visualisation*, pages 39–45. Australian Computer Society, Inc., 2005.

[10] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[11] T. C. Lethbridge, E. Plödereder, S. Tichelaar, C. Riva, P. Linos, and S. Marchenko. The dagstuhl middle model (DMM). http://www.site.uottawa.ca/~tcl/dmm/DMMDescriptionV0006.pdf, 2002.

[12] J. I. Maletic, A. Marcus, and M. L. Collard. A task oriented view of software visualization. In *VISSOFT 2: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE Computer Society, 2002.

[13] A. Marcus, L. Feng, and J. I. Maletic. Comprehension of software analysis data using 3D visualization. In *Proc. 11th IWPC*. IEEE Computer Society, 2003.

[14] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: an agile information visualization framework. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 135–144, Brighton, United Kingdom, 2006. ACM. ACM ID: 1148513.

[15] T. Panas, R. Lincke, and W. Löwe. Online-configuration of software visualizations with vizz3d. In *Proc. 2nd SoftVis*, pages 173–182, New York, NY, USA, 2005. ACM.

[16] S. P. Reiss. The paradox of software visualization. In *Proc. 3rd VISSOFT*, pages 59–63, 2005.

[17] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 551–560, Waikiki, Honolulu, HI, USA, 2011. ACM. ACM ID: 1985868.

4.2 Summary

The software visualization generator combines the generative and the model-driven paradigms to produce role- and task-specific visualizations automatically according to user requirements specified in a DSL. These visualizations may represent structural, behavioral, and/or evolutionary aspects of a software system in 2D, 2.5D, or 3D.
The generator refers to ***SQ2:*** *How can 2D, 2.5D, and 3D software visualizations be generated automatically?* and is the first precondition for answering ***SQ3:*** *What role does the factor dimensionality play in solving software engineering tasks?* (see Section 1.2). It is used to generate 2D, 2.5D, and 3D software visualizations for the controlled experiment described in Chapter 6. Moreover, it is used to generate the recursive disk metaphor presented in Chapter 7.

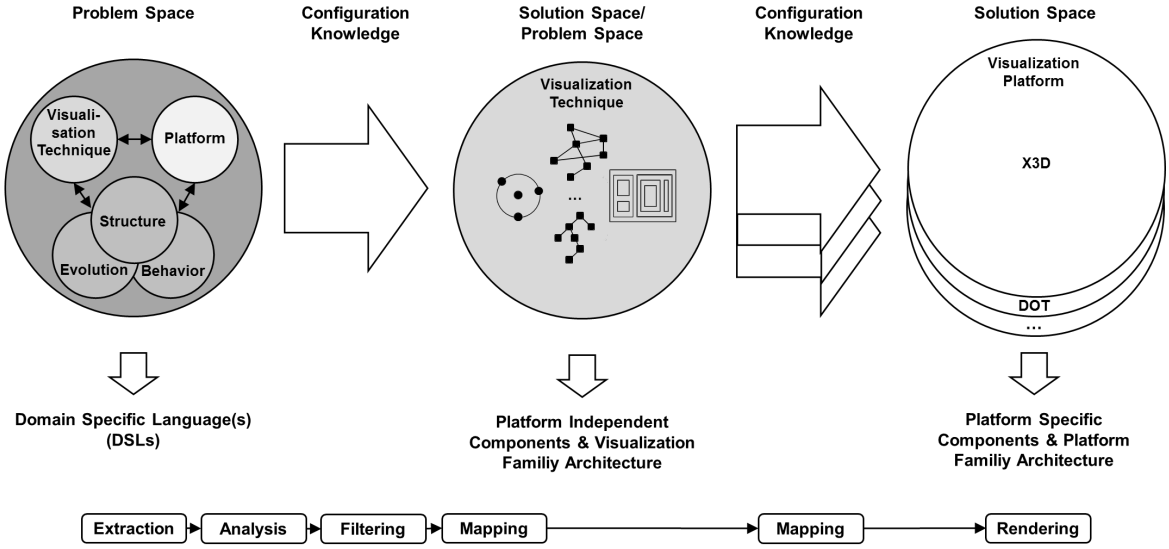


Figure 4.1: Generative software visualization domain model [Müller et al. 2011].

The main concepts of the generator are summarized by Figure 4.1 and by the following definition: *"The visualization process should be arranged such that, given a particular requirements specification, a highly customized and optimized visualization can be automatically generated on demand from elementary, reusable implementation components belonging to a visualization family by means of configuration knowledge."* [Müller et al. 2011]. The generative software visualization domain model is a chained mapping combining three spaces. First, the user specifies the aspect or combination of aspects of a software system to be visualized, the visualization technique, and the visualization platform with a DSL. Second, the DSL is handed over to the first solution space where the layout of the visualization technique is computed. This results in a platform independent model. Third, the second space turns into a problem space and the visualization is transformed to a specific visualization platform.
In order to implement this theoretical concept, it is necessary to identify concrete techniques for the elements of the generative software visualization domain model, i.e., a technology projection is applied. Here, the model-driven paradigm comes into play. Hence, all spaces

are described by formal models and the configuration knowledge provides mappings between the spaces using M2Ps and M2Ms. The metamodels for the aspects and for the visualization technique are serialized in MSE [Kuhn and Verwaest 2008]. The structure is modeled with Famix [Ducasse et al. 2011], the behavior with Dynamix [Greevy 2007], and the evolution with Hismo [Ducasse et al. 2004]. The metamodels for the aspects and for the visualization technique are defined in an Xtext grammar. The metamodel for the platform is defined by the X3D XSD [X3D Schema 2014]. As X3D scene graphs can be rendered on any platform, the software visualizations are platform independent. With AOPT they are optimized for the web and can be interactively explored in any browser supporting X3DOM. The transformation of the models, i.e., the configuration knowledge, is realized with Xtend. Among others, the visualization platform is covered by X3D and X3DOM. All transformations are configured and controlled by MWE2 workflows. The generator is implemented in Eclipse using JDT, PDE, TMF, and EMP. It has a modular and extensible architecture and consists of three parts. The Xtext metamodels are stored in plug-in projects named `org.svis.xtext.[metamodel]`. The core of the generator including the Xtend transformations is the plug-in project `org.svis.generator`. The MWE2 workflows to configure and to control the transformations are in the plug-in project `org.svis.generator.test`. The complete architecture and dependencies of the generator are visualized with a component diagram in Figure 4.2.

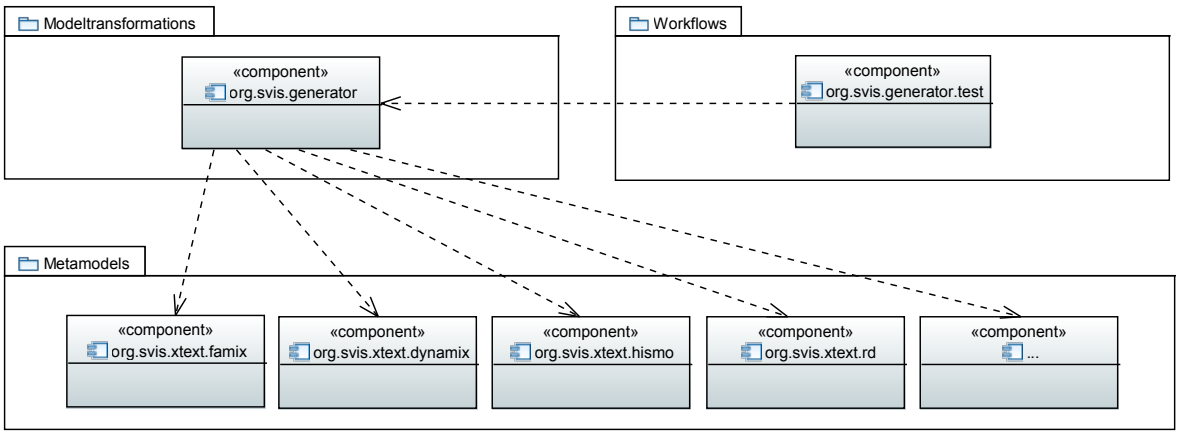


Figure 4.2: Architecture and dependencies of the generator in a component diagram.

5 Structured Approach

Müller, Richard, Pascal Kovacs, Jan Schilbach, Ulrich Eisenecker, Dirk Zeckzer, and Gerik Scheuermann. 2014. “A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization.” In *Proceedings of the 5th International Conference on Visualization Theory and Applications*, Lisbon, Portugal.[5]

5.1 A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization

[5] A spelling mistake of the word Representation in Figure 1 is corrected in the attached version of the paper.

A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization

Richard Müller[1], Pascal Kovacs[1], Jan Schilbach[1], Ulrich W. Eisenecker[1], Dirk Zeckzer[2] and Gerik Scheuermann[2]

[1]*Information Systems Institute, University of Leipzig, Leipzig, Germany*

[2]*Institute of Computer Science, University of Leipzig, Leipzig, Germany*

{*rmueller, kovacs, schilbach, eisenecker*}*@wifa.uni-leipzig.de,* {*zeckzer, scheuermann*}*@informatik.uni-leipzig.de*

Keywords: Software Visualization, Evaluation, Controlled Experiment, 3D.

Abstract: In the field of software visualization controlled experiments are an important instrument to investigate the specific reasons, why some software visualizations excel the expectations on providing insights and ease task solving while others fail doing so. Despite this, controlled experiments in software visualization are rare. A reason for this is the fact that performing such evaluations in general, and particularly performing them in a way that minimizes the threats to validity, is hard to accomplish. In this paper, we present a structured approach on how to conduct a series of controlled experiments in order to give empirical evidence for advantages and disadvantages of software visualizations in general and of 2D vs. 3D software visualizations in particular.

1 INTRODUCTION

Determining the circumstances why and when a software visualization is well suited to support a specific software engineering task remains a big challenge. Several factors have to be considered, e. g., the type of software under inspection, the representation used to depict the software artifact, navigation and interaction as well as the implementation.

A suitable approach to determine these circumstances is the controlled experiment. It is a generally accepted research and evaluation method in information visualization (Carpendale, 2008; Andrews, 2008; Munzner, 2009; Isenberg et al., 2013), in software engineering (Sjoberg et al., 2007), and in software visualization (Tichy and Padberg, 2007; Di Penta et al., 2007). But one single controlled experiment is not sufficient because there are too many factors that might influence the result. For example, (Dwyer, 2001) did not conduct a planned experiment because it *"(. . .) would be inconclusive due to the number of unconstrained variables involved."*

One important question to be addressed in software visualization is the role of dimensionality. The strengths and weaknesses of 3D visualizations have been controversially discussed over the last two decades. (Teyseyre and Campo, 2009) provide a concise overview of the ongoing scientific discourse.

From a technical point of view, major weaknesses of 3D are the intensive computation and the complex implementation. The computational effort is more and more diminishing due to the increasing computing power. To minimize the development effort there are several promising approaches (Bull et al., 2006; Müller et al., 2011). Another point is that nowadays technical issues like ghosting, calibration, and resolution are no longer an issue as Custom-off-the-Shelf solutions exist. The ongoing technical evolution of 3D, such as in cinema and on TV (Huynh-Thu et al., 2011), interaction devices like Kinect (Smisek et al., 2011) or Leap Motion (Weichert et al., 2013), and merging of web- and home-entertainment systems (Zorrilla et al., 2013) offers new opportunities for software visualization.

From a visualization point of view, the major weaknesses of 3D are occlusion and more complex navigation. Strengths are the additional dimension, often used to depict time, the integration of local into global views, the composition of multiple 2D views in a single 3D view, and the facilitation of perception of the human visual system.

Even at the latest VISSOFT conference there were five papers dealing with 3D in software visualization (Waller et al., 2013; Sharif and Jetty, 2013; Benomar and Poulin, 2013; Balogh and Beszedes, 2013; Fittkau et al., 2013).

For these reasons, we raise the questions again why and when is 3D better or worse than 2D in software visualization. Hence, a series of experiments is needed which investigates the role of dimensionality in several configurations varying a single factor systematically in each experiment while keeping the remaining ones constant or measure their influence on the result.

The contribution of this paper is the underlying structured approach for conducting such a series of controlled experiments in order to give empirical evidence for advantages and disadvantages of software visualizations, especially for 2D vs. 3D.

2 RELATED WORK

Our approach is based on the lessons learned from other experiments in software visualization (Sensalire et al., 2009), e. g., concerning experiment's duration and location as well as tool and task selection. In addition, we incorporate the hints, guidelines and frameworks for controlled experiments in information visualization and software engineering (Basili et al., 1986; Pfleeger, 1995; Kosara et al., 2003; Sjoberg et al., 2007; Carpendale, 2008; Keim et al., 2010; Wohlin et al., 2012), e. g., conduct a pilot study and training tasks, take care of and document all factors that may influence the results, and clearly describe the threats to validity.

For our series of experiments we apply Munzner's process model for design and validation of visualizations (Munzner, 2009). The four nested layers of the design process are *domain problem characterization*, *data and operation abstraction*, *encoding and interaction technique*, and *algorithm*. Each part has corresponding validation methods. (Meyer et al., 2012) extended this model with blocks and guidelines. Blocks are outcomes of the design process at each level and guidelines describe the relations between these blocks. The model as well as its extension makes visualization research transparent and comparable and supports researchers to structure their approach and to identify research gaps.

Important prior work comparing 2D and 3D visualizations with controlled experiments in information visualization as well as in software visualization has been performed. Ware et al. examined the perception and the layout of graphs displayed in different dimensions and environments (Ware et al., 1993; Ware and Franck, 1994; Ware and Franck, 1996; Ware and Mitchell, 2008). (Levy et al., 1996) examined users' preferences for 2D and 3D graphs in different scenarios. (Cockburn and McKenzie, 2001) compared a 2D and a 3D representation of a document management system. (Koike and Chu, 1998) conducted experiments to compare two version control and module management systems RCS (2D) and VRCS (3D). Irani et al. developed 3D geon diagrams and examined their benefit empirically (Irani and Ware, 2000; Irani and Ware, 2003). (Wettel et al., 2011) provided empirical evidence in favor of a 3D metaphor representing software as a city. (Sharif and Jetty, 2013) assessed the effect of SeeIT 3D on comprehension. All these experiments show that there are benefits offered by 3D over 2D in performance, error rates, or preference. (Cockburn and McKenzie, 2002) evaluated the effectiveness of spatial memory in 2D and 3D. They found that navigation in 3D spaces can be difficult. Nonetheless, there is still a lack of empirical evidence supporting the 2D versus 3D discussion especially in the field of software visualization (Teyseyre and Campo, 2009). With our series of controlled experiments we aim at extending the knowledge about

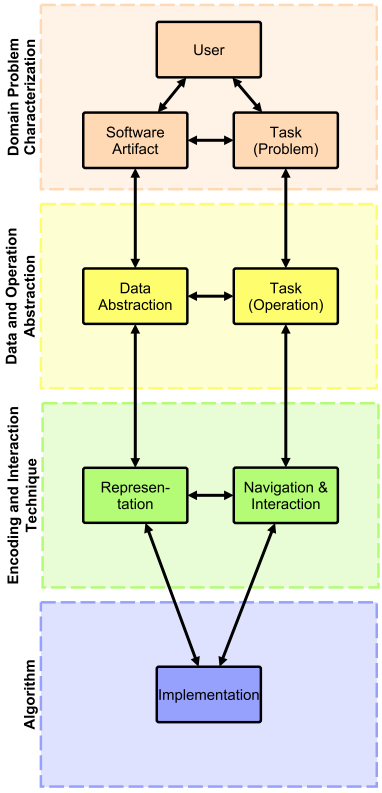


Figure 1: Domain specific adaption of Munzner's extended model for software visualization (Munzner, 2009; Meyer et al., 2012).

advantages and disadvantages of the third dimension in software visualization.

3 A NEW PERSPECTIVE ON CONTROLLED EXPERIMENTS

In our approach, we adapt Munzner's extended model for visualization design and validation to the software visualization domain (Munzner, 2009; Meyer et al., 2012).

First, we derived the influence factors from several taxonomies in the field of information visualization in general and in the field of software visualization in particular (Myers, 1990; Stasko and Patterson, 1993; Price et al., 1993; Roman and Cox, 1993; Maletic et al., 2002; Storey et al., 2005; Gallagher et al., 2008). These factors are *user*, *task*, *software artifact*, *navigation and interaction*, *representation*, and *implementation*.

Then, we assigned these factors to Munzner's extended model. Consequently, user, problem tasks, and software artifact characterize the domain problem. Data abstractions and operation tasks are in the data and operation abstraction layer. Representation as well as navigation and interaction belong to the encoding and interaction technique layer. Finally, implementation corresponds to the algorithm layer. The resulting model is depicted in Figure 1.

These two steps provide an overview of the main factors. In order to understand their relations they have to be detailed and linked. This is supported by the nested structure of the model and by the blocks and guidelines. Table 1 details the factors from Figure 1 with possible instantiations where each factor is marked with the color from the corresponding layer. This list does not claim to be complete. Rather, it is open for extension by other researchers conducting controlled experiments in the field of software visualization.

In a problem-driven approach the experimenter has to define the scope of the domain. In software visualization, a *user* has a specific *role*, a certain *background*, previous *knowledge*, and *circumstances*. The user solves a *problem task* with a *software artifact* where the artifact has a *type* and a *size*, and represents a specific *aspect* of a software system. To abstract from the domain, the necessary information from the software artifact is extracted into a suitable *data abstraction* (implementation). The problem task is divided into several *operation tasks*. On the next layer, the data is represented using a certain *technique* in a certain *dimension*. The operation tasks are processed with *navigation and interaction techniques* supported by *input* and *output* devices. The final layer contains the *implementation*. The representation and interaction techniques have to be implemented with an *algorithm* on a *platform* processing the *data* from the software artifact. The visualization process might be *full-*, *semi-*, or *not automated* at all.

4 PLANNING A SERIES OF CONTROLLED EXPERIMENTS

We plan to conduct a series of experiments investigating the influence of dimensionality. Thus, our research aims at the encoding and interaction technique layer. In the model, dimensionality is a subfactor of representation and might be influenced by several other factors. Apart from group matching and randomization, these factors are purposely either varied, kept constant, or measured (Siegmund, 2012). To vary the factor, it turns into an *independent variable*, whose value is intentionally changed. To reduce or at least to minimize the influence of the other factors, they are transformed into *controlled variables* and have to be kept constant. The remaining factors being difficult or not possible to control are measured to analyze their influence on the result.

As an example, imagine a controlled experiment with the following research question: Does an inherent 3D software visualization reduce time to solve software engineering tasks, compared to a 2D software visualization? In the derived hypothesis *time* is the dependent variable and *dimensionality* the independent one. We apply a between-subjects design. That means, there are a control group (2D) and an experimental group (3D) where every participant is member of only one group. In order to isolate dimensionality as a factor under study we have to keep the other factors constant or at least quasi-constant. The participants act in the role of a *developer* and solve two typical problem tasks, such as finding a bug or identifying a dominating class. The tasks are detailed with corresponding operation tasks. The visualizations are *automatically* generated from *source code* of a *medium-sized* software artifact representing its *structure*. The 2D and the inherent 3D visualization have to be as similar as possible only differing in dimensionality. A suitable representation is a *graph* respectively a nested node-link technique with corresponding layout algorithms and shapes for 2D (e.g., rectangle) and 3D (e.g., cuboid). To solve their tasks, the participants should gain an *overview*, *zoom in* and *out*, *filter*, and identify *relations* in the visualization. To overcome the interaction barrier between 2D and 3D input devices, both visualizations are controlled

Table 1: Possible influence factors on the effectiveness of a software visualization.

