Live Trace Visualization for System and Program Comprehension in Large Software Landscapes

Dissertation

M.Sc. Florian Fittkau

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2015

Kiel Computer Science Series (KCSS) 2015/7 v1.0

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via http://www.explorviz.net

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

▻ Florian Fittkau. *Live Trace Visualization for System and Program Comprehension in Large Software Landscapes.* Number 2015/7 in Kiel Computer Science Series. Department of Computer Science, 2015. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Fittkau2015,
  author    = {Florian Fittkau},
  title     = {Live Trace Visualization for System and Program Comprehension
                in Large Software Landscapes},
  publisher = {Department of Computer Science, Kiel University},
  year      = {2015},
  month     = dec,
  number    = {2015/7},
  isbn      = {978-3-7392-0716-2},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering, Kiel University.}
}
```

© 2015 by Florian Fittkau

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Herstellung und Verlag: BoD – Books on Demand, Norderstedt

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Wilhelm Hasselbring Christian-Albrechts-Universität zu Kiel
2. Gutachter:	Prof. Dr. Michele Lanza University of Lugano

Datum der mündlichen Prüfung: 30. November 2015

Zusammenfassung

In vielen Unternehmen nimmt die Anzahl der eingesetzten Anwendungen stetig zu. Diese Anwendungen – meist mehrere hunderte – bilden große Softwarelandschaften. Das Verständnis dieser Softwarelandschaften wird häufig erschwert durch, beispielsweise, Erosion der Architektur, personelle Wechsel oder sich ändernde Anforderungen. Des Weiteren können Ereignisse wie Performance-Anomalien häufig nur in Verbindung mit den Anwendungszuständen verstanden werden. Deshalb wird ein möglichst effizienter und effektiver Weg zum Verständnis solcher Softwarelandschaften in Verbindung mit den Details jeder einzelnen Anwendung benötigt.

In dieser Arbeit führen wir einen Ansatz zur live Trace Visualisierung zur Unterstützung des System- und Programmverständnisses von großen Softwarelandschaften ein. Dieser verwendet zwei Perspektiven: eine Landschaftsperspektive mit UML Elementen und eine Applikationsperspektive, welche der 3D Softwarestadtmetapher folgt. Unsere Hauptbeiträge sind 1) ein Ansatz, genannt ExplorViz, um live Trace Visualisierung von großen Softwarelandschaften zu ermöglichen, 2) ein Überwachungs- und Analyseansatz, welcher in der Lage ist die große Anzahl an Methodenaufrufen in einer großen Softwarelandschaft aufzuzeichnen und zu verarbeiten und 3) Anzeige- und Interaktionskonzepte für die Softwarestadtmetapher, welche über klassische 2D Anzeige und 2D Eingabegeräten hinausgehen.

Umfassende Laborexperimente zeigen, dass unser Überwachungs- und Analyseansatz für große Softwarelandschaften elastisch skaliert und dabei nur einen geringen Overhead auf den Produktivsystemen erzeugt. Des Weiteren demonstrieren mehrere kontrollierte Experimente eine gesteigerte Effizienz und Effektivität beim Lösen von Verständnisaufgaben unter Verwendung unserer Visualisierung. ExplorViz ist als Open Source Anwendung verfügbar unter `www.explorviz.net`. Zusätzlich stellen wir umfangreiche Pakete für unsere Evaluierungen zur Verfügung um die Nachvollziehbarkeit und Wiederholbarkeit unserer Ergebnisse zu ermöglichen.

Abstract

In many enterprises, the number of deployed applications is constantly increasing. Those applications – often several hundreds – form large software landscapes. The comprehension of such landscapes is frequently impeded due to, for instance, architectural erosion, personnel turnover, or changing requirements. Furthermore, events such as performance anomalies can often only be understood in correlation with the states of the applications. Therefore, an efficient and effective way to comprehend such software landscapes in combination with the details of each application is required.

In this thesis, we introduce a live trace visualization approach to support system and program comprehension in large software landscapes. It features two perspectives: a landscape-level perspective using UML elements and an application-level perspective following the 3D software city metaphor. Our main contributions are 1) an approach named ExplorViz for enabling live trace visualization of large software landscapes, 2) a monitoring and analysis approach capable of logging and processing the huge amount of conducted method calls in large software landscapes, and 3) display and interaction concepts for the software city metaphor beyond classical 2D displays and 2D pointing devices.

Extensive lab experiments show that our monitoring and analysis approach elastically scales to large software landscapes while imposing only a low overhead on the productive systems. Furthermore, several controlled experiments demonstrate an increased efficiency and effectiveness for solving comprehension tasks when using our visualization. ExplorViz is available as open-source software on `www.explorviz.net`. Additionally, we provide extensive experimental packages of our evaluations to facilitate the verifiability and reproducibility of our results.

Preface
by Prof. Dr. Wilhelm Hasselbring

Software visualization is a non-trivial research field, and with his thesis Florian Fittkau has made original contributions to it. Florian Fittkau investigates how "live trace visualization" can be leveraged for the analysis and comprehension of large software landscapes. Specific contributions are the ExplorViz method with its monitoring and trace processing for landscape model generation, as well its 3D visualization.

Highly innovative are the new techniques for printing such 3D visualizations into physical models for improved program comprehension in teams and the new techniques for immersion into these 3D visualizations via topical virtual reality equipment.

Besides the conceptual work, this work contains a significant experimental part and a multifaceted evaluation. This engineering dissertation has been extensively evaluated with advanced student and lab experiments, based on a high-quality implementation of the ExplorViz tools.

This thesis is a good read and I recommend it to anyone interested in recent software visualization research.

Wilhelm Hasselbring
Kiel, December 2015

Contents

1 Introduction **1**
1.1 Motivation and Problem Statement 3
1.2 Scientific Contributions 4
1.3 Preliminary Work 10
1.4 Document Structure 22

I Foundations **27**

2 Application Monitoring **29**
2.1 Instrumentation 30
2.2 Trace Concept 33
2.3 Live versus Offline Analysis 34

3 Software Visualization **37**
3.1 Visualization Pipeline 39
3.2 Software City Metaphor 41
3.3 Trace Visualization 57
3.4 Remote-Procedure-Call Visualization 66

II The ExplorViz Approach **75**

4 Research Design **77**
4.1 Research Methods 79
4.2 Research Questions and Research Plan 80

5 The ExplorViz Method **87**
5.1 Fundamental Approach 89
5.2 Assumptions 91

6 Monitoring and Trace Processing **95**
6.1 Application-Level Monitoring 97
6.2 Remote-Procedure-Call Monitoring 99
6.3 Trace Processing 101

7 Landscape Model Generation **111**
7.1 Landscape Meta-Model 114
7.2 Trace-to-Model Mapping 115

8 ExplorViz Visualization **117**
8.1 Use Cases 119
8.2 Basic Concepts 120
8.3 Landscape-Level Perspective 123
8.4 Application-Level Perspective 128
8.5 Gesture-Controlled Virtual Reality Approach 139
8.6 Physical 3D-Printed City Models 144

III Evaluation **157**

9 ExplorViz Implementation **159**
9.1 Overall Architecture 161
9.2 Monitoring 162
9.3 Analysis 163
9.4 Repository 164
9.5 Visualization Provider 165

10 Monitoring and Trace Processing Evaluation: Lab Experiments **169**
10.1 Goals 171
10.2 Monitoring Overhead Evaluation 171
10.3 Live Trace Processing Evaluation 174
10.4 Scalability and Elasticity Evaluation 178
10.5 Summary 186

11 Visualization Evaluation: Controlled Experiments **189**
11.1 Goals . . . 191
11.2 Application Perspective Evaluation . . . 191
11.3 3D-Printed City Model Evaluation . . . 215
11.4 Landscape Perspective Evaluation . . . 235
11.5 Summary . . . 256

12 Extensibility Evaluation: Integrating a Control Center Concept **259**
12.1 Goals . . . 261
12.2 Control Center Concept . . . 261
12.3 Case Study . . . 268
12.4 Summary . . . 272

13 Related Work **275**
13.1 Monitoring and Trace Processing . . . 277
13.2 Visualization . . . 279
13.3 Evaluation . . . 283

IV Conclusions and Future Work **287**

14 Conclusions **289**
14.1 Monitoring and Trace Processing . . . 290
14.2 Visualization . . . 290
14.3 Evaluation . . . 291

15 Future Work **295**
15.1 Monitoring and Trace Processing . . . 297
15.2 Visualization . . . 297
15.3 Evaluation . . . 299

V Appendix **303**

A Experimental Data for the Extensibility Evaluation **305**
A.1 Questionnaire . . . 306

List of Figures 307

List of Tables 313

List of Listings 315

Acronyms 317

Bibliography 319

1 Introduction

This chapter provides an introduction to this thesis. Chapter 1.1 describes the motivation for our research. Afterwards, Chapter 1.2 presents the scientific contributions. Preliminary work is discussed in Chapter 1.3. Finally, Chapter 1.4 lists the structure of this thesis.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

2. [Fittkau 2013] F. Fittkau. Live trace visualization for system and program comprehension in large software landscapes. Technical report 1310. Department of Computer Science, Kiel University, Germany, Nov. 2013

1.1 Motivation and Problem Statement

In many enterprises, the number of software systems is constantly increasing. This can be a result of changing requirements due to, e.g., changing laws or customers, which the company has to satisfy. Furthermore, the legacy systems often interact with each other through defined interfaces. For example, the database may be accessed by different programs. In the whole, the applications form a large, complex software landscape [Penny 1993] which can include several hundreds or even thousands of applications.

The knowledge of the communication, internals, and utilization of this software landscape often gets lost over the years [Moonen 2003; Vierhauser et al. 2013] due to, for instance, missing documentation. For those software landscapes, tools that support the program and system comprehension of the software landscape become important. For example, they can provide essential insights into the landscape in the maintenance phase [Lewerentz and Noack 2004]. A software engineer might need to create or adapt features in the landscape. Therefore, she often needs to know the communication between the existing programs and also the control flow inside the application is of interest to find the locations where she needs to do the adaptations [Koschke and Quante 2005]. In this context, the goal of DFG SPP1593 "Design For Future - Managed Software Evolution" is to invent approaches for so-called knowledge-carrying software to overcome the challenges of missing documentation [Goltz et al. 2015].

Another challenge concerning a large software landscape is the question which applications are actually used and to what extent they are used. The operation and support of software can cause substantial costs. These costs would not incur when the unused application gets removed from the software landscape. However, asking every user whether she uses each application is often not applicable and if it is, she might indirectly use applications such as a database, for instance.

Recent approaches in this field of software visualization, e.g., [Panas et al. 2003; Greevy et al. 2006; Wettel and Lanza 2007; Hamou-Lhadj 2007; Dugerdil and Alam 2008], focus on the visualization of a single application. A drawback of visualizing only one application is omitting the communication and linkage between the applications involved in a transaction.

Another drawback of current approaches is the possible lack of traces associated to a feature. For example, a software engineer might analyze a feature called *add to cart*. The investigation of this feature might lead to interest in the related feature *checkout cart*. However, this feature might not be available as a trace. Often the required trace can be generated manually for one application but this can become cumbersome in a large software landscape. In addition, one trace can only reveal information on its particular execution of operations, for instance, the response time of this single execution of a operation. If this response time is a statistical outlier, the user might draw false conclusions about the application.

Due to the huge amount of method calls conducted in a large software landscape – typically millions of method calls per second –, monitoring and creating the required traces of the executions for the visualization can become a further challenge [Vierhauser et al. 2013]. One server is not capable of processing such a huge amount of data in parallel to the actual execution of the software landscape.

1.2 Scientific Contributions

This thesis makes the following three major scientific contributions (SC1 – SC3) including nine subcontributions:

SC1: An approach named ExplorViz for enabling live trace visualization of large software landscapes

- **SC1.1:** A software landscape visualization featuring hierarchies to provide visual scalability
- **SC1.2:** An interactive extension of the software city metaphor for exploring runtime information of the monitored application
- **SC1.3:** A landscape meta-model representing gathered information about a software landscape
- **SC1.4:** A proof-of-concept implementation used in three controlled experiments for comparing our visualization approach to the current state of the art in system and program comprehension scenarios

SC2: A monitoring and analysis approach capable of logging and processing the huge amount of conducted method calls in large software landscapes

- **SC2.1:** A scalable, elastic, and live analysis architecture for processing the gathered monitoring data by using cloud computing
- **SC2.2:** A proof-of-concept implementation used in three lab experiments showing the low overhead of the monitoring approach, and the scalability and elasticity of our analysis approach by monitoring up to 160 instances of a web application

SC3: Display and interaction concepts for the software city metaphor beyond classical 2D displays and 2D pointing devices

- **SC3.1:** A gesture-controlled virtual reality approach for the software city metaphor
- **SC3.2:** An approach to create physical 3D-printed models following the software city metaphor
- **SC3.3:** Proof-of-concept implementations and a controlled experiment comparing physical 3D-printed models to using virtual models on the computer screen in a team-based program comprehension scenario

For all evaluations, we provide experimental packages to facilitate the verifiability, reproducibility, and further extensibility of our results. In the following, each contribution is described.

SC1: ExplorViz Approach For Enabling Live Trace Visualization of Large Software Landscapes

The first scientific contribution (SC1) of this thesis is an approach to enable live trace visualization for large software landscapes named ExplorViz which supports a software engineer during system and program comprehension tasks. Our live trace visualization for large software landscapes combines distributed and application traces. It contains a 2D visualization on the landscape level. In addition, it features a 3D visualization utilizing

the software city metaphor on the application level. By *application level*, we refer to the issues concerning one application and only this application. Whereas the *landscape level* provides knowledge about the different applications and nodes in the software landscape.

Since a live visualization updates itself after a defined interval, we feature a time shift mode where the software engineer can view the history of old states of the software landscape. Furthermore, she is able to jump to an old state and pause the visualization to analyze a specific situation.

To cope with the high density of information which should be visualized, the major concept of ExplorViz is based on interactively revealing additional details, e.g., the communication on a deeper system level, on demand. The concept is motivated by the fact that the working memory capacity of humans is limited to a small amount of chunks [Ware 2013]. Miller [1956] suggests seven, plus or minus two, chunks which is also referred to as Miller's Law. The ExplorViz concept also follows Shneiderman's Visual Information-Seeking Mantra: "Overview first, zoom and filter, then details on demand" [Shneiderman 1996].

This contribution contains four subcontributions (SC1.1 – SC1.4) which are briefly described in the following.

SC1.1: Software Landscape Visualization Featuring Hierarchies to Provide Visual Scalability Our landscape-level perspective shows the nodes and applications of a software landscape. In addition, it summarizes nodes running the same application configuration into node groups. These equal application configurations typically exist in cloud environments. However, to understand the overall architecture of the software landscape, the user is interested in the existing application configuration. Afterwards, the details about the concrete instances can be interactively accessed.

To provide further visual scalability, the nodes and node groups are visualized within their belonging systems which act as an organizational unit. Again, the details about a system can be accessed interactively and out-of-focus systems can be closed to show only details about relevant systems.

SC1.2: Interactive Extension of the Software City Metaphor for Exploring Runtime Information On the application level, we use the 3D software city metaphor to display the structure and runtime information of a monitored application. Again, the visual scalability is provided by interactivity. When accessing the perspective, the components are only opened at the top-level, i.e., details are hidden. In our terms, components are organizational units provided by the programming language, e.g., packages in Java. By interactively opening and closing the components, the software engineer is able to explore the application and the gathered runtime information.

SC1.3: Landscape Meta-Model for Representing Information of a Software Landscape Furthermore, we provide a landscape meta-model for representing the gathered information of the software landscape. This model can be used as input for other tools. Thus, the gathered data is also reusable for other scenarios, e.g., automatically updating the configuration of an enterprise application landscape based on the monitoring data.

SC1.4: Proof-of-Concept Implementation Used in Three Controlled Experiments The full ExplorViz approach is implemented as open-source software and available from our website.[1] To evaluate our live trace visualization approach, we conducted three controlled experiments.

The first controlled experiment compared the usage of ExplorViz to using the trace visualization tool Extravis [Cornelissen et al. 2007] in a program comprehension scenario of the quality tool PMD.[2] The experiment showed that ExplorViz was more efficient and effective than Extravis in supporting the solving of the defined program comprehension tasks. The second experiment was a replication of this experiment design where we used a smaller object system named Babsi.[3] In this replication, the used time difference was not significant. However, the correctness of the task solution was significantly increased in the ExplorViz group. The third experiment compared our hierarchical landscape-level perspective to a

[1] `http://www.explorviz.net`

[2] `https://pmd.github.io`

[3] `http://babsi.sf.net`

mix of flat state-of-the-art landscape visualizations found in Application Performance Management (APM) tools in a system comprehension scenario. Again, the time difference was not significantly different but the correctness of the solutions was significantly increased in the ExplorViz group.

SC2: Monitoring and Analysis Approach for Applications in Large Software Landscapes

In large software landscapes, several millions of method calls can be conducted each second. Therefore, the monitoring and the analysis approach requires to scale with the size of the software landscape. Furthermore, the approach should be elastic to avoid producing unnecessary costs. A further requirement for the approach is the low overhead of the monitoring to keep the impact on the production systems as low as possible. According to those requirements, we developed our monitoring and analysis approach which is outlined in the following.

SC2.1: Scalable, Elastic, and Live Analysis Architecture Using Cloud Computing To provide a scalable, elastic, and live monitored data analysis approach, we utilize cloud computing and an automatic capacity manager named *CapMan*.[4] Our approach is similar to the MapReduce pattern [Dean and Ghemawat 2010] but we feature multiple dynamically inserted preprocessing levels. When the master analysis node impends to get overutilized, a new preprocessing worker level is automatically inserted between the master and the monitored applications and thus the CPU utilization of the master node is decreased. If it impends to get overutilized again, another level of workers is inserted. In theory, this happens every time the master impends to get overutilized. If a worker level is not utilized enough anymore, it is dynamically removed and thus resources are saved.

SC2.2: Proof-of-Concept Implementation Used in Three Lab Experiments We implemented our monitoring and analysis approach as proof-of-concept implementation and provide necessary additional components such as the

[4] https://github.com/ExplorViz/capacity-manager

capacity manager as open-source software on our website. For the evaluation of our monitoring and analysis approach, we conducted three lab experiments.

We evaluated the low overhead in the first lab experiment by comparing Kieker[5] [van Hoorn et al. 2012], which was already shown to impose a low overhead [Eichelberger and Schmid 2014], to our monitoring component using the monitoring benchmark MooBench [Waller 2014].[6] As a result, we achieved a speedup of about factor nine and a 89 % overhead reduction.

The second lab experiment extended the first experiment by the live analysis of the generated monitoring data. This experiment showed that adding the analysis step only negligibly impacts the throughput and thus is capable of live analyzing the monitored data. Furthermore, we achieved a speedup of about 250 in comparison to Kieker.

We used our private cloud for the third lab experiment to evaluate the scalability and elasticity of our approach by monitoring elastically scaled JPetStore[7] instances. In the peak, 160 JPetStore instances were monitored by our approach with two dynamically started worker levels resulting in about 20 million analyzed method calls per second.

SC3: Display and Interaction Concepts for the Software City Metaphor

In addition to providing a live trace visualization, we investigated new ways to display and interact with the software city metaphor [Knight and Munro 1999] beyond the display on classical 2D monitors and usage of classical 2D pointing devices. For a more immersive user experience, we provide a Virtual Reality (VR) approach featuring an Oculus Rift DK1[8] as display and a Microsoft Kinect v2[9] for gesture recognition. Furthermore, we construct physical 3D-printed software city models from our application-level perspective to enhance, for instance, the amount of conducted gestures in a team-based program comprehension scenario. Both approaches and an evaluation are described in the following.

[5] `kieker-monitoring.net`

[6] `http://kieker-monitoring.net/research/projects/moobench`

[7] `http://ibatisjpetstore.sf.net`

[8] `http://www.oculus.com`

[9] `http://www.microsoft.com/en-us/kinectforwindows`

SC3.1: Gesture-Controlled Virtual Reality Approach By using an Oculus Rift DK1 and Microsoft Kinect v2 for our VR approach, we achieve a more immersive user experience for exploring the software city metaphor. The Oculus Rift enables to perceive the model in 3D as if the user is flying above the city. To provide an even more immersive experience, we utilize gestures for interacting with the model.