Factor/Sub-Factor	Examples for possible instantiations
User	
Role	Manager, Requirements Engineer, Architect, Developer, Tester, Maintainer, Reengineer, Documenter, Consultant, Team, Researcher (Storey et al., 2005)
Background	Age, Gender, Color Blindness, Ability of Stereoscopic Viewing
Knowledge	Education, Programming Experience, Domain Knowledge
Circumstances	Occupation, Familiarity with Study Object/Tools (Siegmund, 2012)
Task	
Problem	Development, Maintenance, Re-Engineering, Reverse Engineering, Software Process Management, Marketing, Test, Documentation (Maletic et al., 2002)
Operation	Retrieve Value, Filter, Compute Derived Value, Find Extremum, Sort, Determine Range, Characterize Distribution, Find Anomalies, Cluster, Correlate (Amar et al., 2005)
Software Artifact	
Type	Requirements, Architecture, Source Code, Stack Trace, Revision History
Size	Small, Medium, Large (Wettel et al., 2011)
Aspect	Structure, Behavior, Evolution (Diehl, 2007)
Representation	
Dimensionality	2D, 2.5D, Augmented 2D, Adapted 2D, Inherent 3D (Stasko and Wehrli, 1993)
Technique	Graph, Tree, Abstract/Real World Metaphor, Decorational/Representational Animation (Gračanin et al., 2005; Diehl, 2007; Höffler and Leutner, 2007)
Navigation & Interaction	
Technique	Overview, Zoom, Filter, Details-on-Demand, Relate, History, Extract (Shneiderman, 1996; Lee et al., 2006; Keim and Schneidewind, 2007; Yi et al., 2007)
Input	Keyboard, Mouse, Gamepad, Flystick, Kinect, Touch Device, Leap Motion, Brain-Computer Interface
Output	Paper, Monitor, Projector, Virtual Reality Environment, Oculus Rift
Implementation	
Algorithm	Radial Layout, Balloon Layout, Treemap, Information Cube, Cone Tree (Herman et al., 2000)
Platform Dependence	Platform Independent, Platform Dependent
Automation	Full, Semi, Manual
Data Abstraction	Famix, Dynamix, Hismo (Nierstrasz et al., 2005; Greevy, 2007; Ducasse et al., 2004)

with a *touch device*. Thus, the difference between 2D input devices (e. g., keyboard and mouse) compared to 3D ones (e. g., flystick) is eliminated. To minimize the differences concerning the environment with regard to the output all participants solve their tasks in the same *virtual reality* environment under equal conditions. Therefore, they wear special 3D glasses to receive the immersive view of the 3D visualization. The participants using the 2D visualization also wear them to eliminate influences due to, e. g., brightness differences. Finally, the remaining factors that are difficult or not to control have to be measured. The participants are tested concerning *color vision deficiency* and *stereoscopic view ability* to control their effect on the participant's performance. With respect to the statistical analysis additional data about *education*, *programming experience*, and *domain knowledge*, i. e. virtual reality, touch devices, and 3D, are collected.

With our structured approach making the influence factors and their relationships explicit, we are able to vary different factors in different experiments while keeping other relevant factors constant or measure their influence on the result. For example, we keep the whole setting as described above and we vary the representation technique, e. g., with another metaphor using the additional dimension to integrate structural and behavioral information or to represent

quality metrics, we change the size of the software artifact, or we use different tasks. Furthermore, we are supported in documenting the experimental design, in analyzing the threats to validity and in comparing our results with other researchers.

5 CONCLUSION AND FUTURE WORK

In this paper we have presented our structured approach for conducting a series of controlled experiments in software visualization. We derived important influence factors from information and software visualization literature and assigned them to Munzner's extended model for visualization design and validation. The domain specific adaption to software visualization helps to relate and to control the influence factors.

With this new perspective on controlled experiments in software visualization, we are able to conduct a series of experiments obtaining comprehensive empirical evidence of the advantages and disadvantages of 3D.

REFERENCES

Amar, R., Eagan, J., and Stasko, J. (2005). Low-level components of analytic activity in information visualization. In *Pro. 2005 IEEE Symp. Inf. Vis.*, page 15. IEEE Computer Society.

Andrews, K. (2008). Evaluation comes in many guises. In *Proc. 2008 AVI Work. BEyond Time Errors Nov. Eval. Method. Inf. Vis.*, pages 8–10.

Balogh, G. and Beszedes, A. (2013). CodeMetropolis – a Minecraft based collaboration tool for developers. In *Proc. 1st IEEE Work. Conf. Softw. Vis.*, pages 1–4.

Basili, V. R., Selby, R. W., and Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Trans. Softw. Eng.*, 12(7):733–743.

Benomar, O. and Poulin, P. (2013). Visualizing Software Dynamicities with Heat Maps. In *1st IEEE Work. Conf. Softw. Vis.* IEEE.

Bull, R. I., Storey, M.-A., Favre, J.-M., and Litoiu, M. (2006). An Architecture to Support Model Driven Software Visualization. In *14th Int. Conf. Progr. Compr.*, pages 100–106. IEEE Computer Society.

Carpendale, S. (2008). Evaluating information visualizations. In Kerren, A., Stasko, J. T., Fekete, J.-D., and North, C., editors, *Information Visualization*, volume 4950, pages 19–45. Springer, Berlin, Heidelberg.

Cockburn, A. and McKenzie, B. (2001). 3D or not 3D?: evaluating the effect of the third dimension in a document management system. In *Proc. SIGCHI Conf. Hum. factors Comput. Syst.*, pages 434–441. ACM.

Cockburn, A. and McKenzie, B. (2002). Evaluating the effectiveness of spatial memory in 2D and 3D physical and virtual environments. In *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, pages 203–210. ACM.

Di Penta, M., Stirewalt, R. E. K., and Kraemer, E. (2007). Designing your Next Empirical Study on Program Comprehension. *15th Int. Conf. Progr. Compr.*, pages 281–285.

Diehl, S. (2007). *Software visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

Ducasse, S., Gǐrba, T., and Favre, J. (2004). Modeling software evolution by treating history as a first class entity. In *Proc. Work. Softw. Evol. Through Transform.*, pages 75–86. Elsevier.

Dwyer, T. (2001). Three dimensional UML using force directed layout. In *Proc. 2001 Asia-Pacific Symp. Inf. Vis.*, volume 9, pages 77–85. Australian Computer Society.

Fittkau, F., Waller, J., Wulf, C., and Hasselbring, W. (2013). Live Trace Visualization for Comprehending Large Software Landscapes : The ExplorViz Approach. In *1st IEEE Work. Conf. Softw. Vis.*, pages 1–4.

Gallagher, K., Hatch, A., and Munro, M. (2008). Software Architecture Visualization: An Evaluation Framework and Its Application. *IEEE Trans. Softw. Eng.*, 34(2):260–270.

Gračanin, D., Matković, K., and Eltoweissy, M. (2005). Software Visualization. *Innov. Syst. Softw. Eng.*, 1(2):59–63.

Greevy, O. (2007). Dynamix - a meta-model to support feature-centric analysis. In *1st Int. Work. FAMIX Moose Reeng.*

Herman, I., Melancon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Vis. Comput. Graph.*, 6(1):24–43.

Höffler, T. N. and Leutner, D. (2007). Instructional animation versus static pictures: A meta-analysis. *Learn. Instr.*, 17(6):722–738.

Huynh-Thu, Q., Barkowsky, M., and Le Callet, P. (2011). The Importance of Visual Attention in Improving the 3D-TV Viewing Experience: Overview and New Perspectives. *IEEE Trans. Broadcast.*, 57(2):421–431.

Irani, P. and Ware, C. (2000). Diagrams based on structural object perception. In *Proc. Work. Conf. Adv. Vis. Interf.*, pages 61–67. ACM.

Irani, P. and Ware, C. (2003). Diagramming information structures using 3D perceptual primitives. *ACM Trans. Comput. Hum. Interact.*, 10(1):1–19.

Isenberg, T., Isenberg, P., Chen, J., Sedlmair, M., and Möller, T. (2013). A systematic review on the practice of evaluating visualization. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2818–27.

Keim, D. A., Kohlhammer, J., Ellis, G., and Mansmann, F. (2010). *Mastering The Information Age-Solving Problems with Visual Analytics*. Eurographics Association.

Keim, D. A. and Schneidewind, J. (2007). Introduction to the Special Issue on Visual Analytics. *SIGKDD Explorations*, 9(2):3–4.

Koike, H. and Chu, H. (1998). How does 3-D visualization work in software engineering?: empirical study of a 3-D version/module visualization system. In *Proc. 20th Int. Conf. Softw. Eng.*, pages 516–519. IEEE Computer Society.

Kosara, R., Healey, C. G., Interrante, V., Laidlaw, D. H., and Ware, C. (2003). User Studies: Why, How, and When? *IEEE Comput. Graph. Appl.*, 23(4):20–25.

Lee, B., Sims Parr, C., Plaisant, C., and Bederson, B. B. (2006). Visualizing Graphs as Trees: Plant a seed and watch it grow. In *13th Int. Symp. Graph Draw.*, volume 3843, pages 516–518. Springer.

Levy, E., Zacks, J., Tversky, B., and Schiano, D. (1996). Gratuitous graphics? Putting preferences in perspective. In *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, pages 42–49. ACM.

Maletic, J., Marcus, A., and Collard, M. (2002). A task oriented view of software visualization. In *1st Int. Work. Vis. Softw. Underst. Anal.*, pages 32–40. IEEE Compututer Society.

Meyer, M., Sedlmair, M., and Munzner, T. (2012). The four-level nested model revisited: blocks and guidelines. In *Proc. 2012 Work. BEyond Time Errors Nov. Eval. Method. Vis.*, pages 1–6.

Müller, R., Kovacs, P., Schilbach, J., and Eisenecker, U. (2011). Generative Software Visualizaion: Automatic Generation of User-Specific Visualisations. In *Proc. Int. Work. Digit. Eng.*, pages 45–49.

Munzner, T. (2009). A nested model for visualization design and validation. *IEEE Trans. Vis. Comput. Graph.*, 15(6):921–928.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123.

Nierstrasz, O., Ducasse, S., and Gĭrba, T. (2005). The story of moose: an agile reengineering environment. In *Proc. 10th Eur. Softw. Eng. Conf.*, pages 1–10. ACM.

Pfleeger, S. L. (1995). Experimental design and analysis in software engineering. *Ann. Softw. Eng.*, 1(1):219–253.

Price, B. A., Baecker, R. M., and Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *J. Vis. Lang. Comput.*, 4(3):211–266.

Roman, G.-C. and Cox, K. C. (1993). A taxonomy of program visualization systems. *Computer*, 26(12):11–24.

Sensalire, M., Ogao, P., and Telea, A. (2009). Evaluation of software visualization tools: Lessons learned. In *5th Int. Work. Vis. Softw. Underst. Anal.*, pages 19–26.

Sharif, B. and Jetty, G. (2013). An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension. In *1st IEEE Work. Conf. Softw. Vis.*

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. 1996 IEEE Symp. Vis. Lang.*, pages 336–343.

Siegmund, J. (2012). *Framework for Measuring Program Comprehension*. Phd thesis, Otto-von-Guericke-Universität Magdeburg.

Sjoberg, D. I. K., Dybå, T., and Jorgensen, M. (2007). The Future of Empirical Methods in Software Engineering Research. In *Futur. Softw. Eng. (FOSE '07)*, pages 358–378.

Smisek, J., Jancosek, M., and Pajdla, T. (2011). 3D with Kinect. In *2011 IEEE Int. Conf. Comput. Vis. Work.*, pages 1154–1160.

Stasko, J. and Patterson, C. (1993). Understanding and Characterizing Program Visualization Systems. Technical report, Georgia Institute of Technology, Atlanta.

Stasko, J. and Wehrli, J. (1993). Three-dimensional computation visualization. *Proc. 1993 IEEE Symp. Vis. Lang.*, pages 100–107.

Storey, M.-A. D., Čubranić, D., and German, D. M. (2005). On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proc. 2005 ACM Symp. Softw. Vis.*, pages 193–202. ACM.

Teyseyre, A. R. and Campo, M. R. (2009). An overview of 3D software visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(1):87–105.

Tichy, W. and Padberg, F. (2007). Empirische Methodik in der Softwaretechnik im Allgemeinen und bei der Software-Visualisierung im Besonderen. In *Softw. Eng. 2007 Beitr. Work.*, pages 211–222. Gesellschaft für Informatik.

Waller, J., Wulf, C., Fittkau, F., Döhring, P., and Hasselbring, W. (2013). SynchroVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *1st IEEE Work. Conf. Softw. Vis.*, pages 7–10.

Ware, C. and Franck, G. (1994). Viewing a graph in a virtual reality display is three times as good as a 2D diagram. *IEEE Symp. Vis. Lang.*, pages 182–183.

Ware, C. and Franck, G. (1996). Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Trans. Graph.*, 15(2):121–140.

Ware, C., Hui, D., and Franck, G. (1993). Visualizing object oriented software in three dimensions. In *Proc. 1993 Conf. Cent. Adv. Stud. Collab. Res. Softw. Eng.*, pages 612–620. IBM Press.

Ware, C. and Mitchell, P. (2008). Visualizing graphs in three dimensions. *ACM Trans. Appl. Percept.*, 5(1):1–15.

Weichert, F., Bachmann, D., Rudak, B., and Fisseler, D. (2013). Analysis of the accuracy and robustness of the leap motion controller. *Sensors*, 13:6380–6393.

Wettel, R., Lanza, M., and Robbes, R. (2011). Software systems as cities: A controlled experiment. In *Proc. 33rd Int. Conf. Softw. Eng.*, pages 551–560. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.

Yi, J. S., ah Kang, Y., Stasko, J. T., and Jacko, J. A. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1224–1231.

Zorrilla, M., Martin, A., Sanchez, J. R., Tamayo, I., and Olaizola, I. G. (2013). HTML5-based system for interoperable 3D digital home applications. *Multimed. Tools Appl.*, pages 1–21.

5.2 Summary

As stated in Section 3.2, empirical evaluations in software visualization, especially controlled experiments, are quite rare. One reason is the fact that performing such evaluations in general, and particularly performing them in a way that minimizes the threats to validity, is hard to accomplish. The contributions of this paper help to overcome these challenges. It presents a structured approach on how to conduct a series of experiments in software visualization including a theoretical model to control influence factors. Hence, it is possible to give empirical evidence for advantages and disadvantages of software visualizations, especially for 2D vs. 3D.

The structured approach is the second precondition for answering ***SQ3:*** *What role does the factor dimensionality play in solving software engineering tasks?* (see Section 1.2). It is used to design the corresponding controlled experiment in Chapter 6.

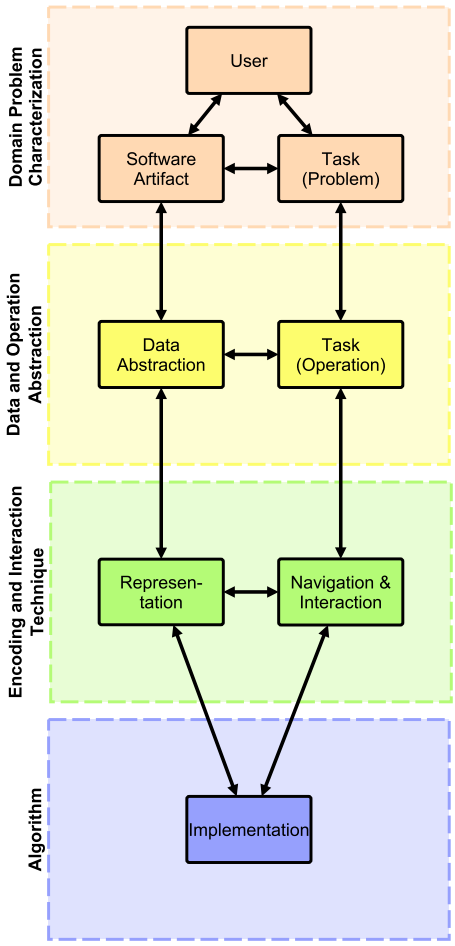


Figure 5.1: Domain specific adaption of Munzner's extended model [Munzner 2009; Meyer et al. 2012] for software visualization [Müller et al. 2014a].

The development of the theoretical model is a domain specific adaption from the field of information visualization to software visualization and comprises two steps. First, the main influence factors are derived from several software visualization taxonomies [Myers 1990;

Stasko and Patterson 1993; Roman and Cox 1993; Price et al. 1993; Maletic et al. 2002; Storey et al. 2005; Gallagher et al. 2005]. These factors are *user*, *task*, *software artifact*, *navigation and interaction*, *representation*, and *implementation*. Second, these factors are assigned to Munzner's extended model for design and validation of visualizations [Munzner 2009; Meyer et al. 2012]. Thus, user, problem tasks, and software artifact characterize the *domain problem*. Data abstractions and operation tasks are in the *data and operation abstraction layer*. Representation as well as navigation and interaction belong to the *encoding and interaction technique layer*. Finally, implementation corresponds to the *algorithm layer*. The resulting model is depicted in Figure 5.1. The main factors from Figure 5.1 are detailed with possible instantiations in Table 5.1 where each factor is marked with the color from the corresponding layer. This list does not claim to be complete. Rather, it is designed as an open list to be extended by other researchers conducting controlled experiments in the field of software visualization.

Table 5.1: Possible influence factors on the effectiveness of a software visualization [Müller et al. 2014a].

Factor/Sub-Factor	Examples for possible instantiations
User	
Role	Manager, Requirements Engineer, Architect, Developer, Tester, Maintainer, Reengineer, Documenter, Consultant, Team, Researcher [Storey et al. 2005]
Background	Age, Gender, Color Blindness, Ability of Stereoscopic Viewing
Knowledge	Education, Programming Experience, Domain Knowledge
Circumstances	Occupation, Familiarity with Study Object/Tools [Siegmund 2012]
Task	
Problem	Development, Maintenance, Re-Engineering, Reverse Engineering, Software Process Management, Marketing, Test, Documentation [Maletic et al. 2002]
Operation	Retrieve Value, Filter, Compute Derived Value, Find Extremum, Sort, Determine Range, Characterize Distribution, Find Anomalies, Cluster, Correlate [Amar and Stasko 2004]
Software Artifact	
Type	Requirements, Architecture, Source Code, Stack Trace, Revision History
Size	Small, Medium, Large [Wettel et al. 2011]
Aspect	Structure, Behavior, Evolution [Diehl 2007]
Representation	
Dimensionality	2D, 2.5D, Augmented 2D, Adapted 2D, Inherent 3D Stasko and Patterson [1993]
Technique	Graph, Tree, Abstract/Real World Metaphor, Decorational/Representational Animation [Gračanin et al. 2005; Diehl 2007; Höffler and Leutner 2007]

Continued on next page

Table 5.1 – continued from previous page

Factor/Sub-Factor	Examples for possible instantiations
Navigation & Interaction	
Technique	Overview, Zoom, Filter, Details-on-Demand, Relate, History, Extract [Shneiderman 1996; Lee et al. 2005; Keim and Schneidewind 2007; Yi et al. 2007]
Input	Keyboard, Mouse, Gamepad, Flystick, Kinect, Touch Device, Leap Motion, Brain-Computer Interface
Output	Paper, Monitor, Projector, Virtual Reality Environment, Oculus Rift
Implementation	
Algorithm	Radial Layout, Balloon Layout, Treemap, Information Cube, Cone Tree [Herman et al. 2000]
Platform Dependence	Platform Independent, Platform Dependent
Automation	Full, Semi, Manual
Data Abstraction	Famix, Dynamix, Hismo [Nierstrasz et al. 2005; Greevy 2007; Ducasse et al. 2004]

6 Controlled Experiment

Müller, Richard, Pascal Kovacs, Jan Schilbach, and Dirk Zeckzer. 2014. “How to Master Challenges in Experimental Evaluation of 2D versus 3D Software Visualizations.” In *IEEE VIS 2014 International Workshop on 3DVis: Does 3D Really Make Sense for Data Visualization?*, Paris, France.