SC3.2: Approach to Create Physical 3D-Printed Software City Models We construct physical 3D-printed models following the software city metaphor of our application-level perspective and detail four envisioned scenarios where physical models could provide benefits. These are team-based program comprehension, effort visualization in customer dialog, saving digital heritage, and educational visualization.

SC3.3: Proof-of-Concept Implementations and a Controlled Experiment for Physical 3D-Printed Models For both approaches, we provide proof-of-concept implementations available in branches of our ExplorViz Git repository.[10] Furthermore, we conducted a controlled experiment investigating the first envisioned usage scenario for the physical models in a team-based program comprehension scenario. Teams (pairs of two subjects) in the experimental group solved program comprehension tasks using only a 3D-printed model and the control group solved the tasks using a virtual model on the computer screen. Two discussion tasks were influenced positively by using the 3D-printed model and one task was influenced negatively. We attribute the positive influence to an observed increased amount of conducted gestures and the negative influence to less readable labels in the 3D-printed model.

1.3 Preliminary Work

This thesis builds on preliminary work which was already published in several research papers. Furthermore, it bases on various student theses

[10] `https://github.com/ExplorViz/ExplorViz`

which were co-supervised by the author. In the following, we first briefly describe and list our publications according to three categories: *Approach*, *Evaluations*, and *Support Projects*. Papers fall into the former two categories if they are closely related and explicitly contribute to those parts of this thesis. The latter category contains work that is related but only indirectly contributes to this thesis. Afterwards, the related student theses and their contributions to this thesis are briefly presented.

Approach

▹ [Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

  In this publication, we present our overall ExplorViz method and each of its steps. Furthermore, first sketches of the landscape-level and application-level perspective are shown.

▹ [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013

  In this work, we describe the idea of multiple worker levels for analyzing the huge amount of generated monitoring records. Therefore, a worker and master concept and a scaling architecture are introduced. In addition, we show the MooBench benchmark results for comparing the analysis component of Kieker 1.8 and ExplorViz.

▹ [Fittkau 2013] F. Fittkau. Live trace visualization for system and program comprehension in large software landscapes. Technical report 1310. Department of Computer Science, Kiel University, Germany, Nov. 2013

This technical report presents a plan of the contributions of this thesis and details an evaluation scenario for the application-level perspective.

▻ [Fittkau et al. 2014a] F. Fittkau, P. Stelzer, and W. Hasselbring. Live visualization of large software landscapes for ensuring architecture conformance. In: *Proceedings of the 2nd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2014)*. ACM, Aug. 2014

Architecture conformance checking is introduced as a further usage scenario beneath supporting system and program comprehension in this paper. Furthermore, we present a preliminary study of the scalability and thus applicability of our analysis approach.

▻ [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

Performance analysis is a further usage scenario of our approach. Beneath introducing important aspects of the functionality for the performance analysis, we exemplify it on monitoring data gathered from the Perl-based application EPrints[11] in this publication.

▻ [Fittkau et al. 2015f] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

In this work, we present our approach to use VR for exploring the application-level perspective to provide an immersive experience. To enable VR, we use an Oculus Rift and provide further gesture-based interaction possibilities using a Microsoft Kinect.

[11] `http://www.eprints.org`

▻ [Fittkau et al. 2015i] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

The approach of constructing physical 3D models of our application-level perspective is presented in this work. Additionally, four potential usage scenarios for these physical models are described.

Evaluations

▻ [Fittkau et al. 2014b] F. Fittkau, A. van Hoorn, and W. Hasselbring. Towards a dependability control center for large software landscapes. In: *Proceedings of the 10th European Dependable Computing Conference (EDCC 2014)*. IEEE, May 2014

IT administrators often lack trust in automatic adaption approaches for their software landscapes. Therefore, we developed a semi-automatic control center concept which is presented in this publication. This control center concept is used as a target specification in our extensibility evaluation for the ExplorViz implementation.

▻ [Waller et al. 2014a] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance 2014 (SOSP 2014)*. University of Stuttgart, Nov. 2014

In this publication, we present a structured benchmark-driven performance tuning approach exemplified on the basis of Kieker. The last performance tuning step is equal to our developed monitoring component of ExplorViz. Therefore, the paper contains a performance comparison between Kieker and the monitoring component of ExplorViz.

▻ [Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. IEEE, May 2015

Providing efficient and effective tools to gain program comprehension is essential. Therefore, we compare the application-level perspective of ExplorViz to the trace visualization tool Extravis in two controlled experiments to investigate which visualization is more efficient and effective in supporting the program comprehension process.

▻ [Fittkau et al. 2015j] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. Technical report 1507. Department of Computer Science, Kiel University, Germany, June 2015

Basing on the application-level visualization, we construct physical, solid 3D-printed models of the application. This publication presents the results of a controlled experiment comparing the usage of physical models to using virtual, on-screen models in a team-based program comprehension process.

▻ [Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015

In this publication, we present our scalable and elastic approach for processing the monitoring data. Furthermore, we describe an evaluation where we monitor 160 JPetStore instances and use dynamically inserted analysis worker levels.

▻ [Fittkau et al. 2015h] F. Fittkau, A. Krause, and W. Hasselbring. Hierarchical software landscape visualization for system comprehension: a controlled experiment. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

This work describes a controlled experiment where we compare our landscape-level perspective with a landscape visualization derived from current APM tools. As demo landscape, we modeled the technical IT infrastructure of the Kiel University.

Support Projects

▻ [Fittkau et al. 2012a] F. Fittkau, S. Frey, and W. Hasselbring. CDOSim: simulating cloud deployment options for software migration support. In: *Proceedings of the 6th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2012)*. IEEE, Sept. 2012

We developed a simulator to rate one Cloud Deployment Option (CDO) named *CDOSim*. After simulating one CDO, the simulator provides an overall rating of the option which is constructed from three metrics, i.e., response times, costs, and Service Level Agreement (SLA) violations.

▻ [Fittkau et al. 2012b] F. Fittkau, S. Frey, and W. Hasselbring. Cloud user-centric enhancements of the simulator CloudSim to improve cloud deployment option analysis. In: *Proceedings of the 1st European Conference on Service-Oriented and Cloud Computing (ESOCC 2012)*. Springer, Sept. 2012

As basis for CDOSim, the cloud simulator *CloudSim* was used. Since CloudSim assumes that the user intends to simulate how her infrastructure performs as a cloud platform, we had to integrate the user-centric perspective of a cloud environment user. These enhancements are detailed in this work.

▻ [Frey et al. 2013] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE, May 2013

CDOSim rates one CDO. However, a software engineer conducting a migration to a cloud environment intends to find the most suitable deployment for her context. Therefore, the tool *CDOXplorer* generates CDOs and rates them by calling CDOSim. Since often a huge amount of options exists, CDOXplorer uses genetic algorithms to generate promising CDOs.

▻ [Waller et al. 2013] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring. SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

SynchroVis uses the 3D city metaphor to display concurrency of one application. It requires to do a static analysis before the program traces can be loaded. Concurrency, e.g., acquiring and releasing a lock object, is displayed through a special lock building. The different threads are visualized through different colors.

▻ S. Frey, F. Fittkau, and W. Hasselbring. CDOXplorer: Simulation-Based Genetic Optimization of Software Deployment and Reconfiguration in the Cloud. IEEE TSE, (submitted), (2014).

This submitted article is an extended version of [Frey et al. 2013] and provides details of the structure, functioning, and quality characteristics of CDOXplorer.

▻ [Brauer et al. 2014] P. C. Brauer, F. Fittkau, and W. Hasselbring. The aspect-oriented architecture of the CAPS framework for capturing, analyzing and archiving provenance data. In: *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW 2014)*. Springer, June 2014

CAPS aims to simplify the instrumentation process of the monitored applications. For instrumentation, an application needs to be uploaded on a web interface and CAPS integrates the required probes to capture provenance data during the execution of the monitored applications.

▻ [Frey et al. 2015] S. Frey, F. Fittkau, and W. Hasselbring. Optimizing the deployment of software in the cloud. In: *Proceedings of the Conference on Software Engineering & Management 2015*. Köllen Druck+Verlag, Mar. 2015

This is an abstract summarizing the results of [Frey et al. 2013].

▻ [Zirkelbach et al. 2015] C. Zirkelbach, W. Hasselbring, F. Fittkau, and L. Carr. Performance analysis of legacy Perl software via batch and interactive trace visualization. Technical report 1509. Department of Computer Science, Kiel University, Germany, Aug. 2015

This work includes the presentation of a monitoring component for Perl-based systems using Kieker[12] [van Hoorn et al. 2012]. Furthermore, a performance analysis using different visualizations is conducted for EPrints.

Co-Supervised Bachelor and Master Theses

▻ [Beye 2013] J. Beye. Technology evaluation for the communication between the monitoring and analysis component in Kieker. Bachelor's thesis. Kiel University, Sept. 2013

[12] http://kieker-monitoring.net

In his bachelor's thesis, Beye evaluated three communication technologies with respect to their performance in the context of transferring monitoring data. The evaluation resulted in identifying TCP as the fastest transportation technology for monitoring data. Therefore, we implemented our communication between monitoring and analysis component with a TCP connection.

▻ [Koppenhagen 2013] E. Koppenhagen. Evaluation von Elastizitätsstrategien in der Cloud im Hinblick auf optimale Ressourcennutzung. (in German). Bachelor's thesis. Kiel University, Sept. 2013

Koppenhagen investigated several cloud scaling approaches. For his evaluation, he implemented the three most promising scaling approaches and evaluated them with regard to performance and cost efficiency by means of a cloud simulator. These approaches can further enhance our elastic trace processing approach but the implementation remains as future work.

▻ [Kosche 2013] M. Kosche. Tracking user actions for the web-based front end of ExplorViz. Bachelor's thesis. Kiel University, Sept. 2013

The bachelor's thesis of Kosche evaluated several concepts of user action tracking for our web-based visualization aiming to record user actions during an experiment run. Since the aspect-oriented frameworks were not compatible with current Google Web Toolkit (GWT) versions, she integrated a manual approach. The recorded data of the experiment runs is part of each experimental package of our controlled experiments.

▻ [Matthiessen 2014] N. Matthiessen. Monitoring remote procedure calls – concepts and evaluation. Bachelor's thesis. Kiel University, Mar. 2014

Matthiessen developed a general Remote Procedure Call (RPC) monitoring approach in cooperation with the master's project at that time. This backpacking approach is still used in the current monitoring component. As proof-of-concept, he implemented and evaluated the concept for monitoring HTTP Servlet connections.

▻ [Stelzer 2014] P. Stelzer. Scalable and live trace processing in the cloud. Bachelor's thesis. Kiel University, Mar. 2014

In his bachelor's thesis, Stelzer conducted a preliminary version of our elasticity evaluation for the live processing of monitoring data using cloud computing and thus tested our approach with multiple static worker levels in his study. In his experiment, there were up to nine monitored JPetStore instances.

▻ [Weißenfels 2014] B. Weißenfels. Evaluation of trace reduction techniques for online trace visualization. Master's thesis. Kiel University, May 2014

Weißenfels investigated several trace reduction techniques for their efficiency and effectiveness. The best trace reduction technique was *trace summarization* which is therefore also used by our analysis component.

▻ [Barbie 2014] A. Barbie. Stable 3D city layout in ExplorViz. Bachelor's thesis. Kiel University, Sept. 2014

The bachelor's thesis of Barbie investigated a different layout algorithm for the 3D city metaphor and thus our application-level perspective. Targeting a stable and compact layout, he developed a layout algorithm using quad trees [Finkel and Bentley 1974]. His evaluations show that under certain circumstances the layout is stable. However, it uses a large amount of calculation time when the 3D city model grows and is not compact in most scenarios.

▻ [Barzel 2014] M. Barzel. Evaluation von Clustering-Verfahren von Klassen für hierarchische Visualisierung in ExplorViz. (in German). Bachelor's thesis. Kiel University, Sept. 2014

When no or inappropriate organizational units for classes are used by the monitored software, e.g., all classes are contained in one Java package, our interactive approach would not work as intended. Therefore, Barzel integrated a hierarchical clustering feature into ExplorViz basing on the relations between the classes and their names.

▻ [Finke 2014] S. Finke. Automatische Anleitung einer Versuchsperson während eines kontrollierten Experiments in ExplorViz. (in German). Master's thesis. Kiel University, Sept. 2014

Finke implemented the automatic tutorial mode into ExplorViz. Furthermore, she developed a configurable experimentation mode which shows generic linked question dialogs. Both modes were successfully used in our controlled experiments.

▻ [Gill 2015] J. Gill. Integration von Kapazitätsmanagement in ein Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015

In the context of the bachelor's project "Control Center Integration", Gill integrated the capacity management phase into ExplorViz. As foundation, we provided our capacity manager *CapMan* which is also used for scaling our trace analysis nodes in the cloud. Furthermore, he implemented the migration of applications from one node to a target node.

▻ [Mannstedt 2015] K. C. Mannstedt. Integration von Anomalienerkennung in einem Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015

Mannstedt integrated the anomaly detection phase into ExplorViz in the context of the bachelor's project "Control Center Integration". As basis for the anomaly detection, he used *OPAD* developed by Bielefeld [2012]. As evaluation, he compared different anomaly detection algorithms.

▻ [Michaelis 2015] J. Michaelis. Integration von Ursachenerkennung in ein Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015

Also in the context of the bachelor's project "Control Center Integration", Michaelis integrated the root cause detection phase into ExplorViz. He implemented four algorithms for root cause detection and evaluated them by a comparison.

▻ [Zirkelbach 2015] C. Zirkelbach. Performance monitoring of database operations. Master's thesis. Kiel University, July 2015

In his master's thesis, Zirkelbach developed an *AspectJ*[13] aspect for monitoring Java Database Connectivity (JDBC) calls and visualizes these information in a tool named *Kieker Trace Diagnosis*.[14] The gathered information include the execution time of the query, its return values, and the Structured Query Language (SQL) statement. Furthermore, he can process prepared statements and relate each execution of the it to the actual SQL statement. Utilizing the developed AspectJ aspect, we are able to provide database monitoring of JDBC calls.

▻ [Krause 2015] A. Krause. Erkundung von Softwarestädten mithilfe der virtuellen Realität. (in German, in progress). Bachelor's thesis. Kiel University, Sept. 2015

[13] https://eclipse.org/aspectj

[14] http://kieker-monitoring.net

Krause presents the implementation of our approach to utilize VR for exploring software visualizations which follow the city metaphor. The approach uses an Oculus Rift DK1 for virtually displaying the city model and utilizes a Microsoft Kinect v2 for gesture recognition. In this thesis, Krause implemented the gesture recognition as a C# program and developed the gestures in joint work with us. In his evaluation, he conducted eleven structured interviews to investigate the usability of the VR approach. In general, the participants liked the approach and see potential for using VR in program comprehension tasks.

▻ [Simolka 2015] T. Simolka. Live architecture conformance checking in ExplorViz. (in German, in progress). Bachelor's thesis. Kiel University, Sept. 2015

In the context of his bachelor's thesis, Simolka implemented our live conformance checking approach presented in [Fittkau et al. 2014a]. The target architecture is modeled in a separate perspective and than the generated landscape model is checked against this target model. If differences between the two models are found, these are visualized by a color-coding.

1.4 Document Structure

This thesis consists of the following five parts:

▻ Part I describes the foundations for this thesis.

  ▻ Chapter 2 details the concepts behind application monitoring required for the thesis.

  ▻ Chapter 3 introduces the field of software visualization and the underlying concepts and existing visualizations with a focus on related topics to our thesis.

- Part II presents our approach for providing a live trace visualization of large software landscapes.
    - Chapter 4 defines the research questions and our research methods.
    - Chapter 5 describes our method, named ExplorViz, for enabling live trace visualization of large software landscapes.
    - Chapter 6 details how we monitor applications and RPCs. Furthermore, our analysis and trace processing approach is presented.
    - Chapter 7 provides a description of our software landscape meta-model and how we generate a landscape model from the analyzed traces.
    - Chapter 8 introduces our live trace visualization approach and the developed two perspectives, i.e., the landscape-level perspective and the application-level perspective.
- Part III shows the conducted evaluations for our approach.
    - Chapter 9 describes the implementation of our approach.
    - Chapter 10 provides three evaluations related to our monitoring and analysis approach.
    - Chapter 11 presents four controlled experiments for evaluating the developed visualization approach.
    - Chapter 12 contains an evaluation where we evaluated the internal quality of our implementation by letting external developers extend the implementation by a prescribed control center concept.
    - Chapter 13 discusses related work.
- Part IV concludes the thesis.
    - Chapter 14 summarizes the thesis and its results.
    - Chapter 15 presents future work.

- ▻ Part V contains the appendix.
    - ▻ Chapter A presents the debriefing questionnaire used in the extensibility evaluation.

Finally, the back matter contains a list of figures, a list of tables, a list of listings, a list of used acronyms, and the bibliography.

Part I

Foundations

2

Application Monitoring

This chapter presents the terms and concepts of application monitoring which build the foundation for our monitoring and analysis approach (SC2). Section 2.1 describes the instrumentation of applications. Afterwards, the trace concept for our monitoring approach is presented in Section 2.2. Finally, the differences between a live and offline analysis are shown in Section 2.3.

Listing 2.1. Manual Instrumentation Example in Java

```
1  public class ExampleClass {
2    public void exampleMethod() {
3      System.out.println("exampleMethod() start");
4
5      // do internal logic
6
7      System.out.println("exampleMethod() end");
8    }
9  }
```

2.1 Instrumentation

To monitor an application, we need to instrument it. Therefore, *probes* are inserted into the application to gather the data required for our visualization. The IEEE defines instrumentation as "devices or instructions installed or inserted into hardware or software to monitor the operation of a system or component." [ISO/IEC/IEEE 24765 2010]

The instrumentation can be conducted manually and automatically. Both approaches are described in the following.

2.1.1 Manual Instrumentation

Manual instrumentation [Bulej 2007] consists of inserting probes for gathering the data in source code locations of interest by hand. A prerequisite for manual instrumentation is the availability of the source code. Otherwise, the software engineer would need to change the machine or byte code which would be cumbersome. Furthermore, manually instrumenting large applications requires a large amount of work such that manual instrumentation should only be applied when few methods need to be instrumented [Parsons 2007].

Listing 2.1 shows an example for manually instrumenting a Java source code. We instrument the method by adding the statements in Line 3 and Line 7. Every time the instrumented method is executed, it outputs the

message “exampleMethod() start” before conducting its internal logic. The message “exampleMethod() end” is written to the output when successfully finishing its internal logic. While this instrumentation only reveals whether the method was called and finished without an exception, the logged information can be enriched by, for example, the execution duration. We refer to the collection of the gathered data of one probe as a *monitoring record* or shortly as a *record*.

2.1.2 Automatic Instrumentation

In the manual instrumentation approach, potentially every method has to be changed by hand since monitoring is concerned about potentially every method in the application. Kiczales et al. [1997] introduced the paradigm of Aspect-Oriented Programming (AOP) to facilitate the development work of such cross-cutting concerns. Therefore, AOP can be used to provide automatic instrumentation of the application.

An implementation of AOP for Java is AspectJ.[1] It introduced four universal terms [Kiczales et al. 2001] which got adopted later by other AOP frameworks. These terms are:

Join Point: A join point is one point in the execution of the application. For example, when a method is executed, the join point is passed to the advice and provides information about the method execution such as the name of the executed method and its parameters.

Pointcut: For selecting the methods that should be woven, pointcuts are used which are defined by a predicate. For instance, *execution(* *(..))* matches all methods that are executed.

Advice: The actual additional code that should be woven is specified in an advice. Furthermore, it defines at which location its code is inserted, i.e., before, after, or around the affected join points.

Aspect: An aspect encapsulates the cross-cutting concern. The corresponding join points, pointcuts, and advices are defined in it.

[1] `https://eclipse.org/aspectj`

Listing 2.2. Around Aspect for Instrumenting Java Source Code in AspectJ

```
1  @Aspect
2  public class MonitoringAspect {
3    @Pointcut("execution(* *(..))")
4    public void monitoredOperation() {}
5
6    @Around("monitoredOperation()")
7    public final Object instrument(final ProceedingJoinPoint p)
8          throws Throwable {
9      System.out.println(p.getSignature() + " start");
10
11     final Object retVal;
12     try {
13       retVal = p.proceed();
14     } catch (final Throwable th) {
15       System.out.println(p.getSignature() + " exception");
16       throw th;
17     }
18
19     System.out.println(p.getSignature() + " end");
20     return retVal;
21   }
22 }
```

The aspects need to be integrated into the execution of the program. Therefore, the so-called weaving of the aspects with the Java bytecode can be conducted at compile time or at the loading of a class definition into the Java Virtual Machine (JVM). For load-time weaving, the monitored program has to be started with an additional parameter to specify the AspectJ Weaver as the Java Agent. Thus, the AspectJ Weaver can run before the actual program logic is executed and weave the class definitions that get loaded.

An example aspect for monitoring all method executions in a Java application is shown in Listing 2.2. The aspect, pointcut, and around advise definition are done with annotations. The pointcut predicate *execution(* *(..))*

in Line 3 means that all methods of the monitored application should be woven with the around advise defined in Line 6 to 21. An around advise combines before and after advices but can also react on exceptions and can directly save internal variables between the code segments. Therefore, the actual method's logic code needs to be called manually by calling `proceed()` (Line 13) on the join point. Furthermore, when outputting the start, exception, and end events, we access the signature of the method that is executed. The differences to the manual instrumentation in Listing 2.1 are minimal but this aspect suffices to instrument a whole application instead of one single method.

2.2 Trace Concept

Each inserted probe generates monitoring records when a monitored method is executed. These records contain a trace identifier in our used trace concept. With this trace identifier, they can be associated to one execution trace which is a special form of a general trace [Jain et al. 1991]. For reasons of simplicity, when we talk about *traces* in this thesis, we refer to *execution traces*.

In Figure 2.1, an example execution trace for the trace concept used by us is shown. Method A calls Method B, Method B calls Method C, et cetera. The events that form the monitoring records are annotated behind the name of each method. A *Before Event* is generated at the entry of the execution of a method and an *After Event* is passed to the analysis when exiting the method. In general, two types of After Events exist, i.e., normal exits and when an exception is thrown (cf. Listing 2.2). This trace concept is also used in, for example, Kieker [van Hoorn et al. 2009b; Waller 2014].

Another trace concept is generating records only at the exit of an executed method. However, generating two events for each method execution provides the benefit that the analysis can determine when an event is missing and a trace has ended.

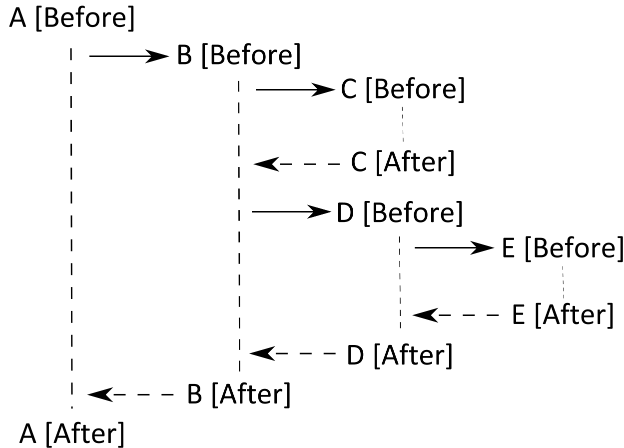


Figure 2.1. Example execution trace and generated monitoring records

2.3 Live versus Offline Analysis

There are two classes of dynamic analyses [Dwyer et al. 2007], i.e., offline and online analyses. When monitoring records are gathered and the program terminates before these records are analyzed, the analysis is called *offline analysis*. In contrast, an *online analysis* conducts the analysis of the gathered monitoring records during the runtime of the monitored program.

This categorization implies that offline analyses require a storage for monitoring records and therefore they are limited by the available storage space. If traces with millions of events – as happened in [Yang et al. 2006] – need to be processed, an online analysis can spare the software engineer from searching for an adequate storage. The online analysis just consumes and processes the monitoring records on-the-fly.

A disadvantage of online analysis is that the gathered data can not be replayed – only the analysis results are available after the run. Therefore, if the configuration of an analysis needs to be changed – which is possible in an offline analysis –, the program needs to be executed again.

For our approach, we choose an online analysis approach due to the huge amount of monitoring records produced in a large software landscape. Since online trace visualization could also be interpreted as the visualization of online traces, we use the terminology of *live* trace visualization for our approach and this thesis.

3

Software Visualization

Software visualization is *"the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software."* [Price et al. 1993]

In this chapter, we describe related concepts and previous work of other researchers that build the foundation for our approach and visualization. The visualization pipeline, which provides the basis for our approach (SC1), is presented in Section 3.1. Section 3.2 describes the software city metaphor and approaches which implement or extend this metaphor providing the foundation for our application-level perspective, and the search for other display and interaction concepts (SC3). Then, closely related trace visualization approaches are presented in Section 3.3. Finally, RPCs visualization approaches are shown in Section 3.4 which build the foundation for our landscape-level perspective.

Previous Publications

Parts of this chapter are already published in the following work:

1. [Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. IEEE, May 2015

3.1 Visualization Pipeline

In general, visualization is an – often adjustable – mapping of raw data to a visual form perceivable for humans [Card et al. 1999]. The steps in this mapping are visualized in the visualization pipeline of Card et al. [1999] (see Figure 3.1). It starts with the Raw Data which is transformed to Data Tables. Data Tables are often more structured than Raw Data and easier to map to Visual Structures. For example, Raw Data is often aggregated and extended with metadata to create structured Data Tables.

After this transformation, the Data Tables are mapped to Visual Structures. Visual Structures are, for example, box plots, color-codings, or charts. Visual Structures have a spatial substrate with marks and graphical properties that should target an expressive encoding of the data. The mapping is expressive when the data and only the data is expressed by the Visual Structure. For example, the Visual Structure should not imply a relationship between data points when this relationship is incorrect. Furthermore, Data Tables can often be mapped in several ways to Visual Structures. Therefore, the representation should also be as effective as possible. A color-coding of the sine wave is less effective than a line chart representing the values, for instance.

The Visual Structures are transformed to actual Views for the human perceiver. There are three common view transformations: location probes, viewpoint controls, and distortions. Location probes use locations in Visual Structures to provide additional related data, e.g., a tooltip showing additional on demand information. Viewpoint controls utilize affine transformations such as panning and zooming to change the viewpoint. The visual transformation of distortion combines overview and detail in a single Visual Structure. For instance, a fisheye view provides detailed information about the focused center while the outer area still contains an overview of the data.

Since the perceiver of the visualization has specific tasks in mind, she needs to interact with the steps in the visualization pipeline to achieve a supporting visualization of the task at hand. Therefore, she is able to change the Data Transformations applied when transforming Raw Data to Data Tables by, e.g., changing the metrics used to produce the Data

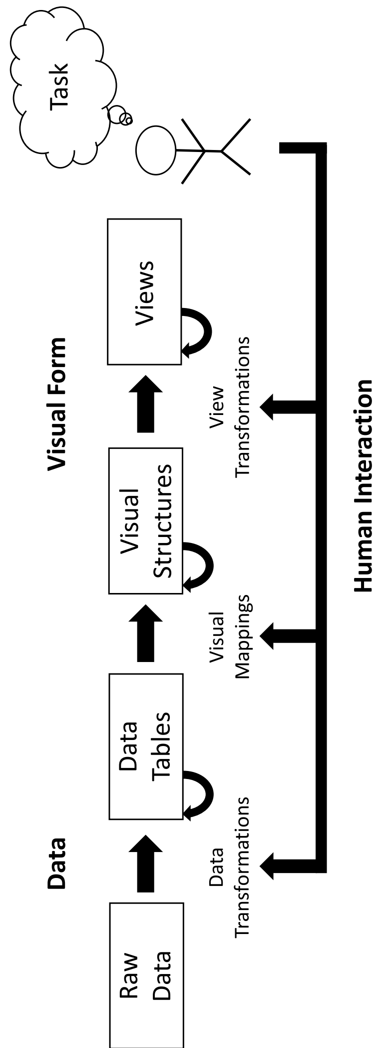


Figure 3.1. Visualization pipeline based on [Card et al. 1999]

Tables. Furthermore, she can influence the Visual Mappings used when mapping the Data Tables to Visual Structures. For example, showing the data in a line chart instead of a box plot. The last interaction possibility is conducting View Transformations such as panning or rotating the shown Visual Structures to change the viewpoint.

3.2 Software City Metaphor

In this section, we describe visualizations following or extending the software city metaphor [Knight and Munro 1999] in chronological order. These visualizations build the foundation for our application-level perspective and our search for new display and interaction possibilities for the software city metaphor.

3.2.1 Software World

In the year 1999, Knight and Munro [1999, 2000] represented a software system based on the model of a real city (see Figure 3.2). The whole software system maps to one *world*. A *country* represents the directory structure of the system. In the context of *Software World*, these districts map to Java packages. A *city* is one file from the software system and each class contained in the file is represented by a *district*. The *districts* are made up of *buildings* which correspond to the methods of the class. Districts can also contained special urban items such as gardens, parks, or monuments which are used to represent special attributes of classes.

Buildings can have several details attached depending on the attributes of the represented method. The height of a building maps to the lines of code of the method. The number of doors of the building maps to the parameter count and the number of windows maps to the declared variable count. Whether the method is public or private influences the color of the building. Furthermore, a small sign – like a house number sign in the real world – is placed on the building to represent the method name.

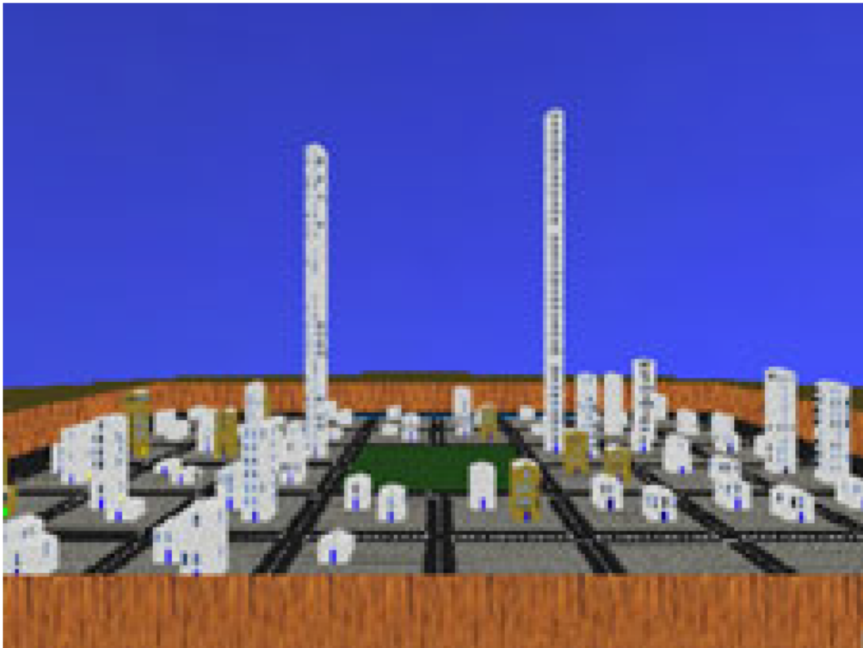


Figure 3.2. Software World (taken from [Anslow et al. 2006])

3.2.2 3D City

With the presentation of the city metaphor, Knight and Munro [1999] wanted to show that moving away from abstract software visualizations is possible. Panas et al. [2003] further followed this direction by creating *3D City* with the aim to create a realistic city representation of a software system. One *city* is displayed in Figure 3.3. A city represents one Java package. The whole 3D City can include multiple cities which are connected by *streets*, if the packages depend on each other, or by *water* if one package only depends on the other. *Clouds* can hide cities that are out of the current focus. *Buildings* in a city represent components (mainly Java classes). To increase the realism, the authors add trees, streets, and street lamps to the scene. The quality

Figure 3.3. 3D City (taken from [Panas et al. 2003])

of the implementation is visualized by the appearance of a building. For example, old and collapsed buildings show that the source code should be refactored.

In addition to those static information, 3D City can also visualize dynamic information. *Cars* move according to one execution trace. They leave traces in different colors for easier identification of the origin and destination component. If there is dense traffic, a large amount of communication is conducted between the components. The speed and type of a car maps to the performance and priority of a method call. Occurring exceptions in the software system are visualized by colliding cars resulting in an explosion.

In addition, 3D City visualizes business information on top of this representation. For example, execution hot spots are indicated by a fire surrounding the corresponding building.

3.2.3 CodeCity

Wettel and Lanza [2007] argue that the previous two approaches do not sufficiently support the program comprehension process of large software systems. According to the authors, the visual mapping in Software World is not well chosen since it leads to cities with thousands of districts for large

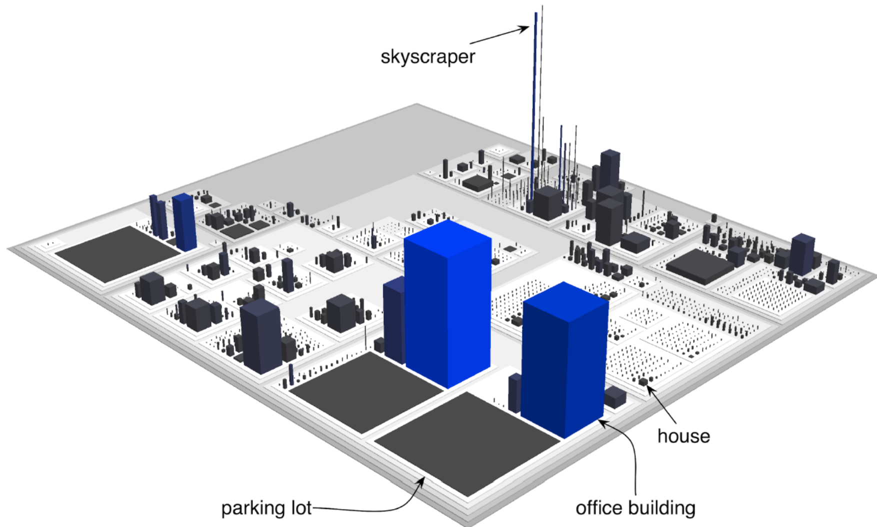


Figure 3.4. ArgoUML visualized in CodeCity with annotated building archetypes (taken from [Wettel 2010])

software systems. Furthermore, they reason that Software World should exploit the package information for the layout of the districts which leads to more locality in the visualization. Moreover, Wettel and Lanza argue that the presented ideas of 3D City lack a sufficient interaction concept.

To overcome the aforementioned shortcomings, Wettel and Lanza [2007] developed *CodeCity*.[1] Figure 3.4 presents the CodeCity visualization of ArgoUML.[2] In CodeCity, *districts* represent the packages and classes are displayed by *buildings*. The hierarchy of the packages is visualized by stacking the districts. The number of attributes (NOA) of a class maps to the height of the corresponding building and the number of methods maps to its width. This mapping forms different archetypes of buildings. Classes

[1] `http://codecity.inf.usi.ch`
[2] `http://argouml.tigris.org`

with many methods but a small number of attributes result in tall and thin skyscrapers. On the opposite, classes with few methods and a high number of attributes result in flat, large platforms named “parking lots”. Beneath those two extremes, there are also “office buildings” representing classes with high functionality and a normal amount of attributes, and “houses” with a low amount of functionality and a low amount of attributes. Due to these archetypes, the distribution of system intelligence is perceived at the first glance on the visualization. Additionally, colors are used to represent special further metrics such as the probability of a god class, for instance.

The efficiency and effectiveness of CodeCity has been evaluated in a controlled experiment in [Wettel et al. 2011]. Wettel et al. compared the usage of CodeCity to using the state of the practice, i.e., Eclipse and Excel, for solving program comprehension tasks. The experiment resulted in a statistically significant increase in task correctness (+24 %) and decreased time spent (-12 %) when using CodeCity for solving the tasks.

3.2.4 Vizz3D

In *Vizz3D* [Panas et al. 2007], the user is able to switch between layout algorithms and used metaphors at runtime. One metaphor available in Vizz3D is the city metaphor (see Figure 3.5). In their visualization, *buildings* represent methods. The *textures* on the buildings are determined by different metrics. For instance, the represented method of a blue colored building has more than 200 lines of code. *Cities* shown as blue plates represent source files. Green *landscapes* map to directories of the software system. The water and sky only provide more realism and have no specific semantics. The connection between the buildings can represent method calls, inheritance relations, or contains relations.

3.2.5 EvoSpaces

EvoSpaces [Dugerdil and Alam 2008] provides two views, i.e., a *day* and a *night* view. The day view represents the static information about a software system. Classes and files map to *buildings* and packages map to *districts*. As in CodeCity, the districts are nested. The height and texture of the

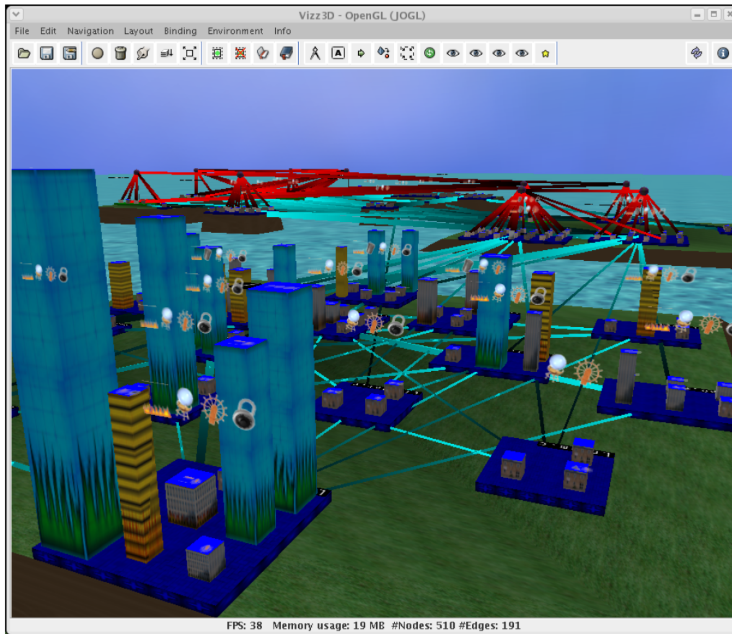


Figure 3.5. Vizz3D visualizing a C++ program (taken from [Panas et al. 2007])

buildings is varied according to metrics. The authors use three types of buildings represented by the texture and each type has three different height categories. Relationships between classes are visualized by pipes.

The night view of EvoSpaces is shown in Figure 3.6. The semantics of buildings and districts is the same as in the day view. However, the solid pipes represent the execution trace. Furthermore, the occurrence of the classes in the trace get color-coded and mapped onto the buildings.

3.2.6 DyVis

Wulf [2010] developed *DyVis* which is a combination of CodeCity and TraceCrawler [Wysseier 2005; Greevy et al. 2006]. Figure 3.7 presents an

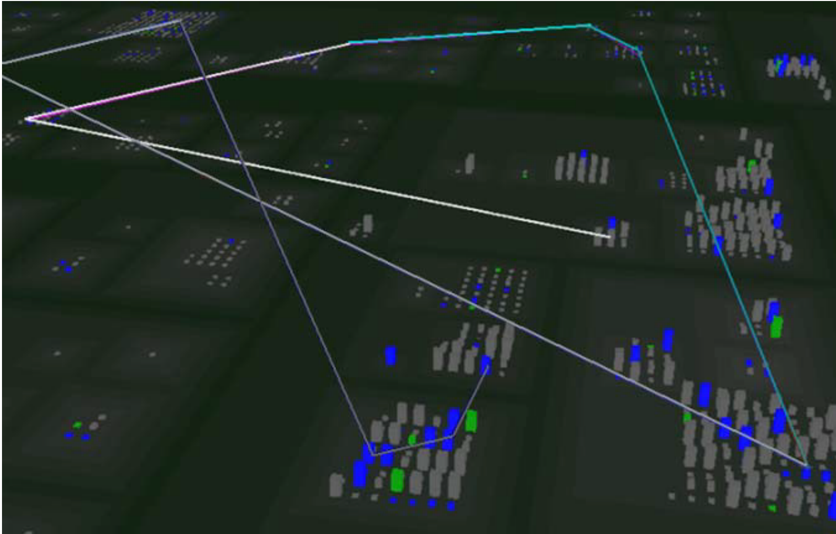


Figure 3.6. Night view of EvoSpaces (taken from [Dugerdil and Alam 2008])

execution trace of JPetStore in DyVis. An upfront static analysis provides information about the packages and classes to DyVis. *Districts* represent the Java packages and classes are visualized by *buildings* similar to CodeCity. However, the buildings have different *floors* similar to TraceCrawler. Each floor represents one instance of the class. Since Java also supports static classes, the first floor is the static class instance. *Streets* run from one floor of a building to another floor of a building (or itself). These represent one method call from one object to another object. Furthermore, the caller and callee are highlighted in green and the street is colored according to the portion of the method's runtime to the overall runtime of the trace.