6.1 How to Master Challenges in Experimental Evaluation of 2D versus 3D Software Visualizations

How to Master Challenges in Experimental Evaluation of 2D versus 3D Software Visualizations

Richard Müller*
Information Systems Institute
University of Leipzig

Pascal Kovacs†
Information Systems Institute
University of Leipzig

Jan Schilbach‡
Information Systems Institute
University of Leipzig

Dirk Zeckzer§
Institute of Computer Science
University of Leipzig

ABSTRACT

Software visualizations in 3D and virtual reality are an interesting and debated research topic in academia. However, the benefits and drawbacks of 3D software visualizations in immersive environments compared to its 2D counterparts are not very well understood due to the lack of empirical evaluations. The challenge is to plan valid experiments with analogous 2D and 3D visualization techniques, while avoiding various influence factors and minimizing the threats to validity. In this paper, we present an experiment as part of a series using a structured approach to meet these challenges.

Index Terms: Information Interfaces and Presentation [H.5.1]: Multimedia Information Systems—Evaluation/methodology Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Virtual reality

1 INTRODUCTION

Performing controlled experiments in software visualization while minimizing the threats to validity is hard to accomplish. There are many influence factors such as the user, the task, the visualized software artifact, its representation with the corresponding navigation and interaction techniques as well as the implementation.

Furthermore, to derive general statements about the benefits or drawbacks of visualizations one single experiment is not sufficient. Rather, a series of experiments is needed [9]. Thus, selected influence factors should be varied in different experiments while keeping the remaining factors constant or measure their influence.

This paper has two contributions. (1) We present an experiment as part of such a series to answer the question: Does the additional dimension in inherent 3D [17] software visualizations lead to advantages in solving software engineering tasks? (2) Further, we explain how we met the challenge to control the influence factors in comparing 2D vs. 3D software visualizations.

2 RELATED WORK

To control the different influence factors and to minimize the threats to validity during an experiment and over the whole series, we used a structured approach for planning and conducting controlled experiments in software visualization [12]. The approach is based on the extended process model for design and validation of visualizations by Munzner et al. [13, 10].

Further, we considered the lessons learned from other experiments in software visualization [15], hints, guidelines, and frameworks [16, 5, 20].

Important prior work in conducting controlled experiments comparing 2D and 3D information and software visualizations has been performed [18, 6, 19]. However, all of these controlled experiments were not performed as part of a series. Therefore, deducing general statements about advantages and disadvantages of the third dimension in software visualization is still difficult.

*e-mail: rmueller@wifa.uni-leipzig.de
†e-mail: kovacs@wifa.uni-leipzig.de
‡e-mail: schilbach@wifa.uni-leipzig.de
§e-mail: zeckzer@informatik.uni-leipzig.de

3 THE EXPERIMENT IN A NUTSHELL

In the experiment, we investigated the comprehension of a medium-sized software system focusing on three main research questions:

1. Does an inherent 3D software visualization reduce the time to solve a task, compared to a 2D software visualization?
2. Does an inherent 3D software visualization increase the correctness of the solution of a task, compared to a 2D software visualization?
3. Does an inherent 3D software visualization require more interaction to solve a task, compared to a 2D software visualization?

From these research questions, we derived the directed hypotheses given in Table 1. All hypotheses refer to a software comprehension task of medium-sized software systems.

Our research questions and hypotheses aim at comparing 2D versus 3D software visualizations. Therefore, the *dimension* of the software visualization is the independent variable that is to be varied. To verify our hypotheses, we measure the following dependent variables in our experiment: the *time* a participant needs to complete the task, the *correctness* of the participant's solution, and the *click time* the participant spends for interaction.

In order to measure the presumed effect, the random sample was divided into two groups. Both groups had to solve the same tasks, but the control group used a 2D and the experimental group used an inherent 3D visualization. We applied a between-subjects design, i.e., every participant was member of only one group.

4 CONTROLLING THE INFLUENCE FACTORS

Next, we describe how we control the influence factors in the experiment. Table 2 shows all considered factors and the way how we control them, i.e., whether we hold them constant or measure them.

Alternative Hypothesis	Null Hypothesis
$H1_1$: The third dimension decreases time to solve a task.	$H1_0$: The third dimension does not decrease time to solve a task.
$H2_1$: The third dimension increases correctness of solved tasks.	$H2_0$: The third dimension does not increase correctness of solved tasks.
$H3_1$: The third dimension increases interaction required to solve a task.	$H3_0$: The third dimension does not increase the interaction required to solve a task.

Table 1: Directed hypotheses operationalizing the research questions.

Factor/Sub-Factor	Characteristic	Control
User		
Role	*Developer, Maintainer*	measured
Background	*20-40 years*	measured
	Male, Female	measured
	Color Blindness	measured
	Ability of Stereoscopic Viewing	measured
Knowledge	*Bachelor, Master, PhD, Post Doc*	measured
	Programming Experience	measured
	Domain Knowledge	measured
Circumstances	Occupation	measured
	Familiarity with Study Object	measured
	Familiarity with Study Tools	measured
Task		
Problem	*T1: Find a method, T2: Identify dependencies*	constant
Operation	*Retrieve Value, Filter, Find Extremum*	constant
Software Artifact		
Type	*Source Code: Java*	constant
Size	*Medium: 200K LOC*	constant
Aspect	*Structure*	constant
Representation		
Dimensionality	*2D, Inherent 3D*	varied
Technique	*Nested Node-Link*	constant
Navigation & Interaction		
Technique	*Overview, Zoom, Details-on-Demand, Relate*	constant
Input	*Tablet*	constant
Output	*Virtual Reality Environment*	constant
Implementation		
Algorithm	*Force-Directed Layout*	constant
Platform Dependence	*Platform Independent*	constant
Automation	*Full*	constant
Data Abstraction	*Famix*	constant

Table 2: Instance of the model: Comparing a 2D vs. an inherent 3D software visualization [12].

4.1 User

Attributes of the user that might have an influence on the results are color blindness, the ability of stereoscopic viewing, and the individual experience in software development, software visualization, virtual reality, and using a tablet. Furthermore, the frequency of activities in playing 3D games, watching 3D movies in cinema or on TV, as well as 3D modeling might also have an effect. In order to check whether these factors are distributed almost equally among both groups, we collected the necessary data and included it in our analysis.

Furthermore, the user experience was measured with a questionnaire. The participants were assisted by pairs of words, in which each pair represented contrasting judgments of the visualization. This assessment was derived from the AttrakDiff questionnaire [8]. Finally, we asked for positive and negative aspects of the visualizations and for suggested improvements.

4.2 Task

A typical scenario in software development and re-engineering is the identification of dependencies in complex software systems to implement new features or to refactor code. The advantage of these tasks is that a deep understanding of the software is not necessary, whereby the training of the participant has less effect on the results. Thus, the effort to train the participants and the effort needed to review the experiment by others are reduced to a minimum [7].

Nevertheless, the participant must have at least some knowledge about the system to solve this task in a proper way. Hence, the first task of the experiment was to identify a single method using a visualization. This gave a first insight into the visualized system. The task was solved, if the participant identified the correct method. In the second task, the participant had to identify six dependencies of this method on other methods and attributes. To solve the second task, the participant had to identify all six dependencies. Every missing or wrongly identified dependency was rated as a single mistake. The first visualization did not contain any dependency to avoid solving the second task at the same time. An additional benefit of this combination of tasks is, that the participant did not have to search for the starting point of the second task, whereby the search had no influence on the result of identifying the dependencies.

4.3 Software Artifact

The analyzed software system is the Apache Tomcat Project [1] being a good example of medium-sized and freely available software systems. To handle the visualization in a proper way, the size of the visualized code was limited to three bigger packages, selected according to the amount of the contained classes, methods, attributes, and relations. Presenting only a subset of the whole system imitates a zoom interaction by the user and limits the time needed per participant.

4.4 Representation

A suitable representation technique of the software artifact to solve the two tasks is a nested node-link visualization. This decision has the advantage that corresponding shapes for 2D and 3D visualizations exist, e.g., a rectangle in 2D is a cuboid in 3D. Additionally, the containment relation, typical for the structure of software systems, was realized by nested elements. The packages, classes, methods, and attributes were mapped to nodes, while the invocation relations were mapped to edges. The complete mapping is described in Table 3.

Entity	2D	3D	Color
Package	Rectangle	Cuboid	Pink
Class	Rectangle	Cuboid	Red
Method	Ellipse	Sphere	Blue
Attribute	Ellipse	Sphere	Green
Method call	Line	Tube	Blue
Attribute call	Line	Tube	Green

Table 3: Mapping of software entities in 2D and 3D.

4.5 Navigation & Interaction

The device for interacting with the visualization was a tablet (ODYS Xelio, Android 4) that contained a touch-surface and communicated with the controlling computer via WLAN. To minimize the effect of the interaction on the outcome of the experiment, a customized interface was implemented as an app for the tablet, which provides similar interactions for both, the 2D and the 3D visualizations, using the touch screen. Figure 1 shows the graphical user interface to control 3D visualizations, containing buttons for moving left, right, up, down, zooming in/out, rotating around the x-, y-, z-axis, and resetting the position of the visualization. For 2D visualizations, the buttons for rotation were omitted, which was the only difference in interaction between the 2D and the 3D visualization. The app uses standard HTTP-GET requests, which were sent to the web server of the InstantPlayer, to control the user's movement in the scene.

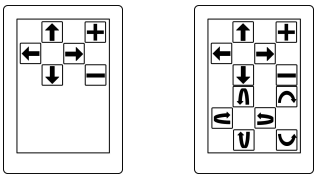


Figure 1: Mockups of the graphical user interface for 2D and 3D.

The experiment was performed in a virtual reality environment with a powerwall composed of three connected screens as the output device. The light in the lab was controlled by darkening all windows. The image for the wall is generated by two projectors per screen using the INFITEC-method. Therefore, the participants had to wear special 3D glasses to receive an immersive view of the 3D visualization. To eliminate the influence of the glasses on the results, the participants using the 2D visualization also had to wear some.

4.6 Implementation

For the 3D visualizations, the layout was computed with the FDL tool [21], whereas the FDP tool of Graphviz [3] was used for 2D visualizations. Both tools apply a force-directed layout algorithm. Extracts from the resulting visualizations for the second task are shown in Figure 2.

We used a generator to create the visualizations for the experiment automatically [11]. The generator utilizes the generative and the model driven paradigm to process different input formats and transform them into different output formats with minimum configuration effort. The input for both visualizations was the source code of the Apache Tomcat Project parsed into a Famix model [14]. The output format of the generator was Extensible 3D (X3D) [2]. X3D served as format for both visualizations and is platform independent.

The visualizations were rendered by the InstantPlayer [4]. It has a wide coverage of the X3D-standard and supports different output device configurations, including stereoscopic virtual reality environments. Additionally, it comes with a built-in web server, with the ability to access and modify the scene via standard HTTP-GET requests.

5 Results

We measured the influence of the independent variable *dimension* on our dependent variables *time*, *correctness*, and *click time*. Due to the sample size of 18 we applied the non-parametric Mann-Whitney U-Test to check our hypotheses. We chose a significance level of $\alpha = 0.05$ corresponding to a 95% confidence interval. Two observations had to be partially excluded from the results due to technical problems, one measurement of time and one of click time. Table 4 shows the detailed results of the experiment.

For the first task, none of the differences in time, in correctness, or in the amount of interaction were significant ($p_{\text{time},1} = 0.271, p_{\text{corr},1} = 0.235, p_{\text{click},1} = 0.303$). The results indicate, however, that the experimental group (3D) took less time (-18.08%), was more accurate ($+28.57\%$), and also used less interaction (-10.59%).

For the second task, time and interaction were significantly different. There was an increase in time of 42.29% from the control group to the experimental group. There was an increase in correctness of $+14.28\%$ in the second task from the control group to the experimental group. While the null hypothesis could not be rejected for time and correctness ($p_{\text{corr},2} = 0.405$), it could be rejected for interaction. In the second task, there was a significant increase of

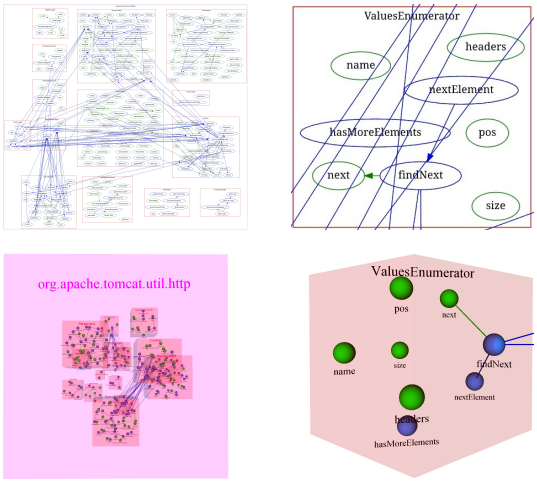


Figure 2: Extracts from 2D and 3D nested node-link visualization (overview/zoom).

111.41% of interactions from the control group to the experimental group ($p_{\text{click},2} = 0.030$). Thus, there is a strong indication that 3D does not decrease the time to analyze a software system. There was no significant difference in correctness. The results from the experiment rather suggest the following conclusions:

Time: The third dimension does not decrease the time to solve a software comprehension task in medium-sized software systems. The decrease of 18.08% for the first task was not significant. Only the increase of 42.29% for the second task was significant.

Correctness: The third dimension does not increase the correctness of a solved software comprehension task in medium-sized software systems. Even though the 3D group made less mistakes, this difference was not significant.

Click time: The third dimension increases the interaction required to solve a software comprehension task in medium-sized software systems. There was an increase of 111.41%.

User experience: In the questionnaire regarding the user experience, all participants were included, also the formerly excluded outliers. The experimental group, i.e., the 3D group, rated slightly more positive. The majority of this group experienced the inherent 3D visualization more motivating, less demanding, more inventive, more innovative, and more clearly structured.

6 Discussion

In this section, we discuss the results of the experiment and the feasibility of the structured approach.

6.1 Experiment

The main objective of the experiment was to investigate the influence of the dimensionality in software visualizations on time, correctness, and interaction. From the three hypotheses only the influence of interaction was significant in the second task. This effect could be explained by the increased amount of degrees of freedom necessary for navigation in inherent 3D software visualizations. In addition to the left, right, up, down, zoom in/out and reset controls there are rotations around the x-, y-, and z-axis in both directions each.

Due to the small sample size and the choice of very basic tasks, the results have to be interpreted with caution. It is possible that out-

	Time [s]				Correctness [%]				Click Time [s]			
	Task 1		Task 2		Task 1		Task 2		Task 1		Task 2	
	2D	3D	2D	3D	2D	3D	2D	3D	2D	3D	2D	3D
n	9	8	9	8	9	9	9	9	8	9	8	9
min	20.86	16.31	29.38	72.98	0	100	0	50.02	1.37	1.10	0.80	1.48
max	130.29	102.80	247.05	191.19	100	100	100	100	4.27	4.84	15.06	26.16
median	49.42	40.86	71.92	128.93	100	100	100	100	2.73	1.88	2.47	6.59
mean	59.09	48.40	89.33	127.11	77.78	100	77.78	88.89	2.77	2.48	3.88	8.21
difference [%]		**−18.08**		**+42.29**		**+28.57**		**+14.28**		**−10.59**		**+111.41**
σ	33.27	31.43	63.92	38.67	44.10	0	39.96	18.63	1.10	1.48	4.69	7.39

Table 4: Descriptive statistics of the dependent variables.

liers distort the results. In future experiments we aim at a minimum of 20 participants per group to gain more reliable results. Additionally, the choice of tasks should be improved in the future. In our experimental design the tasks were not independent of each other. Therefore, if a participant made an error in the first task, the second one mostly resulted in an error, too. For this reason, the different tasks should be all mutually independent. Moreover, there should be more than two tasks in such an experiment. Finally, participants should be assigned to the groups before the experiment based on a pre-questionnaire in order to assure an equal distribution among groups from the beginning.

6.2 Approach

Nonetheless, the experiment shows, that our approach is suitable to plan and conduct comparing experiments in the field of software visualization. The structured approach was helpful to determine the relevant influence factors. Further, we were able to control these factors either by holding them constant or by measuring them. Hence, we measured relevant characteristics from the participants. We used the same tasks, the same software artifact, and the same virtual reality environment. Moreover, we applied a similar visualization (nested node-link) as well as navigation and interaction technique (specific interface) reducing the differences between both groups to a minimum. Finally, we assured that both visualizations contained the same information by using the generator including Famix and X3D.

7 Conclusion and Future Work

In this paper, we presented an experiment examining the influence of the third dimension in software visualization. The underlying structured approach supported us in planning and conducting this experiment and creating comparable empirical data for our series. We outlined different approaches to overcome specific challenges comparing 2D vs. 3D software visualizations.

For the moment, we state that the third dimension does not show significant advantages in solving a task for medium-sized software systems. But as discussed in the previous section, these results have to be interpreted with caution.

In the future, we consider the findings from this experiment and plan to conduct further experiments to investigate other factors responsible for the advantages and disadvantages of software visualizations.

References

[1] Apache Tomcat. `http://tomcat.apache.org/`. Accessed: 2014-09-18.

[2] Extensible 3D (X3D). `http://www.web3d.org/x3d/`. Accessed: 2014-09-18.

[3] Graphviz. `http://www.graphviz.org/`. Accessed: 2014-09-18.

[4] InstantReality. `http://www.instantreality.org/`. Accessed: 2014-09-18.

[5] S. Carpendale. Evaluating information visualizations. In A. Kerren, J. Stasko, J.-D. Fekete, and C. North, editors, *Inf. Vis. Human-Centered Issues Perspect.*, pages 19–45, Berlin, Heidelberg, 2008. Springer.

[6] A. Cockburn and B. McKenzie. 3D or not 3D?: evaluating the effect of the third dimension in a document management system. In *Proc. SIGCHI Conf. Hum. factors Comput. Syst.*, pages 434–441. ACM, Mar. 2001.

[7] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your Next Empirical Study on Program Comprehension. In *15th Int. Conf. Progr. Compr.*, pages 281–285, 2007.

[8] M. Hassenzahl, A. Platz, M. Burmester, and K. Lehner. Hedonic and ergonomic quality aspects determine a software's appeal. In *CHI*, pages 201–208, 2000.

[9] P. Irani and C. Ware. Diagramming information structures using 3D perceptual primitives. *ACM Trans. Comput. Interact.*, 10(1):1–19, 2003.

[10] M. Meyer, M. Sedlmair, and T. Munzner. The four-level nested model revisited: blocks and guidelines. In *Work. BEyond time errors Nov. Eval. methods Inf. Vis.*, pages 1–6, 2012.