DyVis also features a tree-based view of the method calls which happened in the trace (displayed in the right part of Figure 3.7). Furthermore, information about the current visualized method call is displayed in the bottom. At the top, buttons for an automatic playback of the trace are present.

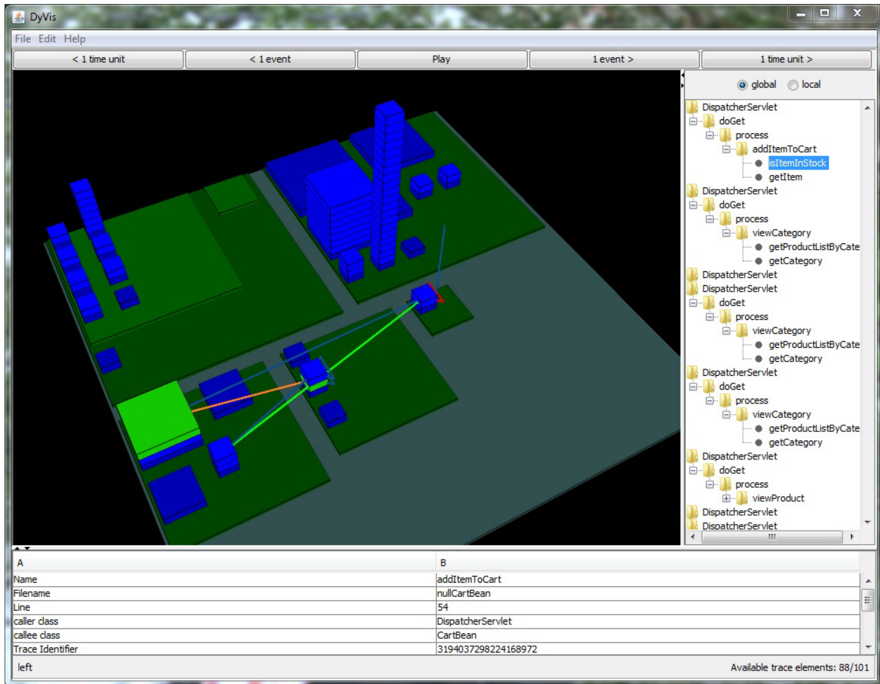


Figure 3.7. An execution trace of the web application JPetStore visualized in DyVis (taken from [Wulf 2010])

3.2.7 Evo-Streets Layout

Evo-Streets[3] [Steinbrückner and Lewerentz 2010] is a stable layout for visualizing evolving software systems using the city metaphor. It is implemented in the visualization tool CrocoCosmos [Lewerentz and Noack 2004]. In Figure 3.8, the JDK6 is visualized using the Evo-Streets layout. Each *street* represents one subsystem and branching streets show contained subsystems. The global system level is visualized by a main street where the subsystems branch off. The attached colored blocks represent modules, i.e., classes

[3] `http://software-cities.org`

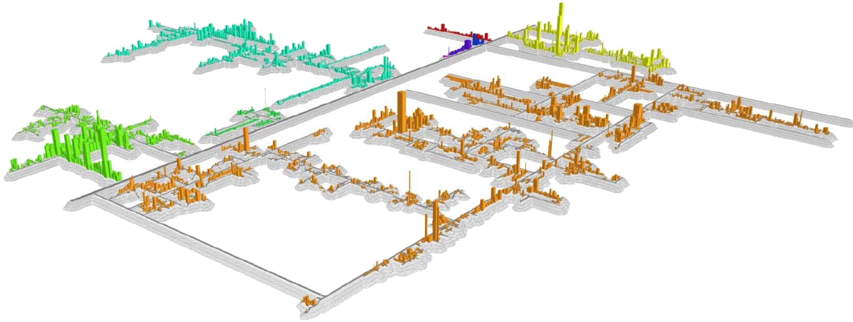


Figure 3.8. Visualization of the Java Development Kit 6 using the Evo-Streets layout (taken from [Steinbrückner and Lewerentz 2010])

in the context of Java. The height of the buildings map to the coupling between the represented class and the rest of the system.

To enable a stable layout, rules for inserting new elements and modifying old elements have to be defined. New elements are attached to the end of a street of the corresponding package by extending the street, for example. In addition to the height of the buildings, the third dimension is used to present an elevation of the streets and buildings. The creation time of the element is mapped to this elevation. Therefore, older elements get a higher elevation in the visualization.

3.2.8 Software Map

Bohnet and Döllner [2011] present an approach – they call *software maps* – to visualize up-to-date information of the internal quality of a software system by using the city metaphor. A software map represents source code files in hierarchical modules similar to CodeCity. Figure 3.9 shows a software map visualizing the 3D creation suite Blender.[4] In contrast to CodeCity, software maps intend to provide managers with information about the internal

[4]`https://www.blender.org`

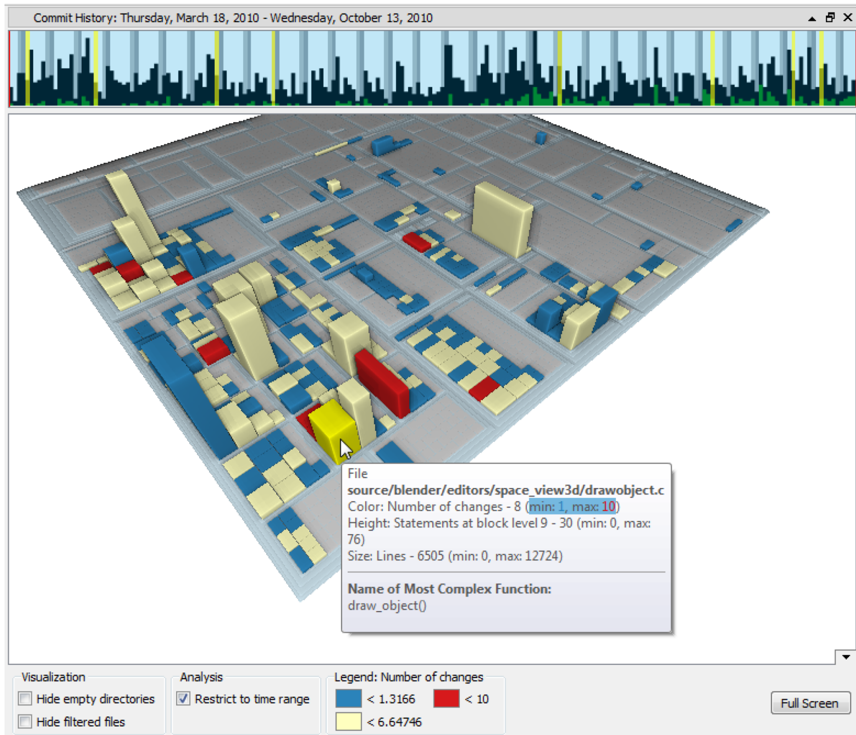


Figure 3.9. Software map for Blender (taken from [Bohnet and Döllner 2011])

quality of the software and about where and which developer modifies source code. Therefore, the metrics visualized by the height, ground area, and color are customizable. In comparison to a sole table representation of metrics, the authors argue that whether a value is good or bad depends on the relatively to the other values which is better visualized by software maps. Limberger et al. [2013] implemented the approach of software maps for web browsers.

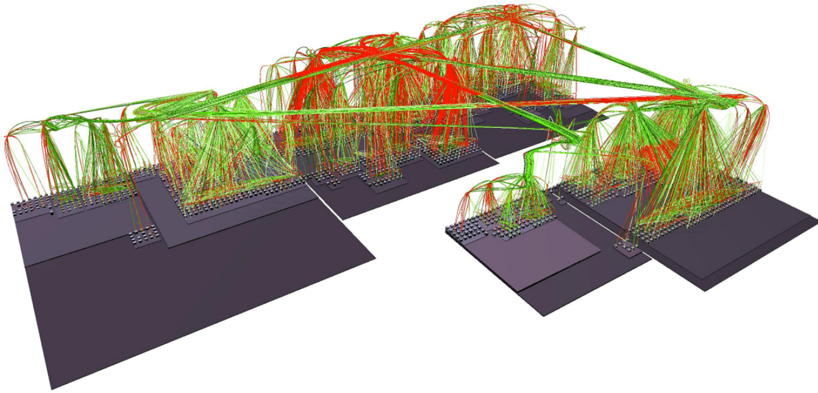


Figure 3.10. Visualization of an execution of jEdit using 3D-HEB (taken from [Caserta et al. 2011])

3.2.9 3D Hierarchical Edge Bundles Extension

Caserta et al. [2011] uses the hierarchical edge bundling technique [Holten 2006] to represent relations in the 3D city metaphor. They call the presented technique *3D Hierarchical Edge Bundles* (3D-HEB). Figure 3.10 shows the resulting visualization of jEdit.[5] The direction of the communication of the bundles is color-coded, i.e., green represents the source and red the destination.

3.2.10 SynchroVis

SynchroVis [Waller et al. 2013] developed by Döhring [2012] extends DyVis by visualizing the concurrent behavior of an application. Figure 3.11 depicts a deadlock of the dining philosophers problem visualized by SynchroVis. For visualizing the concurrent behavior, it provides special buildings. The building with the lock symbol represents the free (uncolored) and locked (colored in the color of the trace holding the lock) semaphors/monitors in

[5] `http://www.jedit.org`

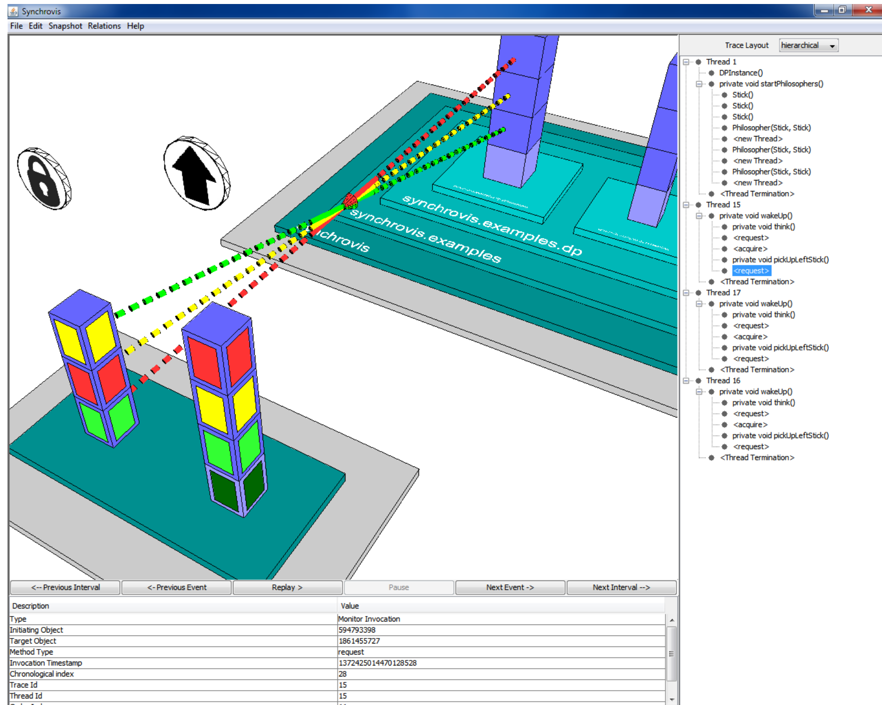


Figure 3.11. Deadlock of the dining philosophers problem visualized by SynchroVis (taken from [Waller et al. 2013])

the system. The beneath *thread building* shows the active threads. *Streets* in SynchroVis also represent the method calls but the color stands for which trace executes the method call.

3.2.11 SkyscrapAR

SykscrapAR [Souza et al. 2012] is an augmented reality approach employing the city metaphor to visualize software evolution. The user can interact with a physical marker platform (see Figure 3.12) in an intuitive way while the

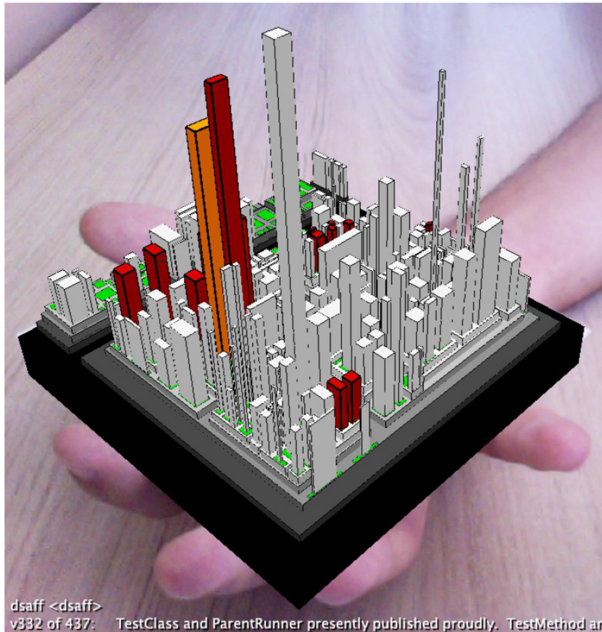


Figure 3.12. JUnit framework in SykscrapAR (taken from [Souza et al. 2012])

actual visualization can be seen on the monitor. In general, the semantics follow the semantics of CodeCity. However, since SykscrapAR visualizes the evolution of a software system, Souza et al. added green lots for each class. These lots are only completely filled by the building when it reaches the maximum size in the visualized revision of the software.

3.2.12 Manhattan

Lanza et al. [2013] developed Manhattan to support the team activity comprehension. It bases on the city metaphor building up on a reimplemented CodeCity and provides a real-time visualization integrated in the Eclipse IDE. The visualization depicts team activity information and

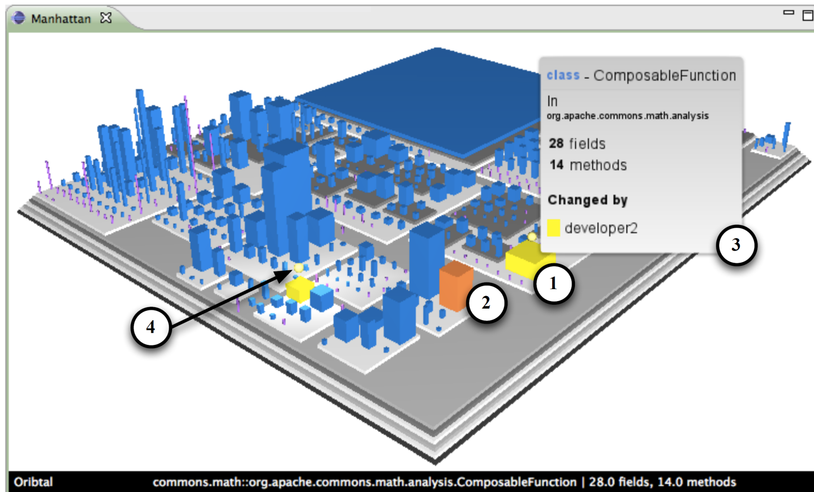


Figure 3.13. Apache Commons Math in Manhattan (taken from [Lanza et al. 2013])

provides developers with information about potential conflicting code. Figure 3.13 shows an example visualization of the Apache Commons Math library.[6] Potential conflicts are represented by a sphere above a class (point 4 in Figure 3.13). Classes modified by a developer are visualized by becoming yellow (point 1). Removed classes do not disappear instantly but are colored orange (point 2). More information, such as which developer frequently changes the focused class, can be fetched on demand through a tooltip (point 3).

3.2.13 CodeMetropolis

CodeMetropolis [Balogh and Beszédes 2013] utilizes Minecraft[7] to visualize a software system and enables online user collaboration. Its main idea is to use high quality graphics provided by current game engines for

[6] `http://commons.apache.org/proper/commons-math`
[7] `https://minecraft.net`

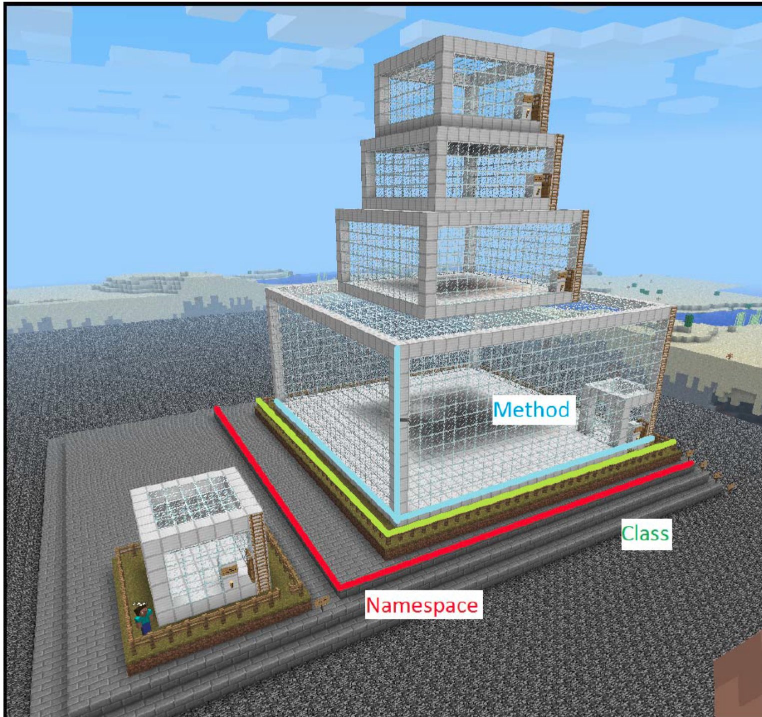


Figure 3.14. Semantics of CodeMetropolis (taken from [Balogh and Beszédes 2013])

data visualization. Therefore, the visualization uses graphical primitives provided by Minecraft and for collaboration the already present multi-player server is used.

The semantics of the visualization are depicted in Figure 3.14. Namespaces/packages are represented by stackable *stone plates*. A class is visualized by green *grass blocks*. On top of the grass, several *floors* represent the contained methods of the class. The size of a floor is defined by normalized code metrics. McCabe Cyclomatic Complexity (McCC) is used for the width and length. Logical Lines of Code (lLOC) is mapped to the height of the

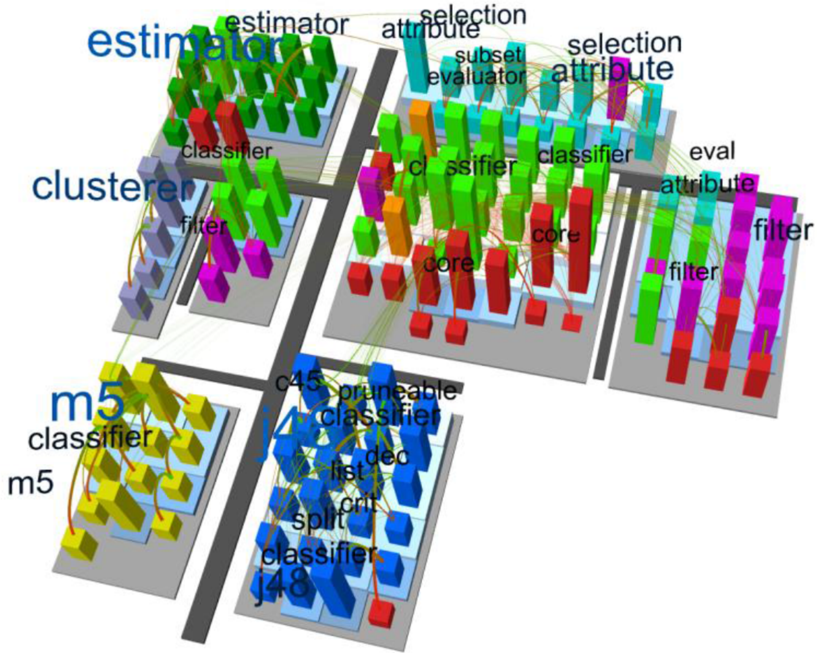


Figure 3.15. SArF map of Weka 3.0 (taken from [Kobayashi et al. 2013])

floor. The player can climb to each floor and walk into it. Similar to Software World, the number of windows and doors visualizes the parameter count. In addition, torches on the wall indicate if a method is tested.

3.2.14 SArF Map

Software Architecture Finder (SArF Map) [Kobayashi et al. 2013] visualizes software clustering results and architectural layers from feature and layer viewpoints using the city metaphor. The features are extracted by their clustering approach [Kobayashi et al. 2012]. Figure 3.15 shows an example

visualization of the features of the data mining tool Weka.[8] The *buildings* represent classes and blocks group these classes when the classes implement one feature. The height of the buildings maps to the lines of code of the corresponding class. The keywords in the feature are used to label the buildings. The relevance between features is represented by connecting streets. In addition to streets, there are colored curves that visualize the dependencies between classes. The belonging package of a class is encoded by the color of a building.

3.3 Trace Visualization

This section describes existing trace visualization approaches in chronological order. Since many approaches exist, we focus on the visualization approaches that are closely related to our visualization. For an overview of trace visualization approaches, we refer to the surveys of Bennett et al. [2008], and Hamou-Lhadj and Lethbridge [2004]. We start by presenting 2D-based approaches and show 3D-based approaches afterwards.

3.3.1 2D-Based

In the following, we describe closely related 2D-based trace visualizations.

Jinsight

Jinsight [De Pauw et al. 2001; 2002] visualizes method executions, class fields, and parameters at runtime of a program. The execution view of Jinsight is shown in Figure 3.16. The bars represent method executions and the execution time window runs from top to bottom. Since gathering field and parameters value can result in a large data amount, the user is able to activate and deactivate the gathering of those information using the visualization during the runtime of the program under study.

[8] `http://www.cs.waikato.ac.nz/ml/weka`

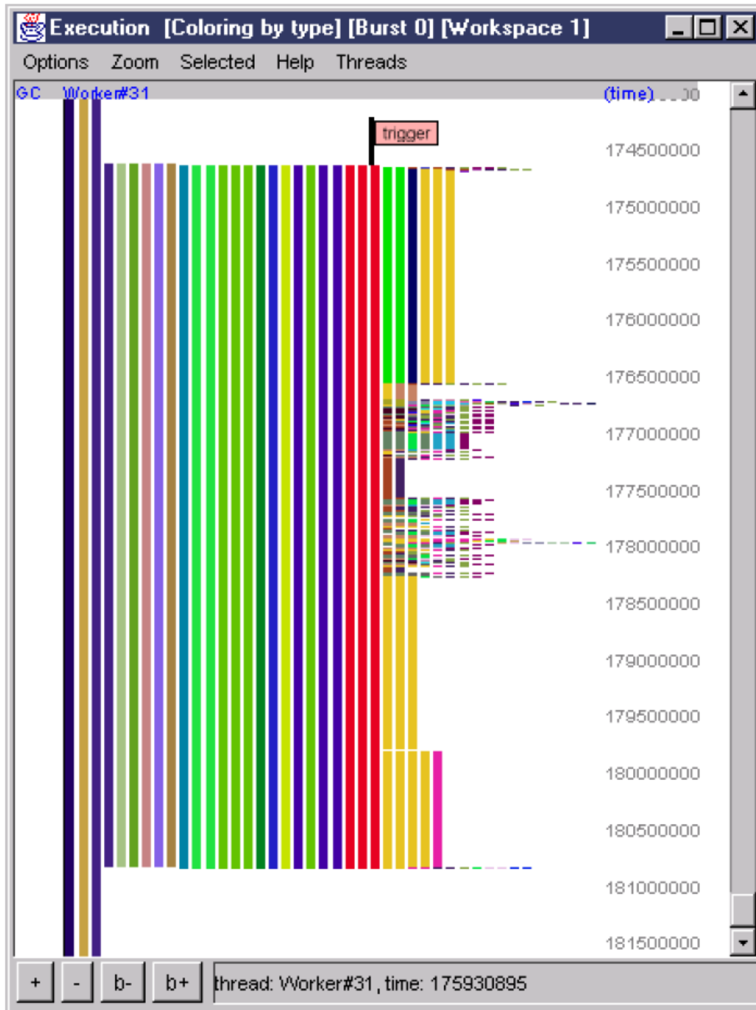


Figure 3.16. Execution view of Jinsight (taken from [De Pauw et al. 2001])

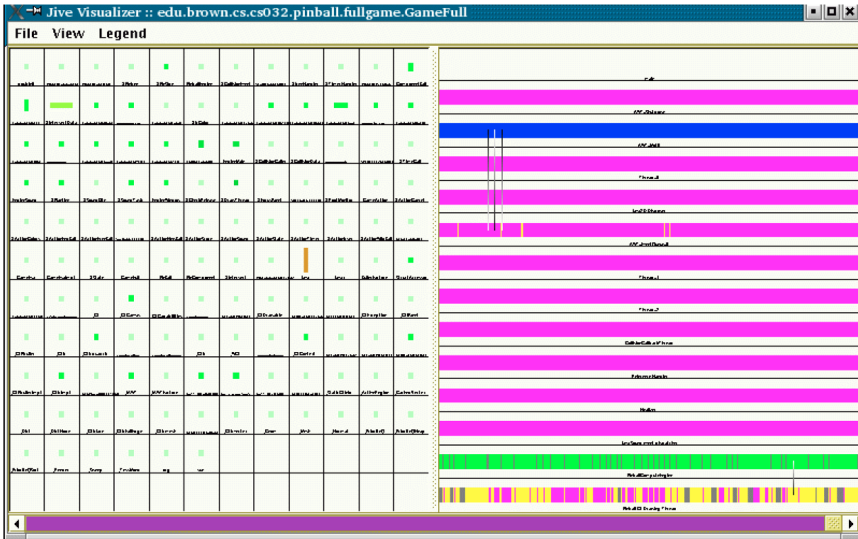


Figure 3.17. JIVE (taken from [Reiss and Tarvo 2012])

JIVE and JOVE

JIVE [Reiss 2003] and *JOVE* [Reiss and Renieris 2005] visualize Java programs during runtime. The visualization of JIVE is divided into two panes (see Figure 3.17). The left part depicts class and package usage information. Five values of the running software are encoded in this visualization. The height of the rectangle indicates the number of conducted method calls. The width of the rectangle shows the amount of object allocations conducted by the methods of the class. The hue represents the allocated instances of the class. The rectangle saturation indicates whether the class was used or not. Finally, its brightness shows the number of synchronization events. On the right part of Figure 3.17, thread data is shown. The hue in the bar represents the thread state at each time interval.

Figure 3.18 shows the visualization of JOVE. Vertical regions, representing files of the system under study, make up the main part of the

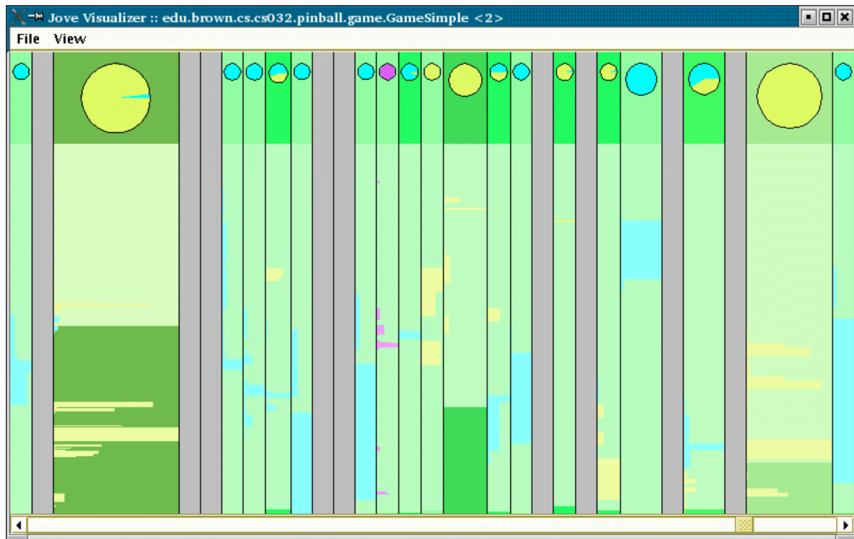


Figure 3.18. JOVE (taken from [Reiss and Tarvo 2012])

visualization. Each vertical region is divided in two parts. The top of the region shows thread information and the bottom shows block information. In their terms, a block is a segment of straight-line code with no internal branches. At the top of each region, a pie-chart shows the time spent of each thread in the class. The width of vertical regions maps to the ratio of executed instructions in the program run.

Extravis

EXTRAVIS was developed by Cornelissen et al. [2007]. In a controlled experiment, Cornelissen et al. [2009] showed that the availability of EXTRAVIS in addition to Eclipse positively influences the effectiveness and efficiency in program comprehension tasks. EXTRAVIS focuses on the visualization of one large execution trace. For this purpose, it utilizes two interactive, linked

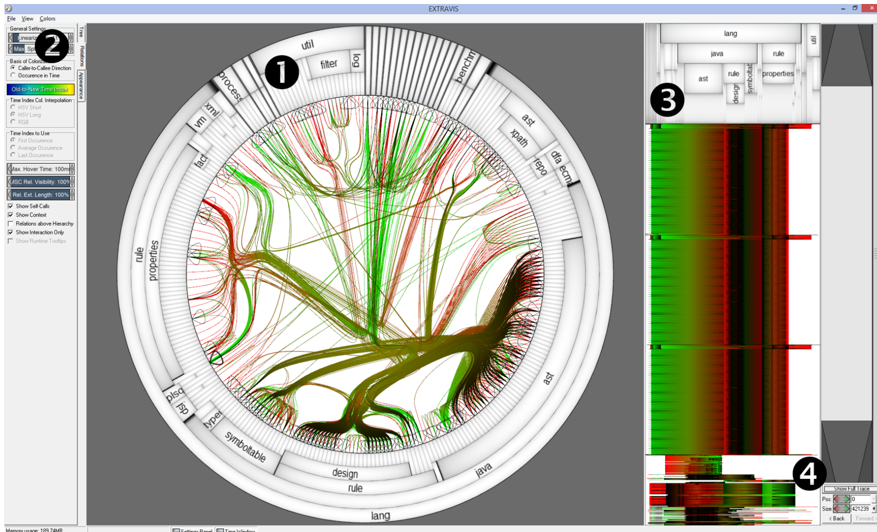


Figure 3.19. A recorded execution trace of PMD visualized in EXTRAVIS

views: the circular bundle view and the massive sequence view. Those two views are described in the following.

Circular Bundle View The centered visualization of EXTRAVIS is the circular bundle view (❶ in Figure 3.19). The classes are arranged at the inner circle. Due to the high number of classes in the analyzed software system PMD[9] (279 visualized classes), the names of the classes are only visible through tooltips on the respective entity. The outer circles represent the packages of PMD. In the inner field of the circle, the method calls between classes is represented by lines. The names of the method calls are visible by hovering over these lines. EXTRAVIS utilizes color coding for the direction of the visualized communication. In its default setting, green represents outgoing calls and red expresses incoming calls. The width of each line corresponds to the call frequency of the method.

[9] http://pmd.sourceforge.net

EXTRAVIS follows a hierarchical, bottom-up strategy [Shneiderman 1980], i.e., all packages show their internal details at the beginning. It is possible to close packages and thus hide the contained classes to gain further insights into the global structure of the visualized software system. Furthermore, edge bundling provides hints about strong relationships between packages. The communication between two classes can be filtered by marking both classes. This selection highlights the method calls in the massive sequence view. In addition to displaying the communication direction, EXTRAVIS enables switching to a chronological trace analysis (❷) by changing the semantics of the line colors. In this mode, color is globally used for representing the occurrence in time of the method call in the trace. In its default setting, dark blue represents the oldest method call and yellow corresponds to the newest method call.

Massive Sequence View The massive sequence view (❸) visualizes the method calls over time similar to a compressed UML sequence diagram. On top, the classes and packages are displayed and their method calls are listed beneath. The direction of the communication is color coded as in the circular bundle view. The massive sequence view enables to filter the method calls according to a time window from point *A* in a trace to point *B* in a trace. This filtering restricts the massive sequence view and the circular bundle view to only contain method calls within the selected time window. A further feature of EXTRAVIS is a history of the previously selected time windows (❹).

ViewFusion

ViewFusion [Trümper et al. 2012] combines ("fuses") the structure and activity information in one view. They use treemaps[10] for showing the structure and icicle plots [Kruskal and Landwehr 1983] for the trace visualization. Figure 3.20 shows Chromium[11] visualized by their ViewFusion tool. The icicle plots are displayed at the top and provide details on demand when hovering near and over the icicles with the mouse. The colors represent the

[10] http://www.cs.umd.edu/hcil/treemap

[11] https://www.chromium.org

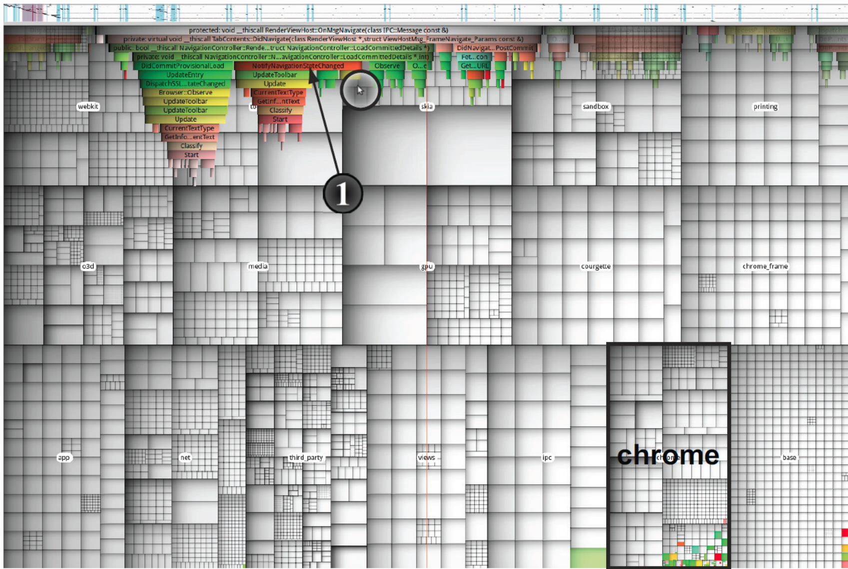


Figure 3.20. Chromium visualized in ViewFusion (taken from [Trümper et al. 2012])

maximum occurring stack depth of each method. Therefore, the method displayed at ❶ also contributes to low-level functionality of Chromium. In the structure view, the colored rectangle are part of the current visualized execution trace. Most of them are in the *chrome* package. Hence, the executed features mainly use functionality of this package.

UML Sequence Diagrams

A large part of trace visualization approaches are similar to UML sequence diagrams [Briand et al. 2004]. Since we use a metaphor-based approach, we only briefly describe five approaches using UML in the following.

SCED [Systä 2000] can be used to model the run-time of Java applications and provides its own state diagram which is similar to UML. **JAVAVIS** developed by Oechsle and Schmitt [2002] uses object and sequence dia-

grams to visualize the execution of a Java program. Malnati et al. [2008] developed **JThreadSpy** which visualizes concurrent behavior on the basis of additions to UML sequence diagrams. Voets [2008] developed the tool **JRET**. It visualizes traces generated by Java applications using UML sequence diagrams and can also visualize loops in the execution. **Kieker Analysis** [Ehlers 2011] represents the method calls and dependencies in a Java application similar to UML diagrams.

3.3.2 3D-Based

In the following, closely related 3D-based trace visualization approaches are described. The approaches 3D City, EvoSpaces, DyVis, and SynchroVis are also part of this category but were already described in Section 3.2.

Call Graph Analyzer

Bohnet and Döllner [2006] present an approach where they combine four feature-related views in the tool *Call Graph Analyzer*. Figure 3.21 shows the Graph Exploration View visualizing the web browser Mozilla Firefox.[12] The focused function is displayed as a disc (left upper part of the figure) and the neighborhood represented by arrow tops (predecessor functions) and arrow bottoms (successor functions). The method call is shown by an asymmetrical arc where the peak point of the arc is shifted to the target of the call. In addition to this view, the authors provide a Source Code View in a linked window, a Function Search View visualizing all functions in a list, and a Bookmark View where a user can bookmark functions of interest.

TraceCrawler

TraceCrawler developed by Wysseier [2005] and Greevy et al. [2006] is an extension to CodeCrawler [Lanza 2003]. It visualizes the object allocations and the chronological sequence of the method executions through a playback mode. The semantics of the visualization are shown Figure 3.22. For each class, a tower divided into floors exists. Each floor represents one

[12] https://www.mozilla.org/en-US/firefox

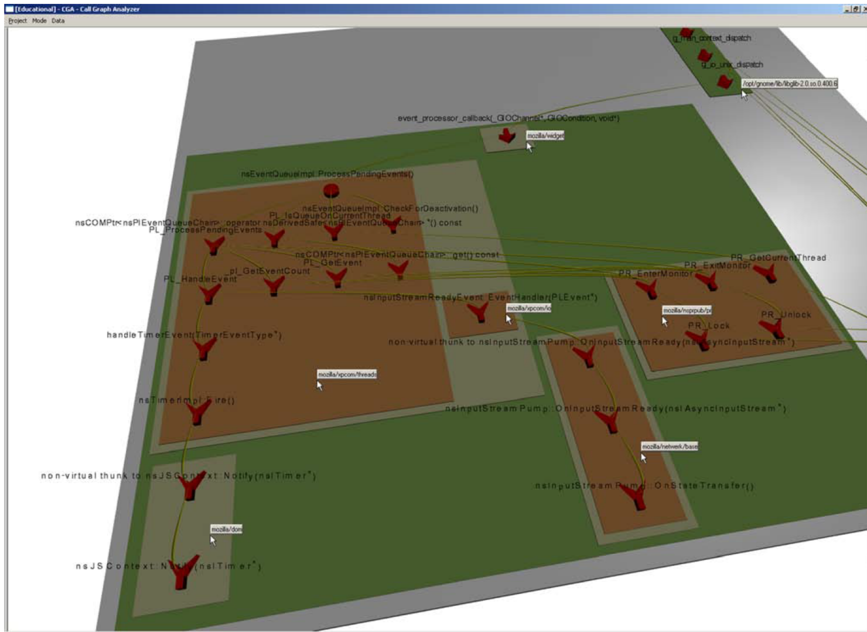


Figure 3.21. Call Graph Analyzer (taken from [Bohnet and Döllner 2006])

instance of the class. The ground floor is a special entity since it represents the class by itself. Lines between the ground floors show the inheritance relations and lines between boxes represent the method calls. During a playback, active instances and active method calls are highlighted. The width, length, and color of the boxes are used to represent metrics about each instance. The applied metrics are configurable. Source code of each class can be viewed on demand.

TraceCrawler features two views. The first view is the Dynamic Feature-Trace View. Within this view, the user is able to step through the execution trace in linkage with the visualization. Therefore, she can view the details about the execution. Figure 3.23 shows the TraceCrawler visualization of a trace of SmallWiki [Ducasse et al. 2005] in the second view, i.e., Instance

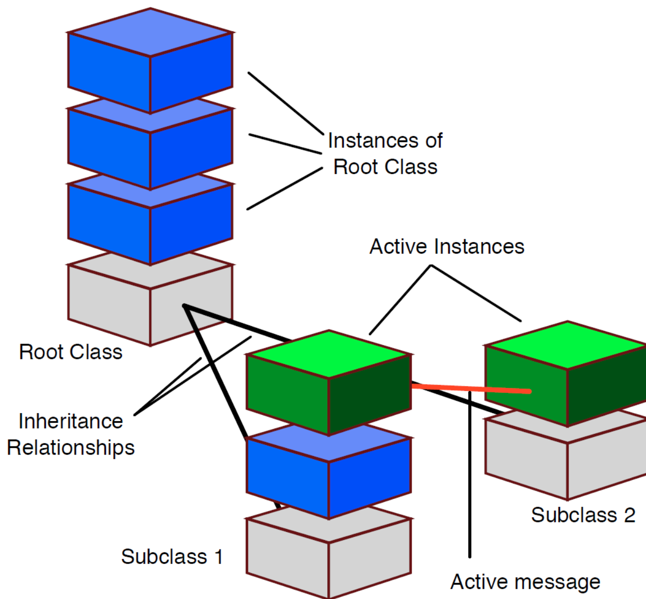


Figure 3.22. Schematic view of the visualization used in TraceCrawler (taken from [Greevy et al. 2006])

Collaboration View. This view provides an overview of the whole trace. Three feature hotspots can be identified, i.e., the Response, Login, and PageView feature.