[11] R. Müller, P. Kovacs, J. Schilbach, and U. Eisenecker. Generative Software Visualization: Automatic Generation of User-Specific Visualizations. In *Proc. Int. Work. Digit. Eng.*, pages 45–49, Magdeburg, Germany, 2011.

[12] R. Müller, P. Kovacs, J. Schilbach, U. Eisenecker, D. Zeckzer, and G. Scheuermann. A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization. In *Proc. 5th Int. Conf. Vis. Theory Appl.*, pages 204–209, Lisbon, Portugal, 2014.

[13] T. Munzner. A nested model for visualization design and validation. *IEEE Trans. Vis. Comput. Graph.*, 15(6):921–928, 2009.

[14] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. In *Proc. 10th Eur. Softw. Eng. Conf. held jointly with 13th SIGSOFT Int. Symp. Found. Softw. Eng.*, pages 1–10, New York, USA, 2005. ACM.

[15] M. Sensalire, P. Ogao, and A. Telea. Evaluation of software visualization tools: Lessons learned. In *5th Int. Work. Vis. Softw. Underst. Anal.*, pages 19–26. IEEE, 2009.

[16] D. I. K. Sjø berg, T. Dybå, and M. Jø rgensen. The Future of Empirical Methods in Software Engineering Research. In *Fut. Softw. Eng.*, pages 358–378. IEEE, May 2007.

[17] J. Stasko and J. Wehrli. Three-dimensional computation visualization. *Proc. 1993 IEEE Symp. Vis. Lang.*, pages 100–107, 1993.

[18] C. Ware and G. Franck. Viewing a graph in a virtual reality display is three times as good as a 2D diagram. *IEEE Symp. Vis. Lang.*, pages 182–183, 1994.

[19] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proc. 33rd Int. Conf. Softw. Eng.*, pages 551–560, Waikiki, Honolulu, USA, 2011. ACM.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

[21] D. Zeckzer, R. Kalcklösch, L. Schröder, H. Hagen, and T. Klein. Analyzing the reliability of communication between software entities using a 3D visualization of clustered graphs. In *Proc. 4th ACM Symp. Softw. Vis.*, pages 37–46, New York, USA, Sept. 2008. ACM.

6.2 Summary

The benefits and drawbacks of 3D software visualizations in immersive environments compared to their 2D counterparts are not very well understood due to the lack of empirical evaluations. The challenge is to plan valid experiments with corresponding 2D and 3D visualization techniques, while controlling various influence factors and minimizing the threats to validity. In this controlled experiment these challenges are met using the generator [Müller et al. 2011] and the structured approach [Müller et al. 2014a] described in the preceding Chapters 4 and 5. The controlled experiment investigates the influence of *dimensionality* in software visualization.
The experiment refers to ***SQ3:*** *What role does the factor dimensionality play in solving software engineering tasks?* (see Section 1.2) and details it with the following research questions:

- *Time:* Does an inherent 3D software visualization reduce the time to solve a task, compared to a 2D software visualization?
- *Correctness:* Does an inherent 3D software visualization increase the correctness of the solution of a task, compared to a 2D software visualization?
- *Interaction:* Does an inherent 3D software visualization require more interaction to solve a task, compared to a 2D software visualization?

From these research questions, the directed hypotheses given in Table 6.1 are derived. All hypotheses refer to a software comprehension task of medium-sized software systems.

Table 6.1: Directed hypotheses operationalizing the research questions [Müller et al. 2014b].

Alternative Hypothesis		**Null Hypothesis**	
$H1_1$:	The third dimension decreases time to solve a task.	$H1_0$:	The third dimension does not decrease time to solve a task.
$H2_1$:	The third dimension increases correctness of solved tasks.	$H2_0$:	The third dimension does not increase correctness of solved tasks.
$H3_1$:	The third dimension increases interaction required to solve a task.	$H3_0$:	The third dimension does not increase the interaction required to solve a task.

The research questions and hypotheses aim at comparing 2D versus 3D software visualizations. Therefore, the *dimension* of the software visualization is the independent variable that is varied. To verify the hypotheses, the following dependent variables are measured in the experiment: the *time* a participant needs to complete the task, the *correctness* of the participant's solution, and the *click time* the participant spends for interaction. Furthermore, the random sample was divided into two groups. Both groups had to solve the same tasks, but

the control group used a 2D and the experimental group used an inherent 3D visualization. A between-subjects design was applied, i.e., every participant was member of only one group. The visualizations for the experiment were produced with the generator [Müller et al. 2011]. Moreover, the structured approach was applied in the experiment [Müller et al. 2014a]. The approach helped to plan the experiment, to determine relevant influence factors, and to control these factors. Table 6.2 shows all considered factors and the way how they were controlled, i.e., whether they were hold constant or measured.

Table 6.2: Instance of the theoretical model for comparing a 2D vs. an inherent 3D software visualization. [Müller et al. 2014b].

Factor/Sub-Factor	Characteristic	Control
User		
Role	*Developer, Maintainer*	measured
Background	*20-40 years*	measured
	Male, Female	measured
	Color Blindness	measured
	Ability of Stereoscopic Viewing	measured
Knowledge	*Bachelor, Master, PhD, Post Doc*	measured
	Programming Experience	measured
	Domain Knowledge	measured
Circumstances	Occupation	measured
	Familiarity with Study Object	measured
	Familiarity with Study Tools	measured
Task		
Problem	*T1: Find a method, T2: Identify dependencies*	constant
Operation	*Retrieve Value, Filter, Find Extremum*	constant
Software Artifact		
Type	*Source Code: Java*	constant
Size	*Medium: 200K LOC*	constant
Aspect	*Structure*	constant
Representation		
Dimensionality	*2D, Inherent 3D*	varied
Technique	*Nested Node-Link*	constant
Navigation & Interaction		
Technique	*Overview, Zoom, Details-on-Demand, Relate*	constant
Input	*Tablet*	constant
Output	*Virtual Reality Environment*	constant
Implementation		
Algorithm	*Force-Directed Layout*	constant
Platform Dependence	*Platform Independent*	constant
Automation	*Full*	constant
Data Abstraction	*Famix*	constant

The results of the experiment indicate that the third dimension does not decrease the *time* to solve a software comprehension task in medium-sized software systems. The decrease of

18.08% for the first task was not significant. Only the increase of 42.29% for the second task was significant. Furthermore, the third dimension does not increase the *correctness* of the solution of a software comprehension task in medium-sized software systems. Even though the 3D group made less mistakes, this difference was not significant. The third dimension increases the *interaction* required to solve a software comprehension task in medium-sized software systems. There was an increase of 111.41% regarding the time spent for interaction. In the post-questionnaire [Hassenzahl et al. 2000], the experimental group (3D group) rated slightly more positive. They found the 3D visualization more motivating, less demanding, more inventive, more innovative, and more clearly structured.
To conclude, the *dimensionality* of software visualizations is not a decisive factor that shows significant advantages in solving a task for medium-sized software systems. It plays a rather tangential role. To increase its benefit, it has to applied in a useful way. More important seems to be the used visualization technique or metaphor. A suitable metaphor combining 2D and 3D usefully is introduced next, in Chapter 7.

7 The Recursive Disk Metaphor

Müller, Richard, and Dirk Zeckzer. 2015. “The Recursive Disk Metaphor - A Glyph-Based Approach for Software Visualization.” In *Proceedings of the 6th International Conference on Visualization Theory and Applications*, Berlin, Germany.

7.1 The Recursive Disk Metaphor - A Glyph-based Approach for Software Visualization

The Recursive Disk Metaphor
A Glyph-based Approach for Software Visualization

Richard Müller[1], Dirk Zeckzer[2]
[1]*Information Systems Institute, Leipzig University, Leipzig, Germany*
[2]*Institute of Computer Science, Leipzig University, Leipzig, Germany*
rmueller@wifa.uni-leipzig.de, zeckzer@informatik.uni-leipzig.de

Keywords: Software Visualization, Glyph-based Visualization

Abstract: In this paper, we present the recursive disk metaphor, a glyph-based visualization for software visualization. The metaphor represents all important structural aspects and relations of software using nested circular glyphs. The result is a shape with an inner structural consistency and a completely defined orientation. We compare the recursive disk metaphor to other state-of-the-art 2D approaches that visualize structural aspects and relations of software. Further, a case study shows the feasibility and scalability of the approach by visualizing an open source software system in a browser.

1 INTRODUCTION

Software is known to be complex, intangible, and invisible (Gračanin et al., 2005). A major challenge in the field of software visualization is to give the abstract artifact software a shape in order to explore and to understand it.

We present a glyph-based approach to make structural software entities and eventually the whole software system visible. Glyph-based visualization is a form of visual design where a data set is represented by a collection of visual objects referred to as glyphs (Borgo et al., 2013). In more detail,

- *"[...] a glyph is a small visual object that can be used independently and constructively to depict attributes of a data record or the composition of a set of data records;*
- *each glyph [...] can be spatially connected to convey the topological relationships between data records or geometric continuity of the underlying data space; and*
- *glyphs are a type of visual sign that can make use of visual features of other types of signs such as icons, indices and symbols."* (Borgo et al., 2013)

To assemble the shape of software from scratch, we start with the basic structural entities of software, i.e., system, namespaces/packages, classes, methods, and attributes. Further, relations should be shown on demand to avoid visual clutter. Common visualization techniques to represent structure and metrics of software in 2D are node-link diagrams, Cartesian, Voronoi, or circular treemaps, and Sunburst (Caserta and Zendra, 2011). We map the entities to circular glyphs. The spatial location of each glyph is predetermined by the underlying structure of the software, i.e., by the containment relations of the entities. As glyphs may contain other glyphs, they are constructed recursively. For these reasons, we call this approach *recursive disk metaphor*. We decided to dismiss global space-efficiency for local space-efficiency allowing a complete representation of namespaces/packages, classes, inner classes, methods, and attributes. While the resulting visualization is not space-filling as other types of treemaps, it still uses space efficiently by avoiding empty space between the glyphs and by omitting the links. The empty space supports the formation of characteristic patterns that can easily be perceived.

We believe that the application of glyphs holds benefits for software visualization, as one major strength of glyphs is that patterns involving multiple data dimensions may be more easily perceived (Ward, 2008):

1. We get a complete shape for the whole software system representing all important structural entities and their relations. This leads to visually differentiable class glyphs.
2. Design flaws may be easily detectable through certain visual anti-patterns during software quality assessment.

2 RELATED WORK

Glyphs have been successfully applied in 2D software visualization. Chuah and Eick (1998) map software management data to timewheel and infobug glyphs. Pinzger et al. (2005) map structural and evolutionary software metrics to Kiviat diagrams. Boccuzzo and Gall (2007) map structural software metrics to well-known glyphs, such as houses, tables, and spears. The final shape looks either well-shaped or mis-shaped and allows conclusions concerning the software design. Besides the unique visual patterns, all approaches use the benefit of glyphs to view many dimensions of the data simultaneously.

According to Caserta and Zendra (2011) current state-of-the-art techniques to visualize static aspects of software in 2D are Treemap (Shneiderman, 1992), Circular Treemap (Wang et al., 2006), Sunburst (Andrews and Heidegger, 1998; Stasko et al., 2000), Dependency Structure Matrix (Sangal et al., 2005), Hierarchical Edge Bundles (Holten, 2006), Treemap metrics (Holten et al., 2005), Class Blueprint/Polymetric Views (CodeCrawler) (Lanza, 2003; Ducasse and Lanza, 2005), Voronoi Treemap (Balzer et al., 2005), UML (Gutwenger et al., 2003), UML MetricView (Termeer et al., 2005), UML Area of Interest (Byelas and Telea, 2006), and SHriMP Views (Rigi) (Storey et al., 1997). However, all of these techniques do not support all structural entities and relations. A comparison of the completeness between the recursive disk metaphor and these 2D software visualizations of static aspects is shown in Table 1. It shows that the recursive disk metaphor is the only technique that visually represents inner classes and relations.

Table 1: Completeness comparison between recursive disk metaphor and 2D software visualizations of static aspects (+ supported/– not supported).

Technique/ Tool	Package	Class	Inner Class	Method	Attribute	Relations
Treemap	+	+	–	–	–	–
Circular Treemap	+	+	–	–	–	–
Sunburst	+	+	–	–	–	–
Dep. Struc. Mat.	+	–	–	–	–	+
Hier. Edge Bund.	+	+	–	–	–	+
Treemap metrics	+	+	–	+	–	–
CodeCrawler	–	+	–	+	+	+
Voronoi Treemap	+	+	+	+	+	–
UML	+	+	–	+	+	+
UML MetricView	+	+	–	+	+	+
UML Area of Int.	+	+	–	+	+	+
Rigi	+	+	–	+	+	+
Recursive Disk	+	+	+	+	+	+

Although, there are applications of radial layouts (Stasko et al., 2000; Barlow and Neville, 2001; Wang et al., 2006; Fischer et al., 2012), they are not very widespread because of some drawbacks (Burch and Weiskopf, 2014). First, they are not as space-efficient as Cartesian treemaps (McGuffin and Robert, 2010). Second, it is more difficult to estimate and compare areas of circles (Cleveland and McGill, 1984). As stated in the introduction, using a circular, glyph-based approach uses space efficiently while allowing the formation of patterns that facilitate the comparison of the structure.

3 THE RECURSIVE DISK METAPHOR

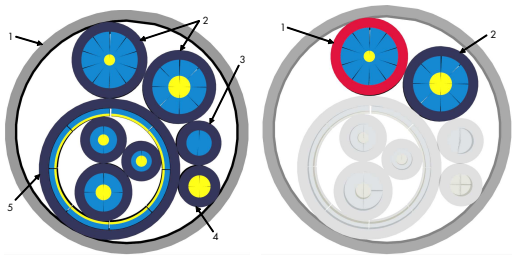
In general, the recursive disk metaphor is applicable to visualize software written in procedural and object oriented languages. However, due to their popularity, we focus on object-oriented languages. Hence, we use Java as reference language to explain the metaphor.

3.1 Glyph Design

A glyph consists of a graphical entity with components, each of which has geometric attributes and appearance attributes (Ward, 2002). For the recursive disk metaphor, we use the geometric attributes shape, size, orientation, position, and direction as well as the appearance attributes color and transparency.

3.1.1 Geometric Attributes

For each software entity, i.e., attribute, method, class, and package as well as the system as a whole circular glyphs are used. The circle for classes is divided into one or more inner circles surrounded by rings. From inside to outside, inner classes, attributes, and methods are mapped to these elements. If one of these entities is missing, it is simply omitted. Attributes and methods are represented by circle or ring segments. The outermost ring of a class forms its border to distinguish it from other classes. In Java packages have



(a) (b)

Figure 1: Basic glyphs and relations with the recursive disk metaphor: (a) 1 - Package with five classes, 2 - General classes with altogether eighteen methods and five attributes, 3 - Method class with two methods, 4 - Data class with four attributes, 5 - Class with eight methods, eight attributes, and three inner classes (b) 1 - Selected class, 2 - Superclass.

neither methods nor attributes. For this reason, they are only represented by the border ring.

Attribute glyphs are all of the same size. The size of a method glyph is estimated using its number of statements. The size of a class glyph is determined by the sum of the number of its attributes, the sizes of its methods, and, if present, the sizes of its inner classes[1]. All values are accumulated and represented by area. Consequently, a class with a large size covers a large area. This area reflects the expense to read and understand the source code of a certain class. As the radius of the rings for packages and classes depends on their elements, it is defined by the minimum bounding circle.

3.1.2 Appearance Attributes

The default color mapping is chosen according to the opponent process theory (Ware, 2004). As most people with color deficiency view have problems distinguishing red and green, the combination of these colors has been avoided. Consequently, the glyphs for attributes are yellow, methods are blue, classes purple, and packages are gray. An example of the appearance of the different glyph types is shown in Figure 1 (a).

Relations between glyphs can be explored interactively. They are visualized using opacity. Only glyphs participating in a relation are opaque while all other, unrelated glyphs are transparent. To visualize relations, a glyph has to be selected and the type of relation has to be chosen. A selected glyph is marked red. There are different types of relations depending on the type of the glyph. For class glyphs there are supertypes and subtypes, for method glyphs there are callers and callees, and for attribute glyphs there are accessors. An example of showing the supertype of a class glyph is illustrated in Figure 1 (b).

[1]The original idea (Eisenecker, 2012) uses sizes of attributes and methods that are proportional to their number of characters of their identifier or definition. However, due to technical restrictions, we use the approach described above.

3.2 Placement Strategy

The layout of the glyphs is structure-driven combining a hierarchical and an ordered circular positioning pattern (Ward, 2002). Hence, the class and package glyphs are arranged according to their hierarchy level and their net area. The net area is the actual area of a glyph derived from its containing elements. On the contrary, the gross area includes additional empty space due to hierarchical placement. The applied layout is a derivation of the classical circle packing algorithm (Wang et al., 2006). The difference is that the glyph with the largest net area is placed in the center of the visualization and the remaining glyphs are ordered descending by their net area and arranged clockwise around the largest glyph in the center. This is done recursively for all class and package glyphs on every hierarchy level. Additionally, the method glyphs in a class glyph are ordered clockwise descending according to their area. Attribute glyphs are arranged in the same manner depending on the size of their type. The result is a shape with an inner structural consistency and a completely defined orientation. The extension of the classical circle packing algorithm with the described placement strategy facilitates the comparison of areas of different glyphs. An example of the arrangement of one package, five classes, and three inner classes is shown in Figure 1 (a).

3.3 Implementation

The underlying technical approach for generating the recursive disk metaphor combines the generative and the model-driven paradigms (Müller et al., 2011). The whole visualization pipeline and the applied implementation techniques are summarized in Figure 2.

The information needed for the visualization is extracted from software systems and stored in Famix (Nierstrasz et al., 2005). During the analysis, these models are checked for syntactic and semantic

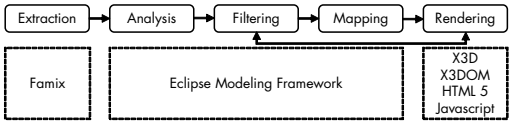


Figure 2: Visualization pipeline and implementation techniques.

validity. They must conform to their meta-model and fulfill some predefined rules, e.g., each entity must have a unique identifier.

There are two types of filtering. The first one is applied at build time. Here, the user can specify the desired packages that should be visualized. Currently, this is realized by a properties file. This will be replaced by a wizard in a future version. The second one is applied at runtime and described in Section 4.1.

The mapping is realized by model transformations and model modifications using the Eclipse Modeling Framework (EMF, 2014). It is divided into two parts. First, the valid and filtered entities from the input model are mapped to a platform independent model. Then, the layout of these entities is computed providing sizes and positions for the visualization. Second, the platform independent model is mapped to a platform specific one, here, Extensible 3D (X3D). Finally, the X3D model is optimized for the web and converted to X3DOM (Behr et al., 2012). The resulting visualization is rendered by a browser.

4 CASE STUDY: FINDBUGS

Findbugs is an open source software that uses static analysis to look for bugs in Java code (Findbugs, 2014). According to our analysis, version 3.0.0 has 61 packages, 1425 classes, 10541 methods, and 5413 attributes. Altogether, there are approximately 200K LOC.

4.1 Navigation and Interaction

As depicted in Figure 3, currently the following interaction techniques are supported to explore Findbugs:

- **Overview/Zoom:** The navigation mode *turntable* allows to zoom in and out, to rotate, and to pan.
- **Filter:** The entities can be hidden and unhidden as well as searched for.