3.4 Remote-Procedure-Call Visualization

In this section, we present related RPCs visualization approaches.

3.4.1 Web Services Navigator

Web Services Navigator [De Pauw et al. 2006] provides 2D graph visualizations of the communication of web services. It provides three views for

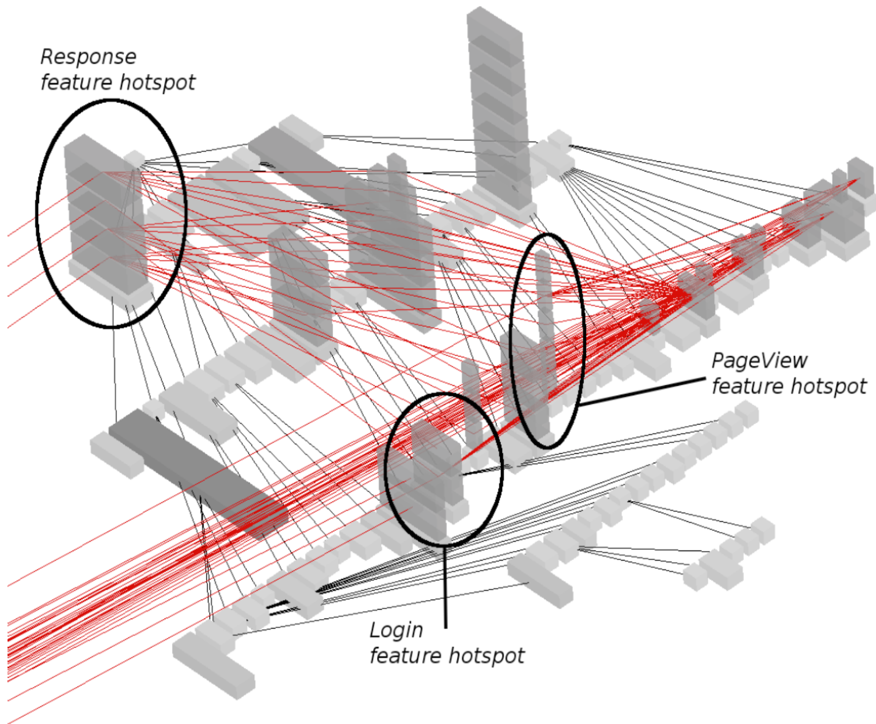


Figure 3.23. Instance Collaboration View of the login feature in SmallWiki visualized in TraceCrawler (taken from [Greevy et al. 2006])

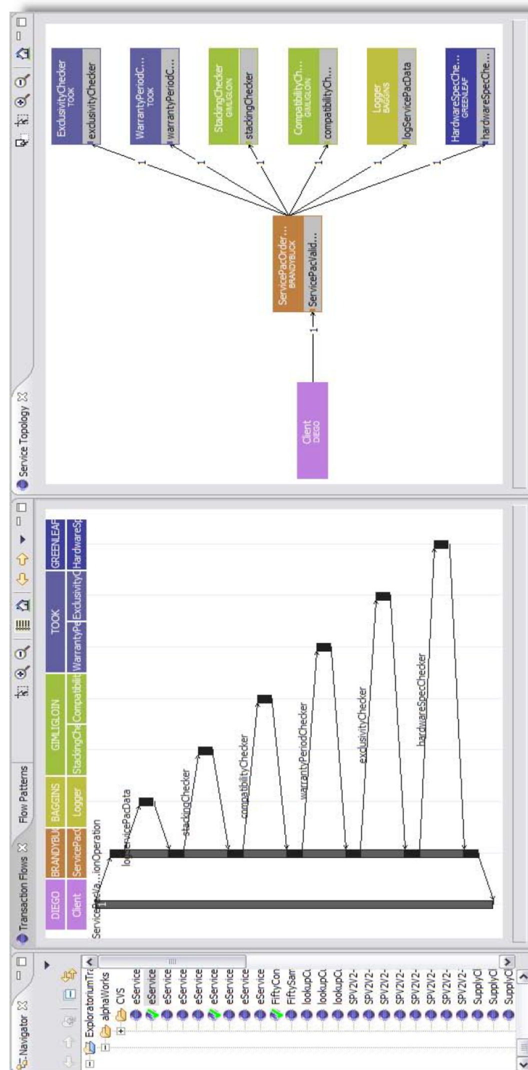


Figure 3.24. Web Services Navigator: trace selection window, Transaction Flows view, and Service Topology view (taken from [De Pauw et al. 2006])

showing the communication and their underlying patterns. In Figure 3.24, the Transaction Flows and Service Topology views are presented. The Transaction Flows view shows each web service call in a UML sequence diagram fashion. Since this view does not scale well, the Service Topology view visualizes each web service as one box and represents the direction flow by a flow-based left-to-right layout. However, the view fails to present the chronological order of the calls. Therefore, De Pauw et al. [2006] provide a third view, i.e., the Flow Patterns view. This view shows the access pattern of the web services successively in one window.

3.4.2 Streamsight

Streamsight [De Pauw et al. 2008] visualizes cooperating distributed components of streaming applications. Therefore, it shows the data flow and not the control flow. Since a very similar visualization could visualize the control flow, we also describe Streamsight in this section. Figure 3.25 depicts an example visualization. The boxes map to processing elements of a streaming application. There are three types of processing elements: sources, analytics, and sinks. The *Generators* in the figure are sources, i.e., no input ports. The *Annotators*, for instance, are analytics processing elements having in- and output ports. Finally, there are sinks that inhibit no output port such as *ConsumerF*. The coloring of each processing element represents the host it is running on. Input ports are visualized on the left side of a processing element and output ports are shown on the right side of a processing element. Therefore, the direction of the communication is also encoded by the layout. Ports can have two states – open and closed – whether or not they produce/consume data. Tooltips provide details about the host name and the processed data.

3.4.3 RanCorr

RanCorr [Marwede et al. 2009] visualizes the dependencies between applications in a root cause analysis scenario. Figure 3.26 shows an example visualization in a JPetStore case study. Each block represents one virtual machine and the “transparent” block shows also the internal components of

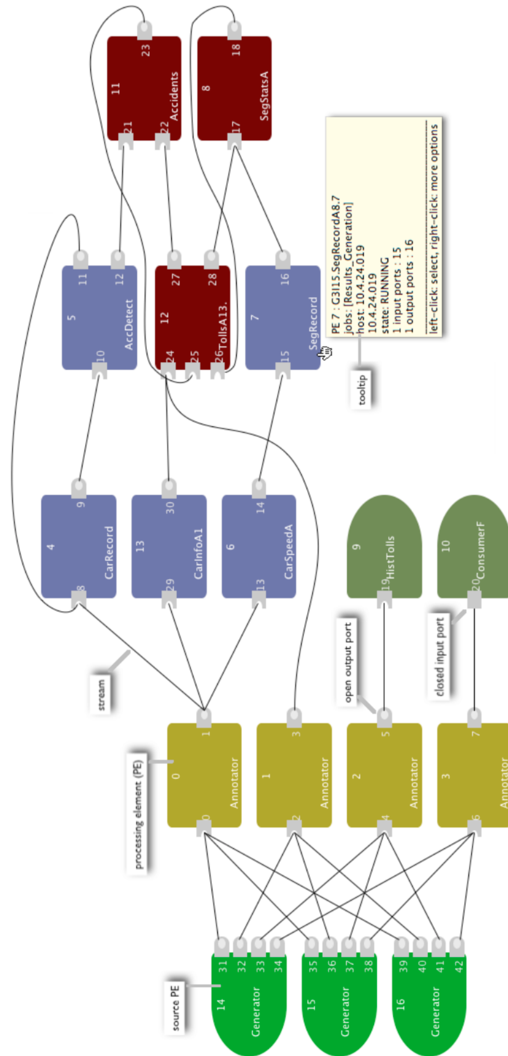


Figure 3.25. Streamsight (taken from [De Pauw et al. 2008])

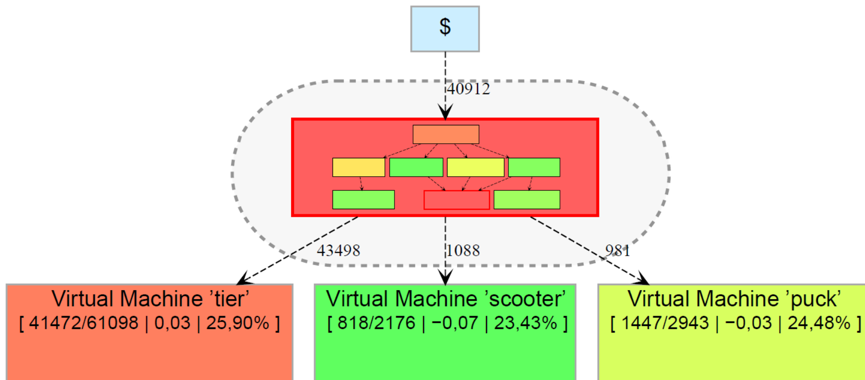


Figure 3.26. Root cause rating view of RanCorr (taken from [Marwede et al. 2009])

JPetStore running on a virtual machine. The dependency lines are labeled with the total count of conducted RPCs. The root cause probability of each virtual machine is visualized by a color range starting at green (unlikely for being the root cause) to red (very likely for being the root cause).

3.4.4 APM Tools

Several commercial APM tools for monitoring and visualizing a software landscape exist: *AppDynamics*,[13] *ExtraHop*,[14] or *New Relic*[15], for instance. Most tools show servers and the running applications on them by using boxes. The applications are often connected by lines with arrows that have the response times or a "health status" beneath them.

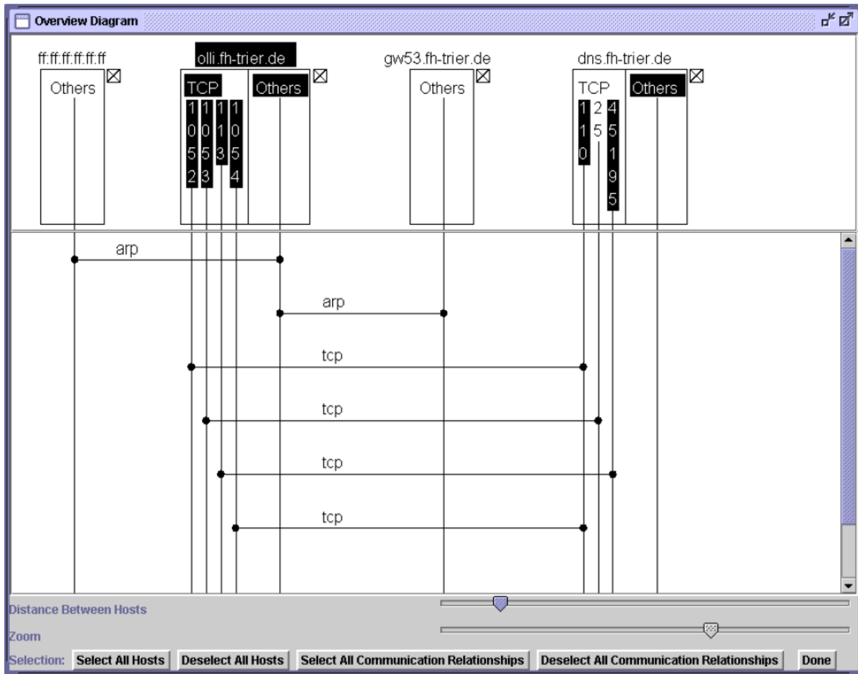


Figure 3.27. VisuSniff (taken from [Oechsle et al. 2002])

3.4.5 UML Sequence Diagrams

There are multiple approaches that use UML sequence diagrams to visualize RPCs. *VisuSniff* [Oechsle et al. 2002] – shown in Figure 3.27 – uses UML sequence diagrams to represent the low level data packages sent and received by an application for educational purposes. Briand et al. [2006] utilize UML sequence diagrams to visualize RPCs by adding a hostname to the object representation [Briand et al. 2004; Briand et al. 2005].

[13] `https://www.appdynamics.com`
[14] `https://www.extrahop.com`
[15] `http://newrelic.com`

Part II

The ExplorViz Approach

4

Research Design

In this chapter, we detail our research design. We start by describing the employed research methods in Section 4.1. Then, our research questions and research plan are presented in Section 4.2.

Previous Publications

Parts of this chapter are already published in the following work:

1. [Fittkau 2013] F. Fittkau. Live trace visualization for system and program comprehension in large software landscapes. Technical report 1310. Department of Computer Science, Kiel University, Germany, Nov. 2013

4.1 Research Methods

This section presents our employed research methods within this thesis. A survey of general research methods in software engineering is described in [Wohlin et al. 2012] and [Juristo and Moreno 2010]. In addition, we follow further guidelines for empirical software engineering [Kitchenham et al. 2002; Jedlitschka and Pfahl 2005; Di Lucca and Di Penta 2006; Di Penta et al. 2007; Sensalire et al. 2009].

The following research methods are employed within this thesis:

- ▻ *Literature Review:* A literature review is a systematic method to review approaches and concepts, which have already been developed by other researchers.
- ▻ *Proof-of-Concept Implementation*: A proof-of-concept implementation evaluates the technical feasibility of an approach. Furthermore, it is often a prerequisite for other research methods such as lab experiments or controlled experiments.
- ▻ *Lab Experiment:* In our terms, a lab experiment investigates technical aspects of a solution in a controlled environment.
- ▻ *Controlled Experiment:* In a controlled experiment [Basili et al. 1999; Basili 2007], the test subjects are typically split into two groups, i.e., the experimental and the control group. While the control group does not receive the treatment, the experimental group does. Therefore, it is likely that a potential different outcome results from this treatment.
- ▻ *Case Study:* A case study investigates an effect in a given context and in a specific time span. Therefore, it does not sample over the variables as it is done in an experiment but analyzes a specific variable configuration which makes it difficult to generalize the results.
- ▻ *Structured Interview:* In a structured interview, the interviewee is asked a predefined list of questions. Therefore, the questions and their order are fixed when conducting multiple interviews.

Furthermore, we employ the Goal, Question, Metric (GQM) approach [Van Solingen and Berghout 1999; Van Solingen et al. 2002]. It provides a structured approach to define a set of goals and related questions. In addition, for each question, metrics are selected to provide a measurable fulfillment condition for the goals.

In this thesis, we employ the following research process: After reviewing the literature and thus the body of knowledge, we first define a GQM plan (see Section 4.2). Then, we create approaches aiming to achieve the defined goals. For each approach, we provide a proof-of-concept implementation which is used to evaluate the approach. The employed methods in the evaluations are experiments, case studies, and structured interviews.

The main method, we employed, are experiments, i.e., the two subtypes lab experiments and controlled experiments. Lab experiments are used to evaluate technical aspects, such as providing a low overhead, scalability, or elasticity, of an approach if no human interaction is involved. When targeting human perception or cognitive processes, we use controlled experiments to investigate if the designed approach provides advantages compared to already existing state-of-the-art approaches.

Notably, when performing experiments, replications [Shull et al. 2002] become important to improve the external validity. Therefore, we conduct one replication for an evaluation of our application-level perspective.

4.2 Research Questions and Research Plan

We envision live trace visualization as a solution to support program and system comprehension in a large software landscape. Therefore, our main goal (G1) is providing a live trace visualization for large software landscapes. Furthermore, a monitoring and analysis approach capable of logging and processing the huge amount of conducted method calls in large software landscapes is the second goal (G2). The last goal (G3) is investigating alternative display and interaction concepts for the software city metaphor beyond classical 2D displays and pointing devices. These three goals are described according to the GQM approach in the following and map to our scientific contributions (SC1 – SC3) described in Section 1.2.

G1: Visualize the Execution Traces in a Large Software Landscape for Program and System Comprehension during its Runtime

In a large software landscape, millions of method calls can occur in one second. Therefore, several thousands of traces have to be visualized in a quickly perceivable way to support the program and system comprehension process. Thus, the first research question (Q1) concerning G1 is: *How to visualize the large amount of traces to support system and program comprehension?* Different visual representation concepts (M1) need to be considered and chosen from. Furthermore, useful features (M2) and interaction concepts (M3) supporting in the comprehension process needs to be identified. The resulting live trace visualization approach is described in Chapter 8.

To enable a visualization, typically the gathered data is represented by a structured model. Therefore, a landscape meta-model representing the data of the monitored software landscape is desirable. We formulate the second research question (Q2) as: *Which information must be provided by a meta-model for enabling live trace visualization?*

Keeping all information about every monitored method call is cumbersome since the monitoring might produce several gigabytes per second. Therefore, information of interest needs to be identified (M4). Furthermore, to provide visual scalability, we require different hierarchy levels (M5). Our landscape meta-model is presented in Chapter 7.

To be an advancement, the designed trace visualization should provide advantages over current related approaches. Therefore, we formulate the third research question (Q3): *Does our live trace visualization provide advantages for supporting the program and system comprehension process compared to other approaches?* The metrics for this question are an increased efficiency (M6) and/or an increased effectiveness (M7) when solving comprehension tasks. To evaluate this research question, we conduct three controlled experiments presented in Chapter 11.

In summary, the GQM plan for our first goal is:

G1: Visualize the execution traces in a large software landscape for program and system comprehension during its runtime.

- ▻ Q1: How to visualize the large amount of traces to support system and program comprehension?
  - ▻ M1: different visual representation concepts
  - ▻ M2: useful features
  - ▻ M3: interaction concepts
- ▻ Q2: Which information must be provided by a meta-model for enabling live trace visualization?
  - ▻ M4: completeness of the information of interest
  - ▻ M5: number of hierarchy levels
- ▻ Q3: Does our live trace visualization provide advantages for supporting the program and system comprehension process compared to other approaches?
  - ▻ M6: increased efficiency
  - ▻ M7: increased effectiveness

G2: Monitor Applications and Analyze the Resulting Execution Traces in Large Software Landscapes

In a large software landscape, several applications have to be monitored. Since the monitoring impacts the productive systems, the monitoring approach should impose only a low overhead. Therefore, the fourth research question (Q4) is: *How to monitor the existing applications?* Metrics for the monitoring component are the implied monitoring overhead (M8) and the maximal throughput of monitored method calls per second (M9). Therefore, we developed a low overhead monitoring approach which is described in Chapter 6.

An additional challenge in large software landscapes is the large amount of conducted method calls and thus also the resulting monitored data is large. For this purpose, we formulate the fifth research question (Q5): *How to analyze the huge amount of generated monitoring data?* Important metrics for the analysis approach are the scalability (M10) and elasticity (M11) with respect to the number of monitored applications. Furthermore, it should be able to process the data in a live fashion (M12). Chapter 6 presents our developed analysis approach where we feature dynamically inserted worker levels to preprocess the large amount of incoming monitored data.

Chapter 10 describes three lab experiments investigating the overhead of our monitoring component, and the scalability and elasticity of our live analysis approach.

In summary, the GQM plan for G2 is:

G2: Monitor applications and analyze the resulting execution traces in large software landscapes.

- ▻ Q4: How to monitor the existing applications?
  - ▻ M8: implied monitoring overhead
  - ▻ M9: throughput of monitored method calls per second
- ▻ Q5: How to analyze the huge amount of generated monitoring data?
  - ▻ M10: scalability with respect to monitored applications
  - ▻ M11: elasticity with respect to monitored applications
  - ▻ M12: live processing capability

G3: Find Alternative Display and Interaction Concepts for the Software City Metaphor

A further goal of the thesis is to find alternative display and interaction concepts for the software city metaphor. Therefore, we formulate the sixth research question (Q6) as: *Which alternative forms of displaying a model following the software city metaphor exist?*

The metric for this research question are the concrete alternative presentation forms (M13). We investigated two display forms, i.e., VR using a Head-Mounted Display (HMD) described in Section 8.5 and 3D-printing a physical form of a model following the software city metaphor presented in Section 8.6.

Since new display forms often benefit from new interaction concepts, we formulate the seventh research question (Q7) as: *Which alternative interaction concepts for a model following the software city metaphor exist?*

The metric for this question are the concrete alternative interaction concepts. Our VR approach uses gesture-based interaction and the physical model uses natural interaction capabilities such as touching and taking the model into the hands. The interaction concepts are described in Section 8.5 and in Section 8.6.

For the evaluation of the approaches, the research question (Q8) sounds: *Does an alternative display and interaction concept provide advantages for supporting the program comprehension process compared to classical 2D displays and pointing devices?*

As in Q3, the metrics for this question are an increased efficiency (M6) and/or increased effectiveness (M7). We conducted a controlled experiment for investigating this circumstance for the physical models (see Section 11.3). The VR approach was evaluated by Krause [2015] in structured interviews.

In summary, the GQM plan for the third goal is:

G3: Find alternative display and interaction concepts for the software city metaphor.

- ▻ Q6: Which alternative forms of displaying a model following the software city metaphor exist?
  - ▻ M13: alternative presentation forms
- ▻ Q7: Which alternative interaction concepts for a model following the software city metaphor exist?
  - ▻ M14: alternative interaction concepts

- ▻ Q8: Does an alternative display and interaction concept provide advantages for supporting the program comprehension process compared to classical 2D displays and pointing devices?
  - ▻ M6: increased efficiency
  - ▻ M7: increased effectiveness

5

The ExplorViz Method

In this chapter, we present our ExplorViz method to tackle the challenge of live trace visualization of software landscapes. Furthermore, the approach describes how we aim to achieve the goals presented in the last chapter. Our approach includes five activities which start by monitoring the existing applications and ends with the visualization of the software landscape.

We start by describing the fundamental approach. Afterwards, the assumptions and limitations of our approach and how they can be overcome are presented.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

2. [Fittkau 2013] F. Fittkau. Live trace visualization for system and program comprehension in large software landscapes. Technical report 1310. Department of Computer Science, Kiel University, Germany, Nov. 2013

3. [Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015

4. [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

5.1 Fundamental Approach

Figure 5.1 illustrates an overview of the activities in our ExplorViz approach. In the following, the activities A1 to A5 are briefly described. Chapter 6 to Chapter 8 present each activity in detail.

A1 – Monitoring

The existing applications in the software landscape are monitored. Besides monitoring method calls in each application, RPCs are also monitored. This provides information on the communication between applications. Due to the possibly large amount of generated monitoring data, the monitoring data might have to be written to different analysis nodes.

Another approach for ensuring that the amount of generated monitoring data can be processed is deactivating the currently not required probes [Ehlers et al. 2011]. However, this approach is infeasible for ExplorViz due to the time shift feature used to provide a history over the landscape visualization which requires that all probes are active.

The result of this activity is a stream of monitoring records for the executed method.

A2 – Preprocessing

Most servers cannot process the huge amount of incoming monitoring records which is typical for large software landscapes. Therefore, we use worker nodes to preprocess the monitoring records utilizing, for example, cloud computing. Since this still might lead to an overutilization of the master server, a dynamic deployment of worker nodes and worker levels is used. To preprocess the monitoring data, the records are first consolidated into traces. Afterwards, we employ a trace reduction technique [Cornelissen et al. 2008], i.e., trace summarization, on each worker node.

If the workload imposed by the monitored data can be processed on a single analysis node, this activity is omitted due to a lower number of running instances and thus lower occurring costs.

The result of this activity are preprocessed traces.

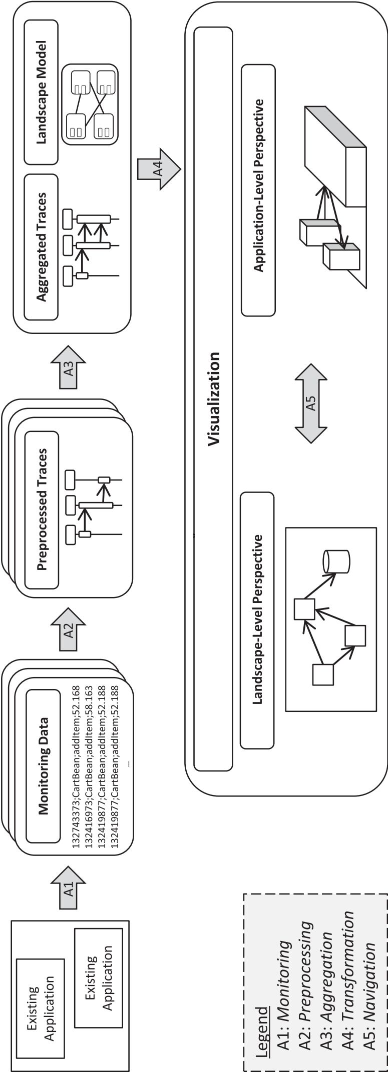


Figure 5.1. Overview of the fundamental activities in our ExplorViz approach

A3 – Aggregation

To enable a global view of the software landscape, the distributed, preprocessed traces are collected and aggregated on a single node by again using the trace consolidation and reduction technique of Activity A2. In addition, a landscape model representing the software landscape is created or updated. This model keeps track of the entities, e. g., applications and communication paths, that were discovered during runtime. Without the landscape model, these entities would not be visible in the following landscape visualizations.

As a result, this activity provides the created or updated landscape model.

A4 – Transformation

This activity consists of a transformation from the landscape model into a visualization model. The resulting visualization model also includes information required for visualizing each entity, for example, its color, size, or position.

The result of this activity is the visualization model used for the actual visualization.

A5 – Navigation

Our live trace visualization includes two perspectives – one for the landscape level and one for the application level. The user can always navigate from one perspective to the other. We decided to provide two different perspectives because we targeted that the user can clearly differentiate between landscape and application level.

5.2 Assumptions

After giving an overview of our approach, we now discuss the assumptions and limitation of it and point out what could be done to avoid them.

For our approach, we assume that each application is instrumented with our monitoring solution. In practice, often the applications already generate monitoring data of some kind, e.g., structured or unstructured log files, or data formats generated by other monitoring solutions. Therefore, mining these log files and adapting these other data formats should be conducted first before monitoring with another tool. For reading other monitoring data formats, we provide an adapter which can be extended to read those files and send them our data format to the analysis nodes. Mining log files or even adapting monitoring data in a universal fashion remains as future work. However, for our goal of providing a live trace visualization of the software landscape this imposes only a minor limitation since we prototypically show and provide one way to monitor the software landscape and generate the visualization.

Furthermore, we assume that the possibility of instrumenting the existing applications exists. However, the existing applications must not be modified in some contexts. For instance, modifying the code after a conducted safety certification is often prohibited. Since our approach does not work without monitoring data, it will not function in this context. However, special hardware architectures exist where monitoring events are written in parallel to the execution. These events could be converted to our monitoring data format to enable the live trace visualization in this context which remains as future work.

A further implication of the former assumption is that some applications might not be instrumented and thus are not visible in the software landscape visualization. Therefore, a manual addition of those applications for the visualization should be provided but this also remains as future work.

A further assumption is imposed by the possibility to instrument and thus monitor the RPCs. Not every RPC technology might be instrumentable. For instance, it can be challenging to identify the RPC in a pure TCP communication protocol. Furthermore, to have a reliable sequence of the calls, an identifier should be inserted into the call. This might break the protocol if no extension mechanism is available. In such cases, other possibilities to monitor or model the RPC should be exploited.

To be able to reduce the traces on one worker node, similar execution traces have to be generated by the applications in a processing interval (e.g.,

five seconds). From our observations, web applications often impose similar traces if the behavior is not user-specific. However, if every trace is different from another, the scalable worker concept will not work.

Since we use cloud computing for scaling the trace analysis infrastructure, we assume that instances can be started or terminated on demand. If no cloud computing infrastructure is available, the analysis infrastructure should be deployed with the maximum expected peek capacity. Therefore, our approach still works but will probably be not as cost-efficient as if cloud computing would be available.

6

Monitoring and Trace Processing

This chapter introduces our monitoring and trace processing approaches which form the first three activities of our ExplorViz approach. Both approaches aim to provide a high throughput of monitoring records and to enable a live processing of the huge amounts of monitoring data that are typical for large software landscapes. Furthermore, the trace processing approach aims to scale with the amount of monitored applications and to be elastic to provide a cost-efficient solution.

First, the application-level monitoring is described in Section 6.1. Then, we present our RPC monitoring approach in Section 6.2. Finally, Section 6.3 illustrates our scalable and elastic trace processing approach.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013
2. [Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015

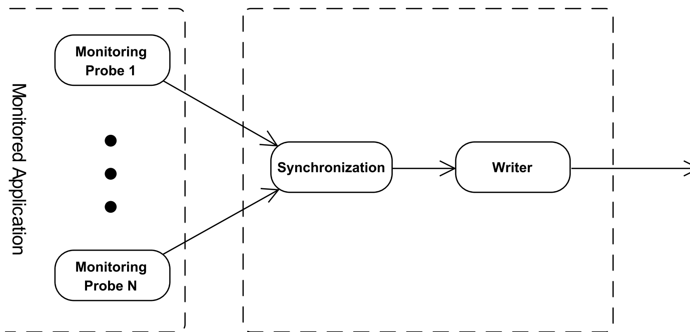


Figure 6.1. Filters involved in our application-level monitoring

6.1 Application-Level Monitoring

In this section, we describe our application-level monitoring approach. Afterwards, we detail the filters of our monitoring data adapter approach which enables to read data formats of other monitoring tools.

6.1.1 Monitoring Approach

Figure 6.1 shows the filters and application boundaries involved in our monitoring approach. The major concept is the spatial separation of the generation of the monitoring records and its processing. Hence, we aim for the collection of the minimal data set and directly transfer those data sets in an as small as possible format onto an analysis server.

At first, the *monitoring probes* get woven around the methods in the existing application. At runtime, these probes generate the required information for our later visualization. For instance, the object identifiers for the instance count, the names of the called method, and the order in which they were called. Since these probes run in the same thread as the application code does, they must be *synchronized* when passing the monitoring data to the writer. Then, the monitoring data is *written* to another analysis server.

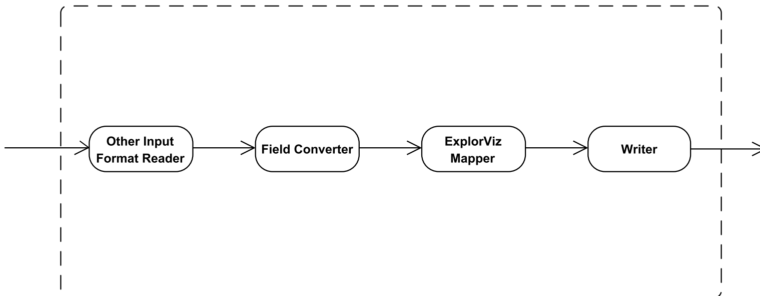


Figure 6.2. Filters in our monitoring data adapter approach

Another technique enabling a high throughput is that no redundant information are sent. For instance, only before events contain the method name. More precisely, they contain an ID for the method name since this often reduces the transferred data by replacing Strings with an integer representation before sending. Notably, this assumes that Strings are used multiple times which is the case when methods are called more than once.

6.1.2 Monitoring Data Adapter Approach

Often existing applications already generate some monitoring data. For this setting, our approach provides a monitoring data adapter. Via this adapter, we still support other inputs such as monitoring records generated by Kieker [van Hoorn et al. 2012]. Due to supporting Kieker, we also provide the ability of monitoring other programming languages than Java, for example, C and C++ [Mahmens 2014], Perl [Wechselberg 2013], or COBOL [Knoche et al. 2012].

Figure 6.2 shows the filters involved in the monitoring data adapter. It starts with *reading the other input format*. This reader can either read from a defined, finite source, e.g., logs from the filesystem, or can run continuously enabling our live trace visualization for other monitoring tools. After reading, the monitoring records typically need to be *converted*. For instance,

the signature of the called method must conform to our specification. After converting the data, the monitoring records are *mapped* to our ExplorViz input format. The mapped, ExplorViz-conform records are then *written* to an analysis component as if they resulted from our monitoring approach.

6.2 Remote-Procedure-Call Monitoring

To enable the visualization of the communication between applications, the RPCs have to be monitored. Different concepts for this RPC monitoring are described by Matthiessen [2014] which originate from a bachelor's project. Basically, two major concepts exist. The first one is using global trace identifiers and the second one is using local trace identifiers. In general, global trace identifiers impose a larger performance impact on each RPC since they must be synchronized. Since we aim for minimal impact on the monitored application and on high throughput, we utilize the second concept of sending additional information with the RPC (i.e., using local trace identifiers).

For example, when sending a SOAP message, we extend the SOAP header with information about the trace identifier and the callstack index. These are the local values of the sender. In addition, it sends a monitoring record containing those two values to our analysis component. The monitoring at the callee site receives the enriched SOAP message. Then, it generates a monitoring record containing the two values from the remote site (remote trace identifier and remote callstack index) and its own trace identifier and own callstack index. Then, this monitoring record is send to our analysis component. During the creation of the landscape model, these two monitoring records are matched and result in one RPC in our model. For a discussion of the other concepts – which we do not use –, we refer to [Matthiessen 2014].

Notably, this technique of backpacking a trace identifier and callstack index to the message comes with two disadvantages. At first, the addition of information to a RPC is often technology dependent. Therefore, every technology needs a special monitoring probe. The second disadvantage is that the technology might not be designed for addition of information

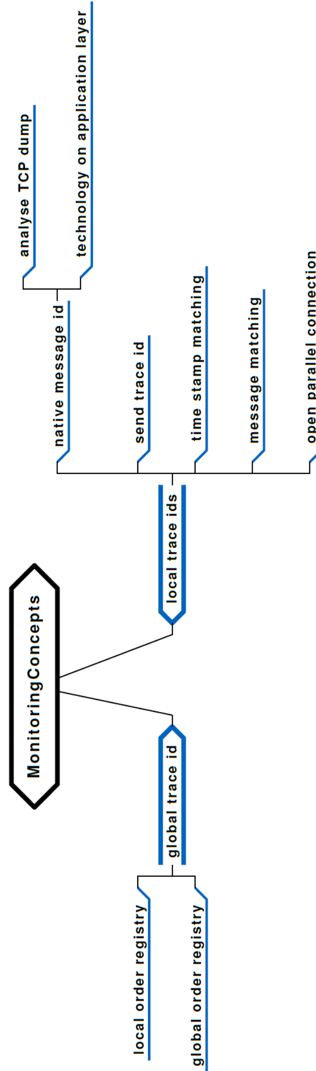


Figure 6.3. Different RPC monitoring concepts (taken from [Matthiessen 2014])

into one RPC. To monitor such technology, often source code changes to the technology are required making the solution also technology version dependent. However, a synchronization for global trace identifiers of every message is slow and thus not applicable for live trace visualization. Therefore, we use the backpacking approach despite the two disadvantages.

6.3 Trace Processing

In this section, we describe our scalable, elastic trace processing approach to circumvent the overutilization of a single analysis node by dynamically adding or removing preproccesing levels.

We start by providing a short motivation for our approach and then outline our basic idea. Then, our general scalable architecture is described. Afterwards, the filters of one analysis node, which enable the connection of multiple analysis workers in series, are explained. Then, we illustrate the scaling process for multiple worker levels. Assumptions and limitations of our trace processing approach were already discussed with the overall approach in Section 5.2.

6.3.1 Motivation

Employing only a single analysis node for live processing the monitoring data can easily become a bottleneck. For example, in our evaluation described in Section 10.4, the analysis would operate at full capacity after receiving load from only four monitored applications. In general, this number is determined by the amount of monitoring and the hardware of the analysis node. However, eventually every node will be fully utilized if the workload rises to some point.

Application-level monitoring tools, e.g., Kieker [van Hoorn et al. 2012], typically offer three configurable strategies, what should be done when the analysis cannot process the current monitoring data. The first strategy simply terminates the monitoring. Since this requires a manual restart of the application to start monitoring again, this behavior is undesirable for a high monitoring quality. However, this typically does not affect the SLAs of the monitored applications.

The second strategy discards new monitoring records until a space in the monitoring queue becomes available. Therefore, this behavior is similar to sampling which only monitors method calls on a defined interval, e.g., every 10th request. This strategy typically imposes no SLA violations at the expense of a reduced monitoring quality. However, it can automatically recover when the workload drops and thus is typically preferable over the first strategy and therefore, often employed in practice.

The third strategy uses blocking until a free space in the monitoring queue becomes available. While this behavior seems appealing on first sight, it can violate the SLAs when the analysis node takes a long time to recover from its high workload. The SLA violations are caused by the waiting of the application for finishing the writing of the monitored data. Therefore, it is not processing user requests often leading to loss in revenue due to annoyed customers.

This situation can become even more expensive, if the capacity manager utilizes the waiting user requests for its upscaling condition for the applications. Since only one analysis node is employed, the newly started application would also wait for the analysis node to finish. Therefore, the capacity manager might keep starting new instances until some node limit is reached and the service provider has to pay for application nodes that are waiting for the analysis of the monitoring data.

Based on the chosen strategy, either the quality of the SLAs or the monitoring quality is reduced with a fully utilized analysis node. One way to postpone this problem is an analysis node with a high number of CPU cores and a high amount of RAM. However, the analysis must be designed to utilize an potentially infinite number of cores and if the workload rises, the number of cores must be increased according to the peeks in the workload. Hence, they become superfluous during low workload.

6.3.2 Idea

Figure 6.4 illustrates the basic idea of our elastic trace processing approach. When the analysis master impends to become overutilized, a new worker level is dynamically added in front of it. Similar to the MapReduce pattern [Dean and Ghemawat 2010], each worker on the new level analyzes one part

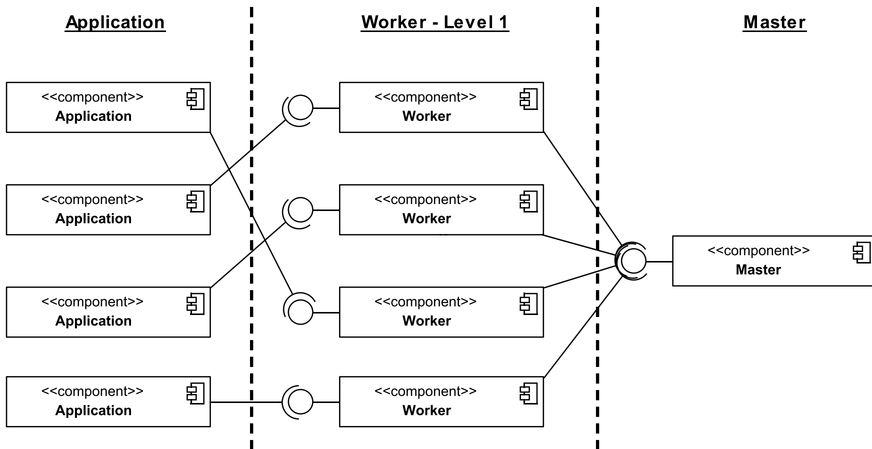


Figure 6.4. Basic idea of dynamic worker levels

of the monitoring data. To circumvent an overutilization of the workers, the associated worker applications are scaled within their worker level. With this preprocessing step, the *Master* is only required to combine the analysis results. Eventually with rising workload, the merging of the results impends to overload the *Master* again. Then, a second level is dynamically inserted between the first level and the *Master*. In theory, this behavior can continue to even more levels.

6.3.3 Scalable Architecture

Next, we show our general scalable architecture. Figure 6.5 displays this architecture including our capacity manager *CapMan*[1] and one master node. Therefore, it represents the initial state when only a small amount of monitoring data has to be analyzed. In our architecture, the capacity manager includes the workload generation and its load balancing due to convenience reasons. Therefore, the applications are accessed by *CapMan* to

[1] `https://github.com/ExplorViz/capacity-manager`

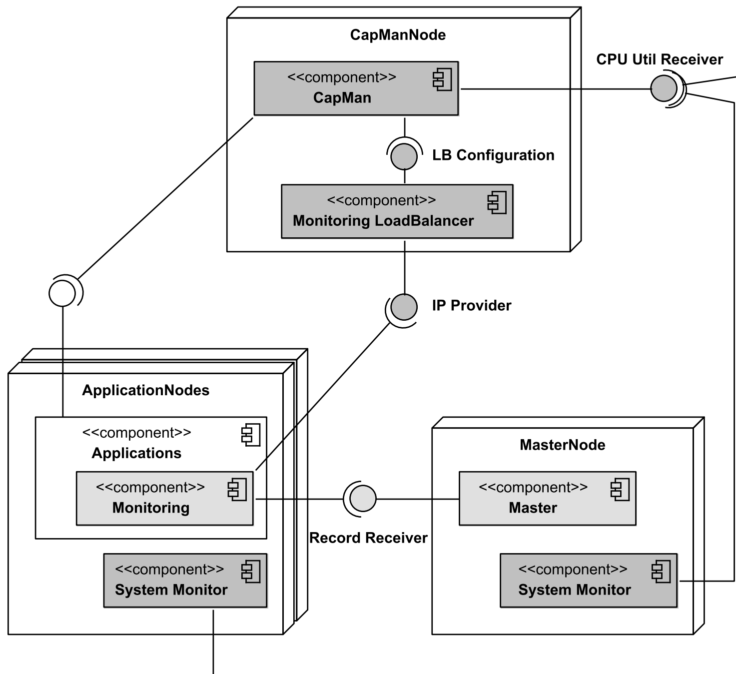


Figure 6.5. Our scalable trace processing architecture

simulate user requests. A *System Monitor* records the CPU utilization of the application nodes and sends this utilization to *CapMan*. *CapMan* uses these values, in addition to the outstanding request count from the workload generation, for scaling the applications. This cycle forms the employed load generation on the applications and their automatic elastic scaling.