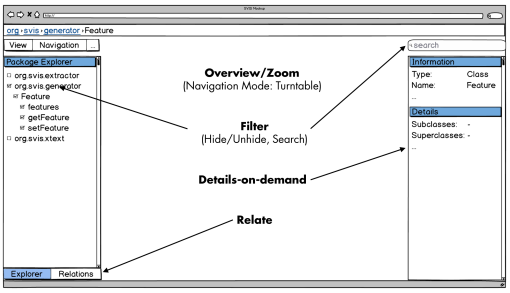


Figure 3: Mockup of the browser interface with focus on interaction techniques.

- **Details-on-demand:** For each entity exists a detailed view.
- **Relate:** Relations between entities can be shown.

From Shneiderman's visualization mantra (Shneiderman, 1992), only history and extract are currently not supported.

4.2 Visual Patterns

The glyph design and the placement strategy lead to a specific appearance of glyphs on class level and on system level forming unique visual patterns.

Hence, a visual differentiation of the kind of classes is possible based on patterns. In Findbugs, the following patterns occur. A general class with attributes and methods has a yellow circle in its center surrounded by a blue ring (Figure 4 (a)). A class with only attributes is yellow (Figure 4 (b)). If it is not a data class, it is an enumeration. A class with only methods is blue (Figure 4 (c)). A class with neither attributes nor methods results in a purple disk (Figure 4 (d)). The ring with a blue circle in its center or the purple disk may be an abstract class or an interface. Nested elements, such as inner classes or classes in packages, lead to some empty space in the resulting figure producing further recognizable visual patterns (Figure 4 (e)).

The recursive disk metaphor can be used to assess the quality of software by exploring visual pat-

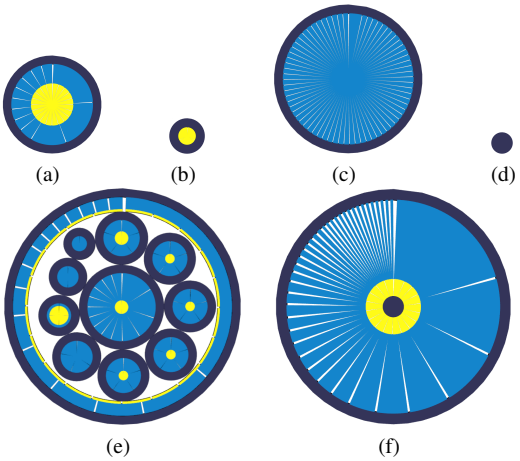
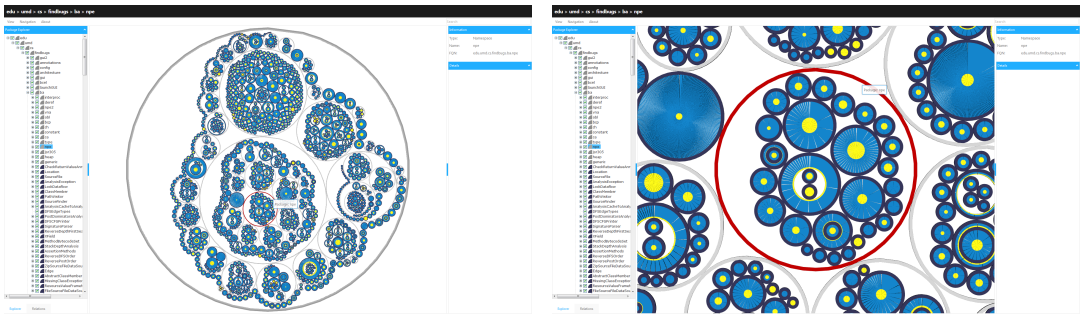


Figure 4: Examples for patterns (a-d) and anti-patterns (e-f) in Findbugs visualized with the recursive disk metaphor: (a) General class with attributes and methods (IncompatibleTypes) (b) Enumeration (IdentityMethodState) (c) Abstract class (BetterVisitor) (d) Interface (ComparableMethod) (e) God class (FindRefComparison) (f) Brain class (FindNullDeref).



(a) (b)

Figure 5: The structure of Findbugs visualized with the recursive disk metaphor in a browser: (a) Overview (b) Zoom.

terns. Design flaws can be identified by anti-patterns. Lanza et al. Lanza et al. (2006) introduced several anti-patterns, such as god class and brain class. An example for each anti-pattern occurring in Findbugs is shown in Figure 4 (e) and (f). Obviously, these two classes have a different appearance and they are bigger than the other classes. Additionally, they tend to appear in the center of their hierarchy level. Consequently, they are readily detectable. We believe that the recursive disk metaphor is ideally suited to detect anti-patterns in software systems. While these anti-patterns could in principle be detected automatically (Lanza et al., 2006), the parameters for these detection algorithms have to be established empirically. Using visualization, no parameters are needed and combinations of anti-patterns can be spotted (Wettel and Lanza, 2008).

All these glyphs form the visualization in Figure 5. It contains two screen-shots of Findbugs visualized with the recursive disk metaphor in a browser. The left screenshot shows the structure of the whole system and the right screenshot represents a detailed view of a part of the system.

5 CONCLUSION AND FUTURE WORK

We presented the recursive disk metaphor using glyph-based visualization for software visualization. The metaphor focuses on the structure of software including all important entities from package to attribute level as well as their relations. Additionally, it has an inner structural consistency and a completely defined orientation. Hence, the glyph-based approach gives the per se intangible and invisible software a shape. It produces unique visual patterns for class structures and for anti-patterns. We compared the recursive disk metaphor to related work and discussed design decisions. Further, we outlined implementation details and presented the interface. Its feasibility and scalability has been shown with a case study.

In the future, we intend to cover additional languages, such as C/C++ and .NET. Additionally, we plan to compare our approach with established approaches for visually detecting anti-patterns (Wettel and Lanza, 2008). Finally, a series of controlled experiments is planned based on the approach by Müller et al. (2014) to empirically evaluate the metaphor.

ACKNOWLEDGEMENTS

We would like to thank Ulrich Eisenecker for the initial idea of this metaphor (Eisenecker, 2012) and the inspiring discussions.

REFERENCES

Andrews, K. and Heidegger, H. (1998). Information slices: Visualising and exploring large hierarchies using cascading, semi-circular discs. In *InfoVis 1998*, pages 9–11.

Balzer, M., Deussen, O., and Lewerentz, C. (2005). Voronoi treemaps for the visualization of software metrics. In *Proc. 2005 ACM Symp. Softw. Vis.*, pages 165–172, New York, USA. ACM Press.

Barlow, T. and Neville, P. (2001). A comparison of 2-D visualizations of hierarchies. In *InfoVis 2001*, pages 131–138. IEEE.

Behr, J., Jung, Y., Franke, T., and Sturm, T. (2012). Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proc. 17th Int. Conf. 3D Web Technol.*, pages 17–26, New York, USA. ACM Press.

Boccuzzo, S. and Gall, H. (2007). CocoViz: Towards Cognitive Software Visualizations. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 72–79. IEEE.

Borgo, R., Kehrer, J., Chung, D., Maguire, E., Laramee, R. S., Ward, M., and Chen, M. (2013). Glyph-based visualization: Foundations, design guidelines, techniques and applications. *Eurographics*.

Burch, M. and Weiskopf, D. (2014). On the Benefits and Drawbacks of Radial Diagrams. In *Handb. Hum. Centric Vis.*, pages 429–451. Springer.

Byelas, H. and Telea, A. (2006). Visualization of areas of interest in software architecture diagrams. In *Proc. 2006 ACM Symp. Softw. Vis.*, pages 105–114, New York, USA. ACM Press.

Caserta, P. and Zendra, O. (2011). Visualization of the Static Aspects of Software: A Survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933.

Chuah, M. and Eick, S. (1998). Information rich glyphs for software management data. *IEEE Comput. Graph. Appl.*, 18(4):24–29.

Cleveland, W. and McGill, R. (1984). Graphical perception: Theory, experimentation, and application to the development of graphical methods. *J. Am. Stat. Assoc.*, 79(387):531–554.

Ducasse, S. and Lanza, M. (2005). The class blueprint: visually supporting the understanding of classes. *IEEE Trans. Softw. Eng.*, 31(1):75–90.

Eisenecker, U. W. (2012). Ideas on the recursive disk metaphor (audio file).

EMF (2014). Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`. Accessed: 2014-11-05.

Findbugs (2014). Findbugs. `http://findbugs.sourceforge.net/`. Accessed: 2014-11-05.

Fischer, F., Fuchs, J., and Mansmann, F. (2012). ClockMap: Enhancing circular treemaps with temporal glyphs for time-series data. In *Eurographics Conf. Vis.*, pages 97–101. ACM.

Gračanin, D., Matković, K., and Eltoweissy, M. (2005). Software Visualization. *Innov. Syst. Softw. Eng.*, 1(2):221–230.

Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., and Mutzel, P. (2003). A new approach for visualizing UML class diagrams. In *Proc. 2003 ACM Symp. Softw. Vis.*, pages 179–188, New York, USA. ACM Press.

Holten, D. (2006). Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):741–8.

Holten, D., Vliegen, R., and van Wijk, J. (2005). Visual Realism for the Visualization of Software Metrics. In *3rd Int. Work. Vis. Softw. Underst. Anal.*, pages 27–32. IEEE.

Lanza, M. (2003). CodeCrawler - A Lightweight Software Visualization Tool. In *2nd Int. Work. Vis. Softw. Underst. Anal.*, pages 54–55.

Lanza, M., Marinescu, R., and Ducasse, S. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer-Verlag Berlin Heidelberg.

McGuffin, M. J. and Robert, J.-M. (2010). Quantifying the space-efficiency of 2D graphical representations of trees. *Inf. Vis.*, 9(2):115–140.

Müller, R., Kovacs, P., Schilbach, J., and Eisenecker, U. (2011). Generative Software Visualization: Automatic Generation of User-Specific Visualizations. In *Proc. Int. Work. Digit. Eng.*, pages 45–49, Magdeburg, Germany.

Müller, R., Kovacs, P., Schilbach, J., Eisenecker, U., Zeckzer, D., and Scheuermann, G. (2014). A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization. In *Proc. 5th Int. Conf. Vis. Theory Appl.*, pages 204–209, Lisbon, Portugal.

Nierstrasz, O., Ducasse, S., and Gîrba, T. (2005). The story of moose: an agile reengineering environment. In *Proc. 10th Eur. Softw. Eng. Conf. held jointly with 13th SIGSOFT Int. Symp. Found. Softw. Eng.*, volume 30 of *ESEC/FSE-13*, pages 1–10, New York, USA. ACM.

Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005). Visualizing multiple evolution metrics. In *Proc. 2005 ACM Symp. Softw. Vis.*, pages 67–75, New York, USA. ACM Press.

Sangal, N., Jordan, E., Sinha, V., and Jackson, D. (2005). Using dependency models to manage complex software architecture. In *Proc. 20th Annu. ACM SIGPLAN Conf. Object oriented Program. Syst. Lang. Appl.*, New York, USA. ACM Press.

Shneiderman, B. (1992). Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99.

Stasko, J., Catrambone, R., Guzdial, M., and McDonald, K. (2000). An evaluation of space-filling information visualizations for depicting hierarchical structures. *Int. J. Hum. Comput. Stud.*, 53(5):663–694.

Storey, M., Wong, K., and Müller, H. (1997). Rigi: a visualization environment for reverse engineering. In *19th ACM Int. Conf. Softw. Eng.*, pages 606–607.

Termeer, M., Lange, C., Telea, A., and Chaudron, M. (2005). Visual Exploration of Combined Architectural and Metric Information. In *3rd Int. Work. Vis. Softw. Underst. Anal.*, pages 21–26, Washington, DC, USA. IEEE.

Wang, W., Wang, H., Dai, G., and Wang, H. (2006). Visualization of large hierarchical data by circle packing. In *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, pages 517–520, New York, USA. ACM Press.

Ward, M. (2002). A taxonomy of glyph placement strategies for multidimensional data visualization. *Inf. Vis.*, 1:194–210.

Ward, M. O. (2008). Multivariate Data Glyphs: Principles and Practice. In *Handb. Data Vis.*, pages 179–198. Springer.

Ware, C. (2004). *Information visualization: perception for design*. Morgan Kaufmann, 2nd edition.

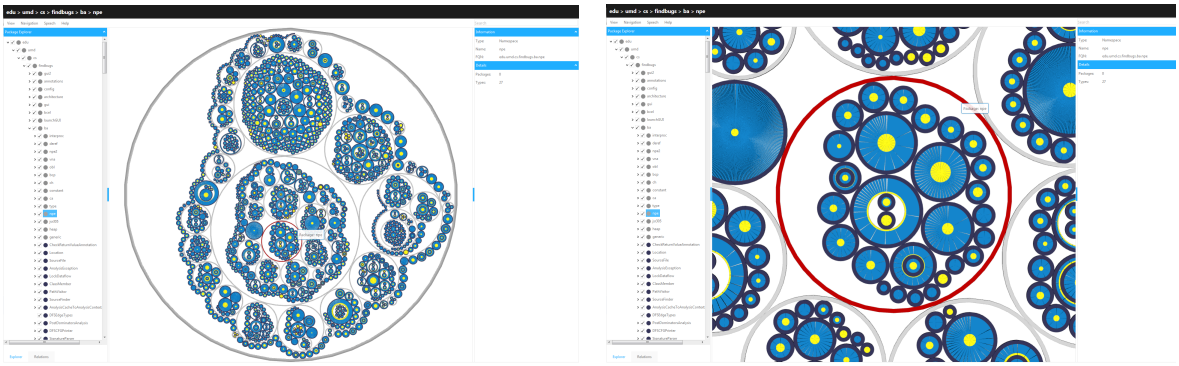
Wettel, R. and Lanza, M. (2008). Visually localizing design problems with disharmony maps. In *Proc. 4th ACM Symp. Softw. Vis.*, pages 155–164, New York, USA. ACM Press.

7.2 Summary

The recursive disk metaphor is a glyph-based visualization for software visualization. It represents all important structural aspects and relations of software using nested circular glyphs. The metaphor is implemented with the generative and model-driven approach and generated automatically.

The metaphor refers to the top level ***RQ:*** *How should a software visualization be designed, to visualize structural, behavioral, and evolutionary aspects of a software system, and how can it be generated automatically?* (see Section 1.2). The design of the metaphor implies the findings from the literature study in Chapter 3, is implemented with the software visualization generator presented in Chapter 4, and considers the results of the controlled experiment described in Chapter 6.

As stated in Section 6.2, the third dimension plays a rather tangible role. For this reason, the basic shape of the recursive disk metaphor is 2D in the first place. The metaphor can be applied to visualize *structural* aspects, such as packages, classes, inner classes, methods, and attributes, as well as their relations. The resulting shape has an inner structural consistency and a completely defined orientation. The glyph design and the placement strategy lead to visually detectable patterns and anti-patterns [Ward 2008]. Listing B.1 shows the Xtext grammar of the recursive disk metaphor. Figure 7.1 contains two screen-shots of Findbugs [2014] visualized with the recursive disk metaphor in a browser. In order to evaluate the scalability of the approach, Vuze [2014] (formerly Azureus), a large software system, has been successfully visualized with the recursive disk metaphor.



(a) (b)

Figure 7.1: The structure of Findbugs visualized with the recursive disk metaphor in a browser [Müller and Zeckzer 2015b]: (a) overview (b) zoom.

However, taking the application categories of the third dimension into account, summarized in Section 3.2, the recursive disk metaphor may be extended with *behavioral* and *evolutionary* views. Therefore, the third dimension can be used with a combination of the categories *extended 2D* and *3D as time*. The structural basis of the recursive metaphor opens a broad spectrum of many different variants for designing behavioral and evolutionary views. Next, one straight forward possibility for a behavioral view is described that is similar to Code-

Crawler [Greevy et al. 2005]. This view builds upon the 2D structural view and extends it with information about class instances and method invocations at runtime. It provides the user an overview of instantiated classes and invoked methods for a certain execution trace. For each instance of a class an instance ring is placed above the structural base. If there are several instances of the same class they are placed above each other with some space left in between. The instance ring holds the invoked method segments. If the same method is invoked multiple times within one instance, the method segments are stacked on top of each other. Thus, depending on the number of instances and method invocations the class grows in the third dimension. The resulting behavioral view is depicted in Fig. 7.2.

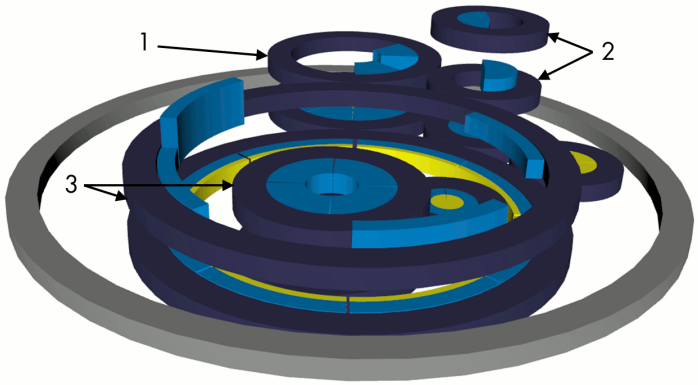


Figure 7.2: Behavior with the recursive disk metaphor: 1 - One instance with two method invocations, 2 - Two instances with two invocations of the same method for the first instance and one method invocation for the second instance, 3 - One instance of an inner class with four method invocations and one instance of the surrounding class with a total of seven method invocations.

The same applies for evolutionary views where the 2D structural view of the recursive disk metaphor can be extended to the third dimension. One possibility is to use the height of a glyph to represent the number of its modifications similar to Evo-Streets [Steinbrückner and Lewerentz 2010].

To sum it up, the recursive disk metaphor is able to visualize all important structural aspects and their relations of a software system in 2D, it is generated automatically, and it scales well for large software systems. Furthermore, the metaphor may be extended to 3D in order to visualize behavioral and evolutionary aspects. Therefore, the categories *extended 2D* and *3D as time* are combined. The extensions of the recursive disk metaphor with behavioral and evolutionary views are currently under development. After their completion, the resulting software visualizations provide holistic views on all three aspects of the software system.

8 Conclusion and Future Work

The final chapter summarizes the contributions of this thesis, concludes the recommendations for 3D software visualizations, and provides an outlook to future work.

8.1 Contributions

There are five main contributions of this thesis (see Section 1.4).

1. Literature study presenting an overview of state-of-the-art in 3D software visualization [Müller and Zeckzer 2015a] (see Chapter 3).
   The study presents the state-of-the-art including trends and research gaps in 3D software visualization. It combines a systematic mapping study [Petersen et al. 2008] and a literature review [vom Brocke et al. 2009]. Especially the venues, the visualized aspects, the evolution, the evaluation methods, and the applications of the third dimension in the field of 3D software visualization are analyzed. This analysis revealed two research gaps. First, there is a lack of empirical evaluations in the field of 3D software visualization. Second, there are only few 3D software visualizations that provide multiple views of a software system including all three aspects, i.e., structure, behavior, and evolution. The first research gap is addressed by the structured approach (3.) and the controlled experiment (4.). The second research gap is addressed by the software visualization generator (2.) and the recursive disk metaphor (5.). Additionally, the analysis led to an update of the application categories introduced by Reiss [1995] and extended it with *3D for cognition*. Hence, there are six categories[6] for the application of the third dimension: *extended 2D*, *full 3D*, *2D layout organized in 3D*, *3D as time*, *stacked views*, and *3D for cognition*.
2. Eclipse-based generator for generating 2D, 2.5D, and 3D software visualizations automatically [Müller et al. 2011] (see Chapter 4).
   The generator combines the generative [Czarnecki and Eisenecker 2000] and the model-driven [Stahl et al. 2006] paradigms to produce automatically role- and task-specific visualizations according to user requirements specified in a DSL. The generated visualizations may represent structural, behavioral, and/or evolutionary aspects of a software system in 2D, 2.5D, or 3D. Furthermore, the visualizations are X3D models. These models can be optimized for the web using AOPT and rendered in any browser supporting X3DOM. This assures the platform independence of the software visualizations. The plug-in architecture of the generator is implemented with Eclipse tech-

[6] The original category *local fish-eye* was not found in the sample and for this reason omitted.

niques including JDT, PDE, TMF, and EMP. This assures its easy extensibility and its seamless integration into the Eclipse IDE.

3. Structured approach for conducting controlled experiments in software visualization [Müller et al. 2014a] (see Chapter 5).

   The approach supports researchers in planning and in designing a series of experiments in software visualization and to control influence factors. The approach is based on the extended process model for design and validation of visualizations by Munzner et al. [Munzner 2009; Meyer et al. 2012]. This model is adapted to the field of software visualization and enhanced with influence factors derived from several software visualization taxonomies.

4. Controlled experiment investigating the role of the third dimension in software visualization [Müller et al. 2014b] (see Chapter 6).

   The experiment was planned and designed with the structured approach (3.) and used visualizations from the generator (2.). The dependent variables in the experiment were *time*, *correctness*, and *interaction*. The independent variable *dimension* was varied (2D and 3D). The random sample was divided into a control group (2D) and an experimental group (3D). To measure the presumed effect, a between-subjects design was applied. Although the experimental group made less mistakes and gave a more positive feedback, the control group solved their software engineering tasks faster. This means that the third dimension does not show significant advantages in solving a task for medium-sized software systems. It plays a rather tangential role. To increase its benefit, it has to applied in a useful way. More important seems to be the applied visualization technique or metaphor.

5. Recursive disk metaphor combining the findings with focus on the structure of software with useful applications of the third dimension [Müller and Zeckzer 2015b] (see Chapter 7).

   The recursive disk metaphor is a glyph-based visualization for software visualization [Borgo et al. 2013]. It represents all important structural aspects and relations of software using nested circular glyphs. The metaphor is implemented with the generative and model-driven approach and uses 2D to visualize the *structural* aspects. However, this 2D shape can be easily extended to 3D in order to additionally visualize *behavioral* or *evolutionary* aspects. Therefore, the application categories *extended 2D* and *3D as time* are combined. These extensions are currently under development. After their completion, the resulting software visualizations provide holistic views on all three aspects of a software system. Besides their automatic generation, the visualizations scale well for large software systems and they are platform independent.

8.2 Recommendations for 3D Software Visualizations

For the moment, it has to be stated that the third dimension does not show significant advantages in solving a task for medium-sized software systems. Hence, the factor *dimensionality* is not decisive and plays a rather tangential role. This finding brings the applied metaphor more into focus. Thus, it is more important to use a suitable metaphor independent from its dimensionality. For the design of the metaphor the glyph-based visualization serves as a sound basis [Borgo et al. 2013]. However, the optimal interplay between 2D and 3D may be the clue to the successful integration of all three aspects. In this context, the application of the third dimension should be thoroughly thought out. For this reason, all application categories are summarized and shown in Figure 8.1. The different categories are not disjoint and can be combined. This list serves as a starting point and does not claim to be complete. Rather, it is designed as an open list to be extended by other researchers.

- *Extended 2D:* A 2D layout is extended to 3D. The additional dimension can be used to display further information, such as software metrics, LOC, complexity, or the number of modifications, relations, or instances (a).
- *Full 3D:* The next technique moves from 2D to 3D and uses the full capabilities of three dimensions (b).
- *2D layout organized in 3D:* The second technique takes a 2D layout and organizes the information in a 3D space. It is usually applied to get more space and to minimize edge-crossings (c).
- *3D as time:* Further, the third dimension is used to represent time (d).
- *Stacked views:* This technique uses the third dimension to display several 2D views simultaneously (e).
- *3D for cognition:* Finally, 3D shapes are applied to support the mental model and to optimize the cognition of the visualization (f).

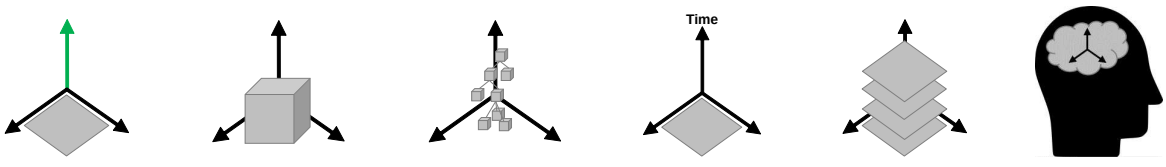


Figure 8.1: Applications of the third dimension in software visualization [Müller and Zeckzer 2015a]: (a) extended 2D (b) full 3D (c) 2D layout organized in 3D (d) 3D as time (e) stacked views (f) 3D for cognition.

8.3 Outlook

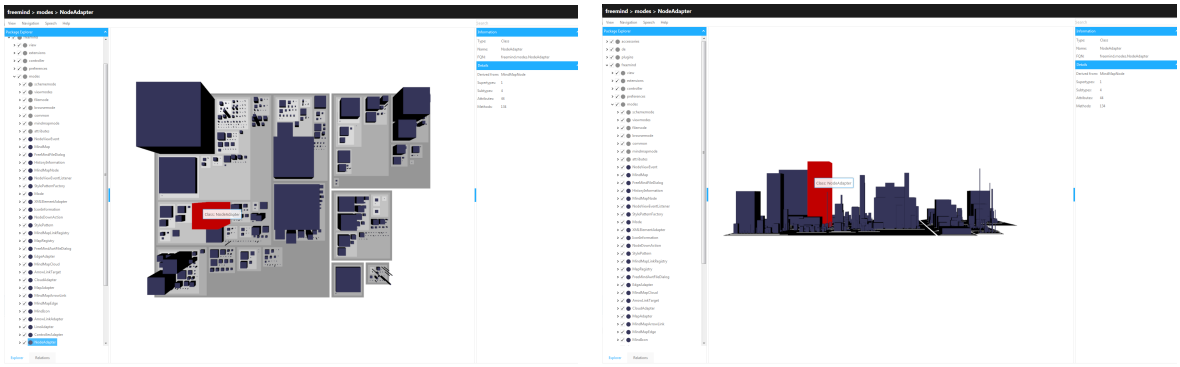
The end of one work is often the beginning of another one. So it is in this case. The final section outlines various theoretical and practical future work.

8.3.1 Literature Study

The literature study will be extended to 2D software visualization. The integration and the analysis of the results of both studies will provide new and promising insights concerning state-of-the-art, trends, and research gaps in the field of software visualization.

8.3.2 Generator

The architecture of the Eclipse-based software visualization generator is modular and easily extensible. Thus, new visualization techniques and metaphors can be implemented with considerably less effort. For example, the city metaphor similar to Wettel and Lanza [2007] has already been implemented with the generative and model-driven approach. A screenshot of Freemind [2014] visualized with the city metaphor in a browser is depicted in Figure 8.2. Various additional software visualizations are planned to be implemented with this approach to benefit from scalability, IDE integration, and platform independence.



(a) (b)

Figure 8.2: The structure of Freemind visualized with the city metaphor in a browser: (a) top view (b) front view.

The next steps in the context of the generator are the migration from Eclipse version 4.3 (Kepler) to version 4.4 (Luna) and the integration into the Eclipse IDE on the basis of RCP. Currently, the DSL to specify user requirements is a properties file. With the Eclipse integration this textual DSL will be replaced by a wizard-based one. Furthermore, it is planned to link the visualization entities with the corresponding source code artifacts.
At the moment, the generator is used for software visualization. However, the basic concept of transforming formal models into visualizations according to a DSL can be easily extended to information visualization in general and further areas of application. This includes for example the emerging fields Business Intelligence (BI) and Big Data (BD).

8.3.3 X3DOM

As tablets and smartphones become more and more important as visualization devices the choice of X3DOM holds potential. For example, Limberger et al. [2013] present a web-based approach that makes software maps more accessible to many different stakeholders

in software engineering projects. As the preferred output of the generator is X3D that can be easily transformed into X3DOM, a similar web-based and collaborative solution with the approach described in this thesis is possible.
Furthermore, the X3DOM visualizations are suitable for controlled experiments as they can be easily integrated in a web-based evaluation environment. In such an environment it is comparatively easy to guide the participants through different tasks and to track relevant information, e.g., time or responses. A first prototype of the web-based evaluation environment is under development.

8.3.4 Recursive Disk Metaphor

The recursive disk metaphor still holds potential for future work. At present, it is possible to visualize software systems implemented in Java. In the future, it is planned to cover additional languages, such as C/C++ and .NET. Additionally, the recursive disk metaphor will be evaluated empirically and examined for its visual software quality assessment capabilities in the context of a proposed research project.

8.3.5 Research Project

The main contributions of this thesis serve as a basis for a proposal of a research project. The main goal of this project is to empirically investigate the advantages and disadvantages of 2D and 3D software visualizations with focus on different metaphors and interaction techniques.

Appendix

A Famix

Listing A.1 shows the MWE2 configuration of the language generator for Famix.

```
module org.svis.xtext.GenerateFamix

import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*

var grammarURI = "classpath:/org/svis/xtext/Famix.xtext"
var fileExtensions = "famix"
var projectName = "org.svis.xtext.famix"
var runtimeProject = "../${projectName}"
var generateXtendStub = true

Workflow {
   bean = StandaloneSetup {
     scanClassPath = true
     platformUri = "${runtimeProject}/.."
   }
   component = DirectoryCleaner {
     directory = "${runtimeProject}/src-gen"
   }
   component = DirectoryCleaner {
     directory = "${runtimeProject}.ui/src-gen"
   }
   component = Generator {
     pathRtProject = runtimeProject
     pathUiProject = "${runtimeProject}.ui"
     pathTestProject = "${runtimeProject}.tests"
     projectNameRt = projectName
     projectNameUi = "${projectName}.ui"
     language = auto-inject {
       uri = grammarURI
       // Java API to access grammar elements (required by several other fragments)
       fragment = grammarAccess.GrammarAccessFragment auto-inject {}
       // generates Java API for the generated EPackages
       fragment = ecore.EMFGeneratorFragment auto-inject {}
       // the old serialization component
       // fragment = parseTreeConstructor.ParseTreeConstructorFragment auto-inject {}
       // serializer 2.0
       fragment = serializer.SerializerFragment auto-inject {
         generateStub = false
       }
       // a custom ResourceFactory for use with EMF
       fragment = resourceFactory.ResourceFactoryFragment auto-inject {}
       // The antlr parser generator fragment.
       fragment = parser.antlr.XtextAntlrGeneratorFragment auto-inject {}
       // Xtend-based API for validation
       fragment = validation.ValidatorFragment auto-inject {
```

```
48        //   composedCheck = "org.eclipse.xtext.validation.ImportUriValidator"
49        //   composedCheck = "org.eclipse.xtext.validation.NamesAreUniqueValidator"
50        }
51        // old scoping and exporting API
52        // fragment = scoping.ImportURIScopingFragment auto-inject {}
53        // fragment = exporting.SimpleNamesFragment auto-inject {}
54        // scoping and exporting API
55        fragment = scoping.ImportNamespacesScopingFragment auto-inject {}
56        fragment = exporting.QualifiedNamesFragment auto-inject {}
57        fragment = builder.BuilderIntegrationFragment auto-inject {}
58        // generator API
59        fragment = generator.GeneratorFragment auto-inject {}
60        // formatter API
61        fragment = formatting.FormatterFragment auto-inject {}
62        // labeling API
63        fragment = labeling.LabelProviderFragment auto-inject {}
64        // outline API
65        fragment = outline.OutlineTreeProviderFragment auto-inject {}
66        fragment = outline.QuickOutlineFragment auto-inject {}
67        // quickfix API
68        fragment = quickfix.QuickfixProviderFragment auto-inject {}
69        // content assist API
70        fragment = contentAssist.ContentAssistFragment auto-inject {}
71        // generates a more lightweight Antlr parser and lexer tailored for content assist
72        fragment = parser.antlr.XtextAntlrUiGeneratorFragment auto-inject {}
73        // generates junit test support classes into Generator#pathTestProject
74        fragment = junit.Junit4Fragment auto-inject {}
75        // project wizard (optional)
76        // fragment = projectWizard.SimpleProjectWizardFragment auto-inject {
77        //    generatorProjectName = "${projectName}"
78        // }
79        // rename refactoring
80        fragment = refactoring.RefactorElementNameFragment auto-inject {}
81        // provides the necessary bindings for java types integration
82        fragment = types.TypesGeneratorFragment auto-inject {}
83        // generates the required bindings only if the grammar inherits from Xbase
84        fragment = xbase.XbaseGeneratorFragment auto-inject {}
85        // provides a preference page for template proposals
86        fragment = templates.CodetemplatesGeneratorFragment auto-inject {}
87        // provides a compare view
88        fragment = compare.CompareFragment auto-inject {}
89      }
90    }
91  }
```

Listing A.1: Language generator for Famix.

Listing A.2 shows the Xtext grammar definition of Famix.

```
1  grammar org.svis.xtext.Famix with org.eclipse.xtext.common.Terminals
2  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
3  generate famix "http://www.svis.org/famix"
4
5  Root:
6   document=Document?;
7
8  Document:
9   {Document}
10  '(' elements+=FAMIXElement* ')';
11
```

```
FAMIXElement:
  FAMIXClass | FAMIXFileAnchor | FAMIXInvocation | FAMIXParameterizableClass | FAMIXAttribute |
        FAMIXInheritance | FAMIXAccess | FAMIXNamespace | FAMIXMethod |FAMIXPrimitiveType | FAMIXComment |
        FAMIXParameter | FAMIXReference | FAMIXParameterizedType | FAMIXAnnotationInstance |
        FAMIXAnnotationInstanceAttribute | FAMIXAnnotationType | FAMIXAnnotationTypeAttribute |
        FAMIXLocalVariable | FAMIXImplicitVariable | FAMIXType | FAMIXParameterType | FAMIXJavaSourceLanguage|
        FAMIXDeclaredException | FAMIXThrownException | FAMIXCaughtException | FAMIXEnum | FAMIXEnumValue;

FAMIXNamespace:
  '(FAMIX.Namespace'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'parentScope' parentScope=IntegerReference ')')?
  ')';

FAMIXFileAnchor:
  '(FAMIX.FileAnchor'
  '(' 'id: ' name=INT_ID ')'
  ('(' 'element' element=IntegerReference ')')?
  '(' 'endLine' endline=INT ')'
  '(' 'fileName' filename=MSESTRING ')'
  '(' 'startLine' startline=INT ')'
  ')';

FAMIXClass:
  '(FAMIX.Class'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'container' container=IntegerReference ')'
  ('(' 'isInterface' isInterface=Boolean ')')?
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  ('(' 'sourceAnchor' type=IntegerReference ')')?
  ')';

FAMIXParameterizableClass:
  '(FAMIX.ParameterizableClass'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'container' container=IntegerReference ')'
  ('(' 'isInterface' isInterface=Boolean ')')?
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  ('(' 'sourceAnchor' type=IntegerReference ')')?
  ')';

FAMIXMethod:
  '(FAMIX.Method'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  ('(' 'cyclomaticComplexity' cyclomaticComplexity=INT ')')?
  ('(' 'declaredType' declaredType=IntegerReference ')')?
  ('(' 'hasClassScope' hasClassScope=Boolean ')')?
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'kind' kind=MSESTRING ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  ('(' 'numberOfStatements' numberOfStatements=INT ')')?
  '(' 'parentType' parentType=IntegerReference ')'
  '(' 'signature' signature=MSESTRING ')'
```

```
  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
  ')';

FAMIXInvocation:
  '(FAMIX.Invocation'
  '(' 'id: ' name=INT_ID ')'
  '(' 'candidates' candidates=IntegerReference ')'
  ('(' 'previous' previous=IntegerReference ')')?
  ('(' 'receiver' receiver=IntegerReference ')')?
  '(' 'sender' sender=IntegerReference ')'
  '(' 'signature' signature=MSESTRING ')'
  ')';

FAMIXAttribute:
  '(FAMIX.Attribute'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'declaredType' declaredType=IntegerReference ')'
  ('(' 'hasClassScope' hasClassScope=Boolean ')')?
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  '(' 'parentType' parentType=IntegerReference ')'
  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
  ')';

FAMIXAccess:
  '(FAMIX.Access'
  '(' 'id: ' name=INT_ID ')'
  '(' 'accessor' accessor=IntegerReference ')'
  ('(' 'isWrite' isWrite=Boolean ')')?
  ('(' 'previous' previous=IntegerReference ')')?
  '(' 'variable' variable=IntegerReference ')'
  ')';


FAMIXPrimitiveType:
  '(FAMIX.PrimitiveType'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'isStub' isStub=Boolean ')'
  ')';

FAMIXComment:
  '(FAMIX.Comment'
  '(' 'id: ' name=INT_ID ')'
  '(' 'container' container=IntegerReference ')'
  '(' 'content' content=MSESTRING ')'
  '(' 'sourceAnchor' sourceAnchor=IntegerReference ')'
  ')';

FAMIXParameter:
  '(FAMIX.Parameter'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'declaredType' declaredType=IntegerReference ')'
  '(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')'
  ')';

FAMIXInheritance:
  '(FAMIX.Inheritance'
```

```
  '(' 'id: ' name=INT_ID ')'
  ('(' 'previous' previous=IntegerReference ')')?
  '(' 'subclass' subclass=IntegerReference ')'
  '(' 'superclass' superclass=IntegerReference ')'
  ')';

FAMIXReference:
  '(FAMIX.Reference'
  '(' 'id: ' name=INT_ID ')'
  '(' 'source' source=IntegerReference ')'
  '(' 'target' target=IntegerReference ')'
  ')';

FAMIXParameterizedType:
  '(FAMIX.ParameterizedType'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  ('(' 'arguments' arguments+=IntegerReference* ')')?
  '(' 'container' container=IntegerReference ')'
  ('(' 'isStub' isStub=Boolean ')')?
  '(' 'parameterizableClass' parameterizableClass=IntegerReference ')'
  ')';

FAMIXAnnotationInstance:
  '(FAMIX.AnnotationInstance'
  '(' 'id: ' name=INT_ID ')'
  '(' 'annotatedEntity' annotatedEntity=IntegerReference ')'
  '(' 'annotationType' annotationType=IntegerReference ')'
  ')';

FAMIXAnnotationInstanceAttribute:
  '(FAMIX.AnnotationInstanceAttribute'
  '(' 'id: ' name=INT_ID ')'
  ('(' 'annotationTypeAttribute' annotationTypeAttribute=IntegerReference ')')?
  '(' 'parentAnnotationInstance' parentAnnotationInstance=IntegerReference ')'
  '(' 'value' value=MSESTRING ')'
  ')';

FAMIXAnnotationType:
  '(FAMIX.AnnotationType'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'container' container=IntegerReference ')'
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
  ')';

FAMIXAnnotationTypeAttribute:
  '(FAMIX.AnnotationTypeAttribute'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  ('(' 'isStub' isStub=Boolean ')')?
  ('(' 'modifiers' modifiers+=MSESTRING* ')')?
  '(' 'parentType' parentType=IntegerReference ')'
  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
  ')';

FAMIXLocalVariable:
  '(FAMIX.LocalVariable'
```

```
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'declaredType' declaredType=IntegerReference ')'
  ('(' 'isStub' isStub=Boolean ')')?
  '(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')'
  ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
  ')';

FAMIXImplicitVariable:
  '(FAMIX.ImplicitVariable'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  ('(' 'parentBehaviouralEntity' parentBehaviouralEntity=IntegerReference ')')?
  ')';

FAMIXType:
  '(FAMIX.Type'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'container' container=IntegerReference ')'
  '(' 'isStub' isStub=Boolean ')'
  ')';

FAMIXParameterType:
  '(FAMIX.ParameterType'
  '(' 'id: ' name=INT_ID ')'
  '(' 'name' value=MSESTRING ')'
  '(' 'container' container=IntegerReference ')'
  ('(' 'isStub' isStub=Boolean ')')?
  ')';

FAMIXJavaSourceLanguage:
  '(FAMIX.JavaSourceLanguage'
  '(' 'id: ' name=INT_ID ')'
  ')';

FAMIXDeclaredException:
  '(FAMIX.DeclaredException'
  '(' 'id: ' name=INT_ID ')'
  '(' 'definingMethod' definingMethod=IntegerReference ')'
  '(' 'exceptionClass' exceptionClass=IntegerReference ')'
  ')';

FAMIXThrownException:
  '(FAMIX.ThrownException'
  '(' 'id: ' name=INT_ID ')'
  '(' 'definingMethod' definingMethod=IntegerReference ')'
  '(' 'exceptionClass' exceptionClass=IntegerReference ')'
  ')';

FAMIXCaughtException:
  '(FAMIX.CaughtException'
  '(' 'id: ' name=INT_ID ')'
  '(' 'definingMethod' definingMethod=IntegerReference ')'
  '(' 'exceptionClass' exceptionClass=IntegerReference ')'
  ')';

FAMIXEnum:
  '(FAMIX.Enum'
  '(' 'id: ' name=INT_ID ')'
```

```
247   '(' 'name' value=MSESTRING ')'
248   '(' 'container' container=IntegerReference ')'
249   ('(' 'isStub' isStub=Boolean ')')?
250   ('(' 'modifiers' modifiers+=MSESTRING* ')')?
251   ('(' 'sourceAnchor' sourceAnchor=IntegerReference ')')?
252   ')';
253
254 FAMIXEnumValue:
255   '(FAMIX.EnumValue'
256   '(' 'id: ' name=INT_ID ')'
257   '(' 'name' value=MSESTRING ')'
258   ('(' 'isStub' isStub=Boolean ')')?
259   '(' 'parentEnum' parentEnum=IntegerReference ')'
260   ')';
261
262 Boolean:
263   'true' | 'false';
264
265 IntegerReference:
266   '(' 'ref: ' ref=[FAMIXElement|INT_ID] ')';
267
268 INT_ID returns ecore::EString:
269   '^'?INT;
270
271 terminal MSESTRING:
272   '\'' ('a'..'z' | 'A'..'Z' | '.' | '_' | '\\' | '/' | '0'..'9' | '<'| '?' | '$'| '{'| '!') ('a'..'z' | 'A'..'
      Z' | '.' | '_' | '\\' | '/' | '0'..'9' | '(' | ')' | '[' | ']' | ',' | ';' | '*' | WS | '>' | '<' | '@'
       | ':' | '?' | '&' | '!' | '{' | '}'| '\"' | '-' | '+' | '=' | '$' | '#' | '~' | '^' | '%' | '|' |'\'\'
      ' | ''')* '\'';
```

Listing A.2: Xtext grammar of Famix.

B Recursive Disk Metaphor

The MWE2 configuration of the language generator for the recursive disk metaphor is similar to Listing A.1. Listing B.1 shows the Xtext grammar definition of the recursive disk metaphor for visualizing the structure of a software system.

```
1  grammar org.svis.xtext.RD with org.eclipse.xtext.common.Terminals
2  generate rd "http://www.svis.org/xtext/RD"
3  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4
5  Root:
6    document=Document?;
7
8  Document:{Document}
9    '(' disks+=Disk* ')';
10
11 Disk:
12   '(Disk'
13   '(' 'id: ' name=INT_ID ')'
14   '(' 'name' value=MSESTRING ')'
15   '(' 'fqn' fqn=MSESTRING ')'
16   '(' 'height' height=Decimal ')'
17   '(' 'radius' radius=Decimal ')'
18   '(' 'type' type=MSESTRING ')'
19   '(' 'level' level=INT ')'
```

```
20    ('(' 'loc' loc=INT ')')?
21    ('(' 'crossSection' crossSection=MSESTRING ')')?
22    ('(' 'spine' spine=MSESTRING ')')?
23    ('(' 'netArea' netArea=Decimal ')')?
24    ('(' 'grossArea' grossArea=Decimal ')')?
25    ('(' 'methodArea' methodArea=Decimal ')')?
26    ('(' 'dataArea' dataArea=Decimal ')')?
27    ('(' 'maxLevel' maxLevel=INT ')')?
28    ('(' 'color' color=MSESTRING ')')?
29    ('(' 'transparency' transparency=Decimal ')')?
30    ('(' 'position' position=Position ')')?
31    ('(' 'data' data+=DiskSegment* ')')?
32    ('(' 'methods' methods+=DiskSegment* ')')?
33    ('(' 'references' references+=Reference* ')')?
34    ('(' 'disks' disks+=Disk* ')')?
35   ')';
36
37  DiskSegment:
38   '(Disk.Segment'
39   '(' 'id: ' name=INT_ID ')'
40   '(' 'name' value=MSESTRING ')'
41   '(' 'fqn' fqn=MSESTRING ')'
42   '(' 'type' type=MSESTRING ')'
43    ('(' 'crossSection' crossSection=MSESTRING ')')?
44    ('(' 'spine' spine=MSESTRING ')')?
45    ('(' 'size' size=Decimal ')')?
46    ('(' 'radius' radius=Decimal ')')?
47    ('(' 'innerRadius' innerRadius=Decimal ')')?
48    ('(' 'outerRadius' outerRadius=Decimal ')')?
49    ('(' 'color' color=MSESTRING ')')?
50   ')';
51
52  Position:
53   '\''x=Decimal' 'y=Decimal' 'z=Decimal'\'';
54
55  Reference:
56   '(' 'ref: ' name=INT_ID ')'
57   '(' 'fqn: ' fqn=MSESTRING ')'
58   '(' 'type' type=MSESTRING ')';
59
60  INT_ID returns ecore::EString:
61   '^'?INT;
62
63  Decimal returns ecore::EDouble:
64   '-'?INT '.' INT;
65
66  terminal MSESTRING:
67   '\'' ('a'..'z' | 'A'..'Z' | '.' | '_' | '\\' | '/' | '0'..'9' | '<'| '?' | '$'| '{'| '!') ('a'..'z' | 'A'..'
       Z' | '.' | '_' | '\\' | '/' | '0'..'9' | '(' | ')' | '[' | ']' | ',' | ';' | '*' | WS | '>' | '<' | '@'
        | ':' | '?' | '&' | '!' | '{' | '}'| '\"' | '-' | '+' | '=' | '$' | '#' | '~' | '^' | '%' | '|' |'\'\'
       ' | ''')* '\'';
```

Listing B.1: Xtext grammar of the recursive disk metaphor for visualizing the structure of a software system.

Glossary

Abstract syntax

The abstract syntax of a language specifies the language's structure, typically as a tree or a graph [Stahl et al. 2006, p. 55ff]. [13]

Action

Actions make the creation of a return type explicit. There are simple actions and assigned actions [Xtext Documentation 2014]. [19]

Assignment

Assignments (=, +=,?=) are used to bind the consumed information to a feature of the currently produced object [Xtext Documentation 2014]. [19]

Behavior

The aspect behavior covers the execution of a software system with real and abstract data [Diehl 2007, p. 3f]. [6]

Component

Components are building blocks that are used to assemble different systems of a system family [Czarnecki and Eisenecker 2000, p. 9]. [11]

Concrete syntax

The concrete syntax is the realization of an abstract syntax that is accepted by a parser and used as notation to define models [Stahl et al. 2006, p. 55ff]. It may be textual, graphical, tabular, or a mix of these [Völter et al. 2013, p. 27]. [13]

Configuration knowledge

The configuration knowledge maps the elements of the problem space to the elements of the solution space and is usually implemented as a generator. The mapping process considers information about illegal feature combinations, default settings, default dependencies, construction rules, and optimizations [Czarnecki and Eisenecker 2000, p. 131f]. [12]

Data type rule

A data type rule creates instances of `EDataType` instead of `EClass`. They are similar to terminal rules but they are context sensitive and allow the use of hidden tokens [Xtext Documentation 2014]. [19]

Domain

A domain is a bounded field of knowledge comprising professional knowledge as well as technical knowledge and is always related to its stakeholders [Czarnecki and Eisenecker 2000, p. 34]. [10]

DSL

A DSL is specialized, problem-oriented, and provides means to describe concrete members of a system family [Czarnecki and Eisenecker 2000, p. 137]. [10]

Dynamic semantics

The dynamic semantics gives a meaning to the metamodel's constructs [Stahl et al. 2006, p. 55ff]. [14]

Enum rule

An enum rule returns enumeration literals from strings and creates an instance of `EEnum` [Xtext Documentation 2014]. [19]

Evolution

The aspect evolution refers to the development process of a software system [Diehl 2007, p. 3f]. This information is usually provided by version control systems, such as CVS, SVN, or Git. [6]

Feature

A feature combines related plug-ins and their fragments to a product [Eclipse Documentation 2014]. [17]

Formal model

A formal model is formulated in the metamodel's concrete syntax and follows its semantic, i.e., it is an instance of the metamodel [Stahl et al. 2006, p. 55ff]. [14]

Forward engineering

The forward engineering process covers the classical software development process from requirements engineering to the design and the implementation of the software system [Chikofsky and Cross 1990]. [8]

Fragment

A fragment is always part of a plug-in. It extends a plug-in non-invasively with further contents or functionality [Eclipse Documentation 2014]. For example, language packages are often implemented as fragments and added after the development of the plug-in has been finished. [17]

Generative domain model

The generative domain model summarizes the concepts and relations of the generative

paradigm. It consists of the problem space, the solution space and the configuration knowledge [Czarnecki and Eisenecker 2000, p. 131f]. [11]

Generator

A generator is a piece of software that produces a system automatically according to the specification [Czarnecki and Eisenecker 2000, p. 333ff]. [11]

GP

"Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge." [Czarnecki and Eisenecker 2000, p. 5] [10]

Grammar mixin

A grammar mixin combines at least two different Xtext grammars [Xtext Documentation 2014]. [18]

M2M

A model-to-model transformation maps one or more source models to a target model [Stahl et al. 2006, p. 60]. [15]

M2P

A model-to-platform transformation generates artifacts that are based on the platform. This process is also called generation [Stahl et al. 2006, p. 60]. [15]

Management

The management process covers the planning, coordinating, measuring, monitoring, controlling, and reporting of the software engineering process [Frailey et al. 2004]. [8]

MDSD

"Model-Driven Software Development (MDSD) is a generic term for techniques, that create runnable software from formal models automatically." [translated into English from Stahl et al. 2007, p. 11] [10]

Metamodel

The metamodel defines the abstract syntax and the static semantics of a language [Stahl et al. 2006, p. 55ff]. [13]

Model modification

A model modification changes or extends source model elements, i.e., the target model is the modified source model and still conforms to the same metamodel [Stahl et al. 2007, p. 199f]. [15]

Model transformation

A model transformation maps a source model to a target model, where both models conform to a different metamodel [Stahl et al. 2007, p. 199f]. [15]

Model weaving

A model weaving combines at least two source models and creates one target model [Stahl et al. 2007, p. 199f]. [15]

MWE2

Modeling Workflow Engine 2 (MWE2) is a declarative, externally configurable generator engine [Xtext Documentation 2014]. [18]

Parser rule

A parser rule uses terminal rules as well as other parser rules and leads to the parse tree [Xtext Documentation 2014]. [19]

Platform

A platform supports the realization of a domain and can be founded on existing building blocks, such as middleware, libraries, frameworks, or components [Stahl et al. 2006, p. 59]. [15]

Plug-in

A plug-in is the smallest functional unit in Eclipse. It can be an extension and it can offer extension points as placeholders for other plug-ins [Eclipse Documentation 2014]. [16]

Problem space

The problem space provides domain specific concepts and features to specify members of a system family by means of a DSL [Czarnecki and Eisenecker 2000, p. 131f]. [11]

Profile

A profile is composed by a predefined set of components where ach component is divided into levels describing increasing capability. Every X3D node belongs to a component and varies in features depending on the component level. There are five main profiles, namely `Core`, `Interchange`, `Interactive`, `Immersive`, and `Full` as well as three special profiles namely, `CADInterchange`, `MedicalInterchange`, and `MPEG-4 interactive` [Brutzman and Daly 2007, p. 13ff]. [25]

Reverse engineering

The reverse engineering process aims at gaining sufficient design-level understanding about an existing software system to help with its maintenance, enhancement, replacement, and reuse [Chikofsky and Cross 1990]. [8]

Scene graph

A scene graph is the basic unit of the X3D runtime environment. It is a directed, acyclic graph or a tree, respectively [Brutzman and Daly 2007, p. 1]. The nodes of this tree correspond to objects of the scene. [27]

Solution space

The solution space covers the elementary and reusable implementation components and the common system family architecture [Czarnecki and Eisenecker 2000, p. 131f]. [12]

Static semantics

The static semantics are a set of constraints and/or type system rules to which formal models have to conform [Stahl et al. 2006, p. 55ff]. [14]

Structure

The aspect structure includes program code, data structures, static call-graphs, relations, and the organization of a software system [Diehl 2007, p. 3f]. [6]

System family

A system family covers a set of of systems that are similar enough in terms of architecture to be assembled by a common set of components [Czarnecki and Eisenecker 2000, p. 31]. [10]

Technology projection

A technology projection is the mapping of the elements of the generative domain model to a paradigm, a programming language, or a platform [Czarnecki 2005]. [12]

Terminal rule

A terminal rule is described using EBNF-like expressions. Return types are atomic values of type `EDataType` [Xtext Documentation 2014]. [18]

Visualization pipeline

A visualization pipeline defines a visualization process including five main steps: extraction, analysis, filtering, mapping, and rendering [dos Santos and Brodlie 2004]. [9]

Workflow

In the context of MWE2, a workflow summarizes components that interact with each other [Xtext Documentation 2014]. [19]

X3D

Extensible 3D (X3D) is an XML-based file format and runtime architecture to represent 3D scenes [X3D Website 2014]. [5]

X3DOM

Extensible 3D Document Object Model (X3DOM) is an open-source framework and runtime architecture for 3D graphics on the web [X3DOM Website 2014]. [5]

Xtend

Xtend is a statically-typed programming language that translates to comprehensible Java source code [Xtend Documentation 2014]. [23]

Xtext

Xtext is a language development framework for programming languages and domain specific languages [Xtext Documentation 2014]. [17]

Bibliography

Alam, S. and Dugerdil, P. (2007). EvoSpaces: 3D Visualization of Software Architecture. In *Int. Conf. Softw. Eng. Knowl. Eng.*, pages 500–506. Knowledge Systems Institute Graduate School. [Cited on page 49]

Amar, R. and Stasko, J. (2004). A knowledge task-based framework for design and evaluation of information visualizations. In *IEEE Symp. Inf. Vis.*, pages 143–150. [Cited on page 66]

AOPT (2014). AOPT. `http://doc.x3dom.org/tutorials/models/aopt/index.html`. Accessed: 2014-10-13. [Cited on page 34]

Balogh, G. and Beszedes, A. (2013). CodeMetropolis – a Minecraft based collaboration tool for developers. In *1st IEEE Work. Conf. Softw. Vis.*, pages 1–4. [Cited on page 49]

Balzer, M., Noack, A., Deussen, O., and Lewerentz, C. (2004). Software landscapes: Visualizing the structure of large software systems. In *Proc. Sixth Jt. Eurographics - IEEE TCVG Conf. Vis.*, pages 261–266. Eurographics Association. [Cited on page 49]

Bassil, S. and Keller, R. (2001). Software visualization tools: Survey and analysis. In *9th Int. Work. Progr. Compr.*, pages 7–17. [Cited on pages 2 and 8]

Behr, J., Eschler, P., Jung, Y., and Zöllner, M. (2009). X3DOM: A DOM-based HTML5/X3D Integration Model. In *Proc. 14th Int. Conf. 3D Web Technol.*, pages 127–135, New York, USA. ACM. [Cited on pages 24 and 30]

Behr, J., Jung, Y., Drevensek, T., and Aderhold, A. (2011). Dynamic and interactive aspects of X3DOM. In *Proc. 16th Int. Conf. 3D Web Technol.*, pages 81–88, New York, USA. ACM. [Cited on page 24]

Behr, J., Jung, Y., Franke, T., and Sturm, T. (2012). Using images and explicit binary container for efficient and incremental delivery of declarative 3D scenes on the web. In *Proc. 17th Int. Conf. 3D Web Technol.*, pages 17–26, New York, USA. ACM. [Cited on pages 10, 24, and 34]

Behr, J., Jung, Y., Keil, J., Drevensek, T., Zoellner, M., Eschler, P., and Fellner, D. (2010). A scalable architecture for the HTML5/X3D integration model X3DOM. In *Proc. 15th Int. Conf. Web 3D Technol.*, pages 185–194, New York, USA. ACM. [Cited on pages 24, 30, and 31]

Boccuzzo, S. and Gall, H. (2007). CocoViz: Towards Cognitive Software Visualizations. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 72–79. IEEE. [Cited on page 49]

Bohnet, J. and Döllner, J. (2005). Konzepte der Softwarevisualisierung für komplexe, objektorientierte Softwaresysteme. Technical Report 6, Hasso-Plattner-Institut für Softwaretechnik, Universität Potsdam, Potsdam. [Cited on pages 2 and 8]

Borgo, R., Kehrer, J., Chung, D., Maguire, E., Laramee, R. S., Ward, M., and Chen, M. (2013). Glyph-based visualization: Foundations, design guidelines, techniques and applications. *Eurographics State Art Reports*, pages 39–63. [Cited on pages 86 and 87]

Brutzman, D. and Daly, L. (2007). *X3D: Extensible 3D graphics for Web authors*. Elsevier. [Cited on pages 24, 25, 27, XXIV, and XXV]

Caserta, P. and Zendra, O. (2011). Visualization of the Static Aspects of Software: A Survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933. [Cited on page 2]

Caserta, P., Zendra, O., and Bodénes, D. (2011). 3D Hierarchical Edge bundles to visualize relations in a software city metaphor. In *6th Int. Work. Vis. Softw. Underst. Anal.* [Cited on page 49]

Chikofsky, E. and Cross, J. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Softw.*, 7(1):13–17. [Cited on pages 1, 8, XXII, and XXIV]

Czarnecki, K. (2005). Overview of Generative Software Development. In Banâtre, J.-P., Fradet, P., Giavitto, J.-L., and Michel, O., editors, *Unconv. Program. Paradig.*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer Berlin Heidelberg. [Cited on pages 10, 12, and XXV]

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. [Cited on pages 10, 11, 34, 85, XXI, XXII, XXIII, XXIV, and XXV]

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–646. [Cited on page 15]

Daum, B. (2008). *Java-Entwicklung mit Eclipse 3.3: Anwendungen, Plugins und Rich Clients*. dpunkt.verlag. [Cited on page 17]

Diehl, S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer. [Cited on pages 2, 5, 6, 9, 34, 66, XXI, XXII, and XXV]

dos Santos, S. and Brodlie, K. (2004). Gaining understanding of multivariate and multidimensional data through visualization. *Comput. Graph.*, 28(3):311–325. [Cited on pages 9 and XXV]

Ducasse, S., Anquetil, N., Bhatti, U., Hora, A. C., Laval, J., and Gîrba, T. (2011). MSE and FAMIX 3.0: an interexchange format and source code model family. [Cited on pages 9, 10, 20, 34, and 57]

Ducasse, S., Gîrba, T., and Favre, J.-M. (2004). Modeling software evolution by treating history as a first class entity. In *Work. Softw. Evol. Through Transform.*, pages 75–86. [Cited on pages 9, 10, 34, 57, and 67]

Eclipse Documentation (2014). Eclipse Documentation. `http://help.eclipse.org/kepler/index.jsp`. Accessed: 2014-04-04. [Cited on pages 16, 17, XXII, and XXIV]

Eclipse Website (2014). Eclipse Website. `http://www.eclipse.org/`. Accessed: 2014-10-15. [Cited on page 15]

Eicker, S., Spies, T., and Kahl, C. (2007). Software Visualization in the Context of Service-Oriented Architectures. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 108–111. IEEE. [Cited on pages 6, 7, and 49]

Findbugs (2014). Findbugs. `http://findbugs.sourceforge.net/`. Accessed: 2014-11-05. [Cited on page 83]

Frailey, D., MacDonell, S., and Gray, A. (2004). Software engineering management. In Bourque, P. and Dupuis, R., editors, *Guid. to Softw. Eng. Body Knowl.*, chapter 8, pages 1–13. The Institute of Electrical and Electronics Engineers, Inc., AUT University. [Cited on pages 1, 8, and XXIII]

Freemind (2014). Freemind. `http://freemind.sourceforge.net/wiki/index.php/Main_Page`. Accessed: 2014-11-05. [Cited on page 88]

Gallagher, K., Hatch, A., and Munro, M. (2005). A Framework for Software Architecture Visualisation Assessment. In *3rd Int. Work. Vis. Softw. Underst. Anal.*, pages 76–81, Budapest, Hungary. IEEE. [Cited on pages 9 and 66]

Garcia, V. C., Lucredio, D., do Prado, A. F., Alvaro, A., and de Almeida, E. S. (2004). Towards an Effective Approach for Reverse Engineering. In *Proc. 11th Work. Conf. Reverse Eng.*, pages 298–299. IEEE. [Cited on pages 1 and 8]

Gračanin, D., Matković, K., and Eltoweissy, M. (2005). Software Visualization. *Innov. Syst. Softw. Eng.*, 1(2):221–230. [Cited on pages 2 and 66]

Greevy, O. (2007). Dynamix - a meta-model to support feature-centric analysis. In *1st Int. Work. FAMIX Moose Reengineering*. [Cited on pages 6, 9, 10, 34, 57, and 67]

Greevy, O., Lanza, M., and Wysseier, C. (2005). Visualizing Feature Interaction in 3-D. In *3rd Int. Work. Vis. Softw. Underst. Anal.*, pages 114–119. IEEE. [Cited on pages 7, 49, and 84]

Hassenzahl, M., Platz, A., Burmester, M., and Lehner, K. (2000). Hedonic and ergonomic quality aspects determine a software's appeal. In *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, pages 201–208. [Cited on page 75]

Herman, I., Melancon, G., and Marshall, M. S. (2000). Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Vis. Comput. Graph.*, 6(1):24–43. [Cited on page 67]

Höffler, T. N. and Leutner, D. (2007). Instructional animation versus static pictures: A meta-analysis. *Learn. Instr.*, 17(6):722–738. [Cited on page 66]

Hundhausen, C. D. (1996). A meta-study of software visualization effectiveness. `http://www.eecs.wsu.edu/~veupl/pub/MetaStudy.pdf`. [Cited on page 5]

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *J. Vis. Lang. Comput.*, 13(3):259–290. [Cited on page 8]

InstantReality Website (2014). InstantReality Website. `http://www.instantreality.org/`. Accessed: 2014-10-10. [Cited on pages 28 and 34]

Irani, P. and Ware, C. (2003). Diagramming information structures using 3D perceptual primitives. *ACM Trans. Comput. Interact.*, 10(1):1–19. [Cited on page 49]

JDT Project Website (2014). JDT Project Website. `http://www.eclipse.org/jdt/`. Accessed: 2014-10-15. [Cited on page 16]

Keim, D. A. and Schneidewind, J. (2007). Introduction to the Special Issue on Visual Analytics. *ACM SIGKDD*, 9(2):3–4. [Cited on page 67]

Kienle, H. M. and Müller, H. A. (2007). Requirements of Software Visualization Tools: A Literature Survey. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 2–9. IEEE. [Cited on page 48]

Knight, C. and Munro, M. (1999). Comprehension with[in] Virtual Environment Visualisations. In *7th Int. Work. Progr. Compr.*, page 4, Los Alamitos, CA, USA. IEEE. [Cited on page 6]

Koschke, R. (2003). Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. Softw. Maint.*, 15(2):87–109. [Cited on page 2]

Kuhn, A., Erni, D., and Nierstrasz, O. (2010). Embedding spatial software visualization in the IDE: an exploratory study. In *Proc. 5th Int. Symp. Softw. Vis.*, pages 113–122, New York, USA. ACM. [Cited on page 49]

Kuhn, A. and Verwaest, T. (2008). FAME—A polyglot library for metamodeling at runtime. In *Work. Model. Runtime*, pages 57–66. [Cited on pages 9 and 57]

Lee, B., Parr, C., Plaisant, C., and Bederson, B. (2005). Visualizing Graphs as Trees: Plant a seed and watch it grow. In *Graph Draw.*, volume 3843, pages 516–518, Berlin, Heidelberg. Springer-Verlag. [Cited on page 67]

Lethbridge, T. C., Tichelaar, S., and Ploedereder, E. (2004). The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. *Electron. Notes Theor. Comput. Sci.*, 94:7–18. [Cited on page 9]

Limberger, D., Wasty, B., Trümper, J., and Döllner, J. (2013). Interactive software maps for web-based source code analysis. In *Proc. 18th Int. Conf. 3D Web Technol.*, pages 91–98, New York, USA. ACM. [Cited on page 88]

Limper, M., Thöner, M., Behr, J., and Fellner, D. W. (2014). SRC - a Streamable Format for Generalized Web-based 3D Data Transmission. In *Proc. Ninet. Int. ACM Conf. 3D Web Technol.*, pages 35–43, New York, USA. ACM. [Cited on page 34]

Löwe, W. and Panas, T. (2005). Rapid construction of software comprehension tools. *Int. J. Softw. Eng. Knowl. Eng.*, 15(6):905–1023. [Cited on page 49]

Maletic, J., Marcus, A., and Collard, M. (2002). A task oriented view of software visualization. In *1st Int. Work. Vis. Softw. Underst. Anal.*, pages 32–40. IEEE. [Cited on pages 9 and 66]

Marcus, A., Comorski, D., and Sergeyev, A. (2005). Supporting the evolution of a software visualization tool through usability studies. In *13th Int. Work. Progr. Compr.*, pages 307–316. IEEE. [Cited on page 5]

Marcus, A., Feng, L., and Maletic, J. (2003). Comprehension of software analysis data using 3D visualization. In *11th Int. Work. Progr. Compr.*, page 105. IEEE Computer Society. [Cited on page 49]

Meyer, M., Sedlmair, M., and Munzner, T. (2012). The four-level nested model revisited: blocks and guidelines. In *Work. BEyond time errors Nov. Eval. methods Inf. Vis.*, pages 1–6. [Cited on pages 65, 66, and 86]

Miller, J. and Mukerji, J. (2003). MDA Guide Version 1.0.1. [Cited on page 12]

Müller, R., Kovacs, P., Schilbach, J., and Eisenecker, U. (2011). Generative Software Visualization: Automatic Generation of User-Specific Visualizations. In *Proc. Int. Work. Digit. Eng.*, pages 45–49, Magdeburg, Germany. [Cited on pages 2, 4, 56, 73, 74, and 85]

Müller, R., Kovacs, P., Schilbach, J., Eisenecker, U., Zeckzer, D., and Scheuermann, G. (2014a). A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization. In *Proc. 5th Int. Conf. Vis. Theory Appl.*, pages 204–209, Lisbon, Portugal. [Cited on pages 2, 4, 65, 66, 73, 74, and 86]

Müller, R., Kovacs, P., Schilbach, J., and Zeckzer, D. (2014b). How to Master Challenges in Experimental Evaluation of 2D versus 3D Software Visualizations. In *IEEE VIS 2014 Int. Work. 3DVis Does 3D really make sense Data Vis.*, Paris, France. [Cited on pages 2, 4, 73, 74, and 86]

Müller, R. and Zeckzer, D. (2015a). Past, Present, and Future of 3D Software Visualization - A Systematic Literature Analysis. In *Proc. 6th Int. Conf. Vis. Theory Appl.*, Berlin, Germany. [Cited on pages 2, 4, 85, and 87]

Müller, R. and Zeckzer, D. (2015b). The Recursive Disk Metaphor - A Glyph-based Approach for Software Visualization. In *Proc. 6th Int. Conf. Vis. Theory Appl.*, Berlin, Germany. [Cited on pages 4, 83, and 86]

Munzner, T. (2009). A nested model for visualization design and validation. *IEEE Trans. Vis. Comput. Graph.*, 15(6):921–928. [Cited on pages 65, 66, and 86]

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123. [Cited on pages 5, 8, and 65]

Nierstrasz, O., Ducasse, S., and Gîrba, T. (2005). The story of moose: an agile reengineering environment. In *Proc. 10th Eur. Softw. Eng. Conf. held jointly with 13th SIGSOFT Int. Symp. Found. Softw. Eng.*, volume 30, pages 1–10, Lisbon, Portugal. ACM. [Cited on page 67]

PDE Project Website (2014). PDE Project Website. `http://www.eclipse.org/pde/`. Accessed: 2014-10-15. [Cited on page 16]

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Proc. 12th Int. Conf. Eval. Assess. Softw. Eng.*, pages 68–77. British Computer Society. [Cited on pages 3, 48, and 85]

Price, B. A., Baecker, R. M., and Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *J. Vis. Lang. Comput.*, 4(3):211–266. [Cited on pages 5, 9, and 66]

Reiss, S. (2005). SoftVis 2005, Call for papers. `http://www.softvis.org/softvis05/`. Accessed: 2014-09-30. [Cited on page 6]

Reiss, S. P. (1995). An Engine for the 3D Visualization of Program Information. *J. Vis. Lang. Comput.*, 6(3):299–323. [Cited on pages 48 and 85]

Ripley, R. M., Sarma, A., and van der Hoek, A. (2007). A Visualization for Software Project Awareness and Evolution. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 137–144. IEEE. [Cited on pages 6, 7, and 49]

Roman, G.-C. and Cox, K. C. (1993). A taxonomy of program visualization systems. *IEEE Comput.*, 26(12):11–24. [Cited on pages 5, 9, and 66]

Sharif, B. and Jetty, G. (2013). An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension. In *1st IEEE Work. Conf. Softw. Vis.* [Cited on page 49]

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. 1996 IEEE Symp. Vis. Lang.*, pages 336–343. IEEE. [Cited on page 67]

Siegmund, J. (2012). *Framework for Measuring Program Comprehension*. Phd thesis, University of Magdeburg, School of Computer Science. [Cited on page 66]

Sjøberg, D. I. K., Dybå, T., and Jørgensen, M. (2007). The Future of Empirical Methods in Software Engineering Research. In *Futur. Softw. Eng.*, pages 358–378. IEEE. [Cited on page 3]

Stahl, T., Völter, M., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 1st edition. [Cited on pages 12, 13, 15, 34, 85, XXI, XXII, XXIII, XXIV, and XXV]

Stahl, T., Völter, M., Efftinge, S., and Haase, A. (2007). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2nd edition. [Cited on pages 12, 13, 15, XXIII, and XXIV]

Stasko, J. and Patterson, C. (1993). Understanding and Characterizing Program Visualization Systems. Technical report, Georgia Institute of Technology, Atlanta. [Cited on pages 9 and 66]

Steinbrückner, F. and Lewerentz, C. (2010). Representing development history in software cities. In *Proc. 5th Int. Symp. Softw. Vis.*, pages 193–202, New York, USA. ACM. [Cited on pages 6, 7, 49, and 84]

Storey, M.-A. D., Čubranić, D., German, D. M., and Cubranic, D. (2005). On the use of visualization to support awareness of human activities in software development. In *Proc. 2005 ACM Symp. Softw. Vis.*, pages 193–202, New York, USA. ACM. [Cited on pages 9 and 66]

Strein, D., Lincke, R., Lundberg, J., and Löwe, W. (2007). An Extensible Meta-Model for Program Analysis. *IEEE Trans. Softw. Eng.*, 33(9):592 – 607. [Cited on page 9]

Teyseyre, A. R. and Campo, M. R. (2009). An overview of 3D software visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(1):87–105. [Cited on pages 2 and 48]

Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. [Cited on pages 12, 13, and XXI]

vom Brocke, J., Simons, A., Niehaves, B., Riemer, K., Plattfaut, R., and Cleven, A. (2009). Reconstructing the giant: On the importance of rigour in documenting the literature search process. In *17th Eur. Conf. Inf. Syst.*, pages 1–13. [Cited on pages 3, 48, and 85]

von Pilgrim, J. and Duske, K. (2008). GEF3D: a framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *Proc. 4th ACM Symp. Softw. Vis.*, pages 95–104, New York, USA. ACM. [Cited on pages 6, 7, and 49]

Vuze (2014). Vuze. `http://sourceforge.net/projects/azureus/`. Accessed: 2014-11-09. [Cited on page 83]

Waller, J., Wulf, C., Fittkau, F., Döhring, P., and Hasselbring, W. (2013). SynchroVis : 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *1st IEEE Work. Conf. Softw. Vis.*, pages 7–10. [Cited on page 49]

Ward, M. O. (2008). Multivariate Data Glyphs: Principles and Practice. In *Handb. Data Vis.*, pages 179–198. Springer. [Cited on page 83]

Wettel, R. and Lanza, M. (2007). Visualizing Software Systems as Cities. In *4th Int. Work. Vis. Softw. Underst. Anal.*, pages 92–99. IEEE. [Cited on pages 6, 7, 49, and 88]

Wettel, R., Lanza, M., and Robbes, R. (2011). Software systems as cities: A controlled experiment. In *Proc. 33rd Int. Conf. Softw. Eng.*, pages 551–560, Waikiki, Honolulu, USA. ACM. [Cited on page 66]

Wilde, T. and Hess, T. (2007). Forschungsmethoden der Wirtschaftsinformatik: Eine empirische Untersuchung. *Wirtschaftsinformatik*, 49:280–287. [Cited on page 3]

X3D Schema (2014). X3D Schema. `http://www.web3d.org/specifications/x3d-3.3.xsd`. Accessed: 2014-10-12. [Cited on pages 24 and 57]

X3D Standard (2014). X3D Standard. `http://www.web3d.org/standards/`. Accessed: 2014-09-18. [Cited on pages 24, 26, 27, and 28]

X3D Website (2014). X3D Website. `http://www.web3d.org/x3d/`. Accessed: 2014-09-18. [Cited on pages 24, 28, and XXV]

X3DOM Fallback Model (2014). X3DOM Fallback Model. `http://www.x3dom.org/wp-content/uploads/2009/10/x3dom-fallback-Release-1.2.png`. Accessed: 2014-10-13. [Cited on page 32]

X3DOM HTML Profile (2014). X3DOM HTML Profile. `http://www.x3dom.org/?page_id=158`. Accessed: 2014-10-13. [Cited on page 30]

X3DOM Website (2014). X3DOM Website. `http://www.x3dom.org/`. Accessed: 2014-09-30. [Cited on pages 30 and XXVI]

Xtend Documentation (2014). Xtend Documentation. http://www.eclipse.org/xtend/documentation.html. Accessed: 2014-10-15. [Cited on pages 23 and XXVI]

Xtext Documentation (2014). Xtext Documentation. http://www.eclipse.org/Xtext/documentation.html. Accessed: 2014-10-15. [Cited on pages 17, 18, 19, XXI, XXII, XXIII, XXIV, XXV, and XXVI]

Yi, J. S., ah Kang, Y., Stasko, J. T., and Jacko, J. A. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1224–1231. [Cited on page 67]

Wissenschaftlicher Werdegang

Allgemeine Hochschulreife

09/1994–06/2002	Robert-Schumann-Gymnasium in Leipzig Leistungskurse: Mathematik und Englisch Abschluss: Abitur (Note 1,4)

Wehrdienst

07/2002–06/2003	Heeresflieger in Fritzlar

Studium

10/2003–02/2009	Studium der Wirtschaftsinformatik an der Universität Leipzig Thema der Diplomarbeit: *Konzeption und prototypische Implementierung eines Generators zur Softwarevisualisierung in 3D* Abschluss: Diplom-Wirtschaftsinformatiker (Note 1,5)
02/2005–01/2009	Studentische Hilfskraft am Lehrstuhl für Wirtschaftsinformatik, insbes. Softwareentwicklung für Wirtschaft und Verwaltung Übungen: *Strukturierte Programmierung*, *Objektorientierte und generische Programmierung*, *Entwicklung verteilter Anwendungen*

Wissenschaftliche Tätigkeiten

11/2008–09/2009	Mitarbeiter im Forschungsprojekt *BEFAS* bei Volkswagen
02/2009–1/2015	Wissenschaftlicher Mitarbeiter und Doktorand am Lehrstuhl für Wirtschaftsinformatik, insbes. Softwareentwicklung für Wirtschaft und Verwaltung Betreuung von Abschlussarbeiten (Bachelor, Master, Diplom) Vorlesungen: *Anforderungsermittlung und Softwareergonomie* (seit 04/2010), *Softwarevisualisierung* (seit 04/2012), *Entwicklung für und mit Eclipse* (10/2012–02/2013)
seit 01/2010	Mitarbeiter im Forschungsprojekt *Softwarevisualisierung in 3D und VR*

Reviewtätigkeiten

SKIL'12 (PC), SKIL'11 (PC), Computer and Informatics (eingeladen)

Leipzig, den 21. Januar 2015 Richard Müller

Selbstständigkeitserklärung

Hiermit versichere ich, dass

1. die vorgelegte Dissertation ohne unzulässige Hilfe, insbesondere ohne die Inanspruchnahme eines Promotionsberaters, und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde und dass die aus fremden Quellen direkt oder indirekt übernommenen Gedanken in der Arbeit als solche kenntlich gemacht worden sind und

2. die vorgelegte Dissertation weder im Inland noch im Ausland in gleicher oder in ähnlicher Form einer anderen Prüfungsbehörde zum Zwecke einer Promotion oder eines anderen Prüfungsverfahrens vorgelegt und insgesamt noch nicht veröffentlicht wurde.

Leipzig, den 21. Januar 2015 Richard Müller