Each involved application contains a *Monitoring* component. At its start, it requests an IP address from the *Monitoring LoadBalancer*.[2] This request contains a *loadbalancing group* property to determine the kind of application which the *Monitoring* component wants to access. For example, the applications use *analysis* to reflect their wish to write monitoring data on

[2] `https://github.com/ExplorViz/load-balancer`

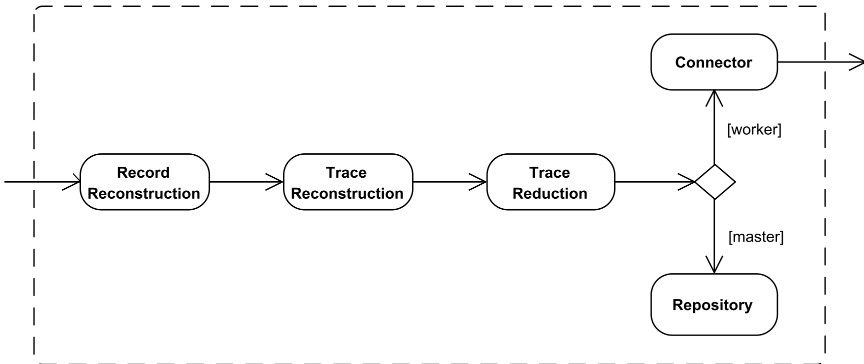


Figure 6.6. Filters in the analysis component (worker and master)

an analysis node. In Figure 6.5, the shown state only exists of one analysis node, i.e., the master node. Therefore, the *Monitoring* component receives the IP address of the master node and sends its monitoring data to the master analysis application.

After a defined interval, the *Monitoring* component again fetches an IP address from the *Monitoring LoadBalancer* and if necessary connects to the newly received IP address. Therefore, the monitoring data is distributed to different nodes when multiple nodes (e.g., on a worker level) are available. This results in an approximate equal utilization of the target nodes. Similar to the application nodes, the CPU utilization of the analysis nodes is sent by a *System Monitor* to *CapMan* which uses these values for scaling the analysis nodes. If a new analysis node is started by *CapMan*, the IP address of the newly started node is registered in the *Monitoring LoadBalancer* under a defined *loadbalancing group* property.

6.3.4 Analysis Approach

To enable a series connection of the different worker levels, the analysis component follows the filters shown in Figure 6.6. The monitoring data is received and a record reconstruction step creates the monitoring records

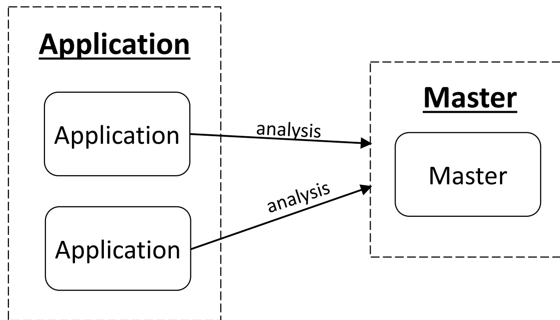


Figure 6.7. Initial state before scaling

objects. The records are passed to the trace reconstruction step which links the loose method call records to an execution trace representing the full execution path of one user request. Afterwards, the traces are passed to a trace reduction filter. The chance of same traces typically increases when multiple user requests are conducted. For example, most of the users will access the main page of a website which will often generate the same execution trace in an application. To save network bandwidth and CPU cycles on the next analysis node in the chain, similar traces are reduced to one trace class. For monitoring how many times the trace class was called, it contains an attribute *called times* and runtime statistics (e.g., minimum and maximum duration) for the monitored method calls. To be able to identify which host might behave differently, the runtime statistics are formed on a per host basis.

If an analysis node is started as a worker node, the trace classes are sent to the next analysis node in line via a connector which sends these trace classes as serialized monitoring records again. If the analysis node is running as the master node, it creates or updates the landscape model.

6.3.5 Scaling Process

In Figure 6.7, the state from Figure 6.5 is visualized in a simplified form. The boxes with dashed lines represent one scaling group, i.e., a group of

applications which is scaled independently by a capacity manager. The name of each scaling group is displayed at the top. There are two scaling groups: Application and Master. Arrows illustrate accesses to the target scaling group. The label of an arrow is the *loadbalancing group* name used to request an IP address from the *Monitoring LoadBalancer*. In the initial state, the applications access the *Master* scaling group by using the loadbalancing group name *analysis*. Decoupling the scaling group name and the loadbalancing group name enables the worker levels to get dynamically inserted or removed between each processing level.

Upscaling

Figure 6.8 illustrates the process of dynamically adding one worker level. After the CPU utilization of the *Master* rises over a defined threshold, this process is triggered. At first, a new loadbalancing group is created which is named *worker-1* and contains the *Master*. Then, *two* new worker nodes are started. We assume the same configuration on each analysis node. Therefore, starting only one worker node would result in the same high CPU utilization encountered on the *Master*. The new worker nodes send their data to the scaling group which is resolved by the loadbalancing group name *worker-1*. This state is visualized in Figure 6.8a. After the worker application on the nodes are started, the two workers are added to the loadbalancing group *analysis* and the *Master* is removed from it. The final state is illustrated in Figure 6.8b. Notably, the order of adding and removing loadbalancing groups is important because the analysis should not be paused during the scaling process.

Downscaling

The downscaling process follows the upscaling process in reverse order. However, we employ a different scaling condition. Our first approach was using the analogous CPU utilization of the *Master* when it falls below a defined threshold. However, this condition is independent from the amount of nodes in the previous worker level. Therefore, it would also trigger when the previous worker level contains, e.g., 10 nodes and shutting down all of

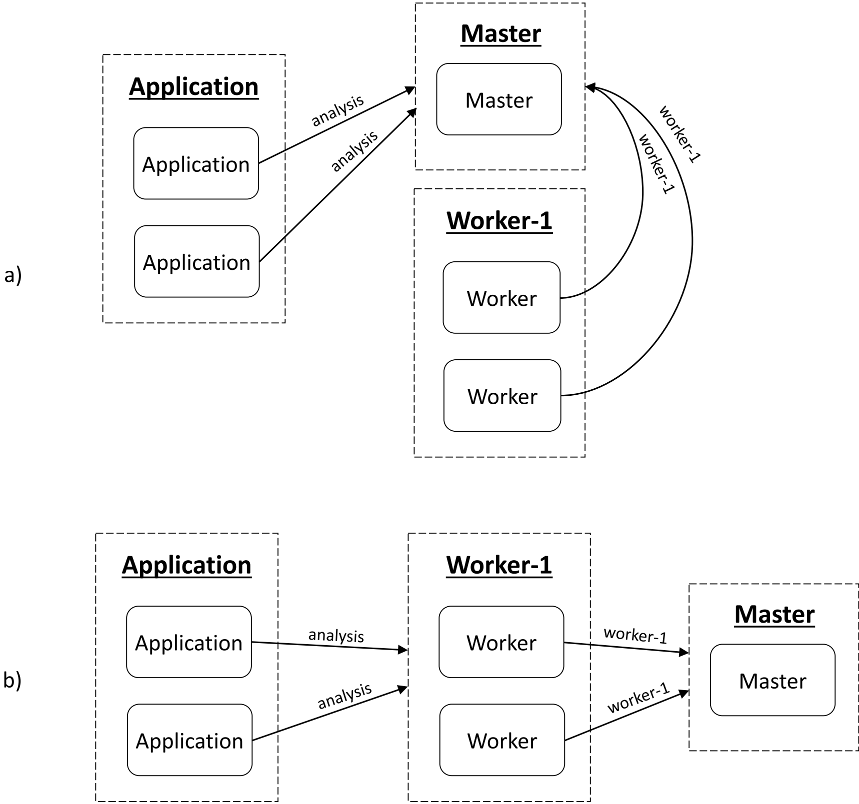


Figure 6.8. Activities forming the upscaling process

them would typically result in an overutilization of the *Master*. This could be lifted by only downscaling when there are exactly two nodes of the previous worker level left. However, this still contains no statement about the utilization of the previous worker level. For example, both workers might be heavily utilized. Hence, we use the CPU utilization of the previous worker level as downscaling condition. When only two nodes are left in the previous worker level and the average CPU utilization falls below a defined threshold in this scaling group, it is shut down and removed by following the upscaling process in analogous reverse order. Therefore, downscaling is not delaying or pausing the analysis either.

7

Landscape Model Generation

After processing the traces, the landscape model is generated from the information contained in the traces. Afterwards, the landscape model provides the required information to construct the visualization. Notably, the meta-model represents the sole information of the software landscape without generated data, i.e., no visualization model.

We start by presenting our landscape meta-model in Section 7.1. Then, Section 7.2 describes how the traces are mapped to a landscape model.

Previous Publications

Parts of this chapter are already published in the following work:

1. [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

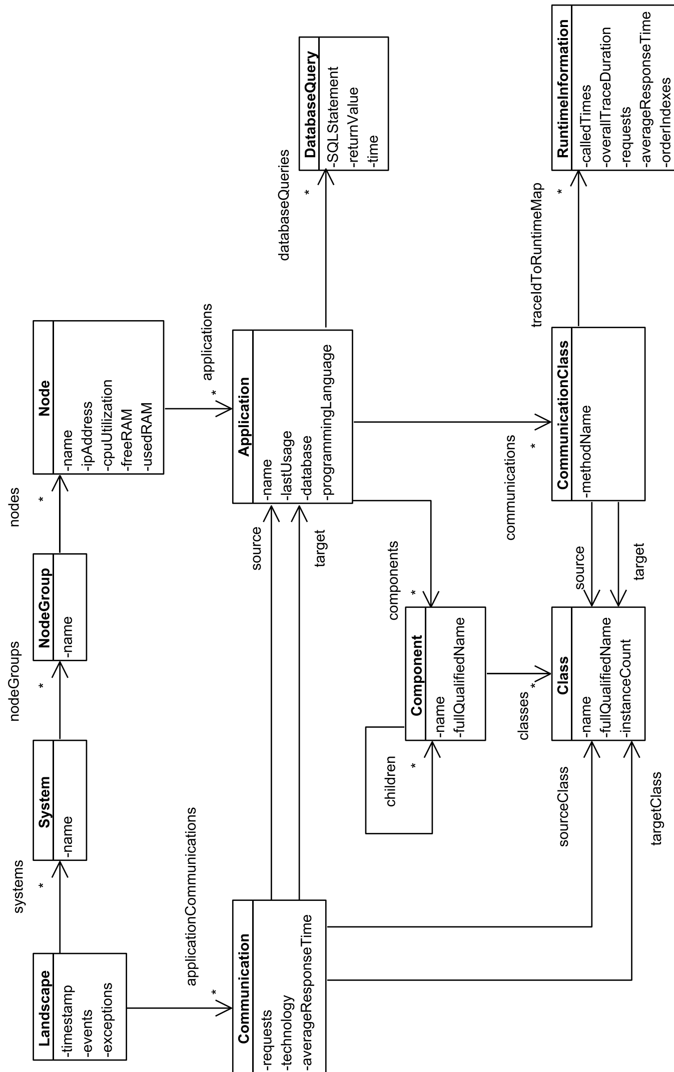


Figure 7.1. Landscape meta-model

7.1 Landscape Meta-Model

Figure 7.1 displays our landscape meta-model. A *Landscape* class represents the top-level entity of the meta-model. It contains a *timestamp* attribute to represent the timestamp of the landscape model's creation. Furthermore, it has a list of landscape *events*, e.g., "Node Demo1 has been added" and a list of *exceptions* which occurred in the landscape. The *Landscape* class has a reference list of *System*s. In our terms, a *System* represents a logical union of multiple applications and servers. The *System* class contains a *name* attribute representing the actual name of the system. Furthermore, *System* holds a reference list of *NodeGroup*s.

A *NodeGroup* forms a logical abstraction from the servers and applications by representing servers which have the same application configuration. An equal application configuration typically occurs in, for instance, cloud environments. A *NodeGroup* contains an attribute *name* which is formed by the range of the lowest IP address and the highest IP address in it. Furthermore, A *NodeGroup* has a reference list to *Node*s. One *Node* has a *name* (hostname) and an IP address. In addition, it contains attributes to represent the current utilization of the server, i.e., free and used RAM amount, and the current average CPU utilization. A *Node* holds a reference list of *Application*s running on it.

An *Application* has the attributes *name*, *last usage* representing the last timestamp where activity was monitored, whether it is a *database* or not, and the *programming language*. In addition, it holds a list of *DatabaseQueries* which represent one database query, i.e., its *SQL statement*, the *return value*, and the execution *time*. An *Application* contains a reference list to *Component*s. *Component*s represent logical organization units of classes. For example, packages can be *Component*s in the context of Java. A *Component* has the attributes *name* and *full qualified name*. The latter represents the full name including the names of parent *Component*s. Furthermore, a *Component* contains a list of *Component*s, i.e., its children, and a list of *Class*es.

A *Class* represents the lowest level entity in our meta-model. It has a *name*, a *full qualified name* representing its name and including parent components' names, and an *instance count* attribute representing how many instances were active.

The *Landscape* class also holds a list of *Communication* paths which exist between *Application*s. In addition, the *source* and *target* class are referenced by the *Communication*. A *Communication* has the attributes *requests* which can also be zero, an attribute *technology* meaning the actual technology used to conduct the RPC, and an *average response time* of the RPCs.

An *Application* also contains a list of *Communication*s but on the *Class* level. The *CommunicationClass* class contains a *method name* attribute and has a *source* and *target Class*. To be able to visualize the actual traces in the application-level perspective, it also holds a reference map between a trace identifier and a *Runtime Information*. A *Runtime Information* provides the information about how many *times* the trace was *called*, the *overall trace duration*, the *request* amount of the method, the *average response time*, and a list of *order indexes* representing the position in the trace.

7.2 Trace-to-Model Mapping

In this section, we describe how the information contained in the processed traces are used to create a landscape model. Each trace includes *HostApplicationMetaDataRecord*s which contain information about the originating system, the IP address and hostname of the server, the application name, and the programming language. Since the traces are generated in a distributed system and potentially each monitoring record might be processed on different analysis servers, every operation monitoring record contains such information. The contained information are used to create the counterpart in the landscape model. For example, the existing systems are iterated and if the current system is not found, a new instance for the new system is created. This procedure also applies to nodes and application instances. After inserting a new node or application into the model, the node groups get updated accordingly.

For generating the components, classes, and communication on the application level, the operation monitoring records of the trace are iterated. The contained method signature is used to create or update components and classes. The method call is used to create or update the communication on the class level. In addition, the *Runtime Information* is created as part of this communication.

Beneath this normal mapping of the traces, there exist separate records for special purposes. As already described in Section 6.2, the special caller and callee records for RPC monitoring are matched to create the communication on the landscape level. Another special record is the *System-MonitoringRecord* which contains information about the CPU utilization and RAM usage. This record is used to update the utilization of a node. Furthermore, the record for monitoring database calls is used to generate a list of database queries for each application.

8 ExplorViz Visualization

In this chapter, our live trace visualization approach for large software landscapes is presented. It consists of two perspectives, namely a landscape-level perspective and an application-level perspective. The former perspective uses a 2D visualization employing a mix of UML deployment and activity diagram elements. The latter perspective consists of a 3D visualization following the city metaphor. For best accessibility, our live trace visualization is web-based.

At first, four investigated use cases for our visualization are described. Then, we present basic concepts and forces which influenced our design of the visualization. Afterwards, the landscape-level perspective and the application-level perspective are introduced. Our VR approach for providing an immersive experience in the application-level perspective is described in Section 8.5. Finally, Section 8.6 details our approach to provide physical models of the application-level perspective.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

2. [Fittkau et al. 2015h] F. Fittkau, A. Krause, and W. Hasselbring. Hierarchical software landscape visualization for system comprehension: a controlled experiment. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

3. [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

4. [Fittkau et al. 2015f] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

5. [Fittkau et al. 2015j] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. Technical report 1507. Department of Computer Science, Kiel University, Germany, June 2015

6. [Fittkau et al. 2015i] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

8.1 Use Cases

We investigated four use cases for our visualization which are detailed in the following.

8.1.1 System and Program Comprehension

As already noted in the goals, the system and program comprehension context is our main use case for the designed visualization [Fittkau et al. 2013b; 2015, a; h]. The knowledge about the communication and usage of the applications forming a large software landscape often gets lost. Therefore, our visualization aims to support the efficient recovery of this knowledge by providing abstractions for easier comprehension of the software landscape. Furthermore, it supports in the program comprehension process of each application in combination with the landscape level. In a distributed environment, investigating one isolated application can be insufficient. For example, when a performance anomaly occurs, it is essential to find the root cause of the anomaly which might have resulted from the interaction of the applications.

8.1.2 Control Center

A further use case for our visualization is building a control center for detecting and solving performance anomalies [Fittkau et al. 2014b]. When an anomaly is detected, we feature different perspectives for analyzing the root cause, planning countermeasures, and visualizing the actual execution of the plan. The concept and implementation for such a control center are presented in Chapter 12 as part of the extensibility evaluation.

8.1.3 Performance Analysis

Performance analysis is another use case for our trace visualization [Fittkau et al. 2015g]. Our monitoring collects the response times of each monitored method and how often it was called. Based on this information, the visualization providing an overview of the system, and a special performance

analysis dialog, the software engineer is able to conduct a performance analysis. For instance, she can limit the visible communication lines to only show the communication where the response times were high.

8.1.4 Architecture Conformance Checking

The last use case, which we investigated for our visualization, is architecture conformance checking on the landscape-level perspective [Fittkau et al. 2014a; Simolka 2015]. It can either be conducted manually or also compared automatically to a target architecture model. For automatic conformance checking, we feature a special perspective to model the target architecture model featuring our visualization as graphical modeling language. Then, the differences of the actual landscape architecture and the target architecture are visualized in the landscape-level perspective.

8.2 Basic Concepts

There are several basic concepts and forces that influenced the design of our visualization. These are detailed in the following.

8.2.1 Interactive Exploration

In a large software landscape, information must be presented in a limited time span. Hence the presentation of the information must be quickly comprehensible and only show information on the required level. Therefore, the major concept of ExplorViz is based on revealing additional details, e.g., the communication on deeper levels, on demand. This interactive exploration principle provides the visual scalability of our approach.

8.2.2 Grouping

A further consequence of the described limited time span is that entities that do not add to the overview in the comprehension process get grouped and aggregated. For example, often servers with the same application

Figure 8.1. Time series of the overall activity in the landscape

configuration get started in cloud environments. Thus, an overview should provide the information which classes of application configurations exist and not show each server as one entity at the start. This concept of introducing abstractions plays an important role in our visualization to support an effective and efficient system and program comprehension process.

8.2.3 Time Shift

For some analyses, particular traces or situations are of interest. During a live trace visualization the software engineer can only analyze a situation for a short duration before the visualization updates itself. ExplorViz supports the analysis of specific situations and traces through a *time shift mode*. The user can pause the visualization to switch into an offline mode where she is able to go back to a particular past visualization. A switch back to a live visualization is always possible resulting in a combination of live and offline trace visualization.

An exemplary time series for the time shift feature is shown in Figure 8.1. It provides the timestamps for the available past visualizations in combination with the overall activity which happened in the software landscape. This overall activity is an indicator for the software engineer to look for special circumstances, for instance, low or high activity in the software landscape.

8.2.4 Restricted Access

Since often not everyone in the network should be able to access the visualization, we feature a login system such that every user must authenticate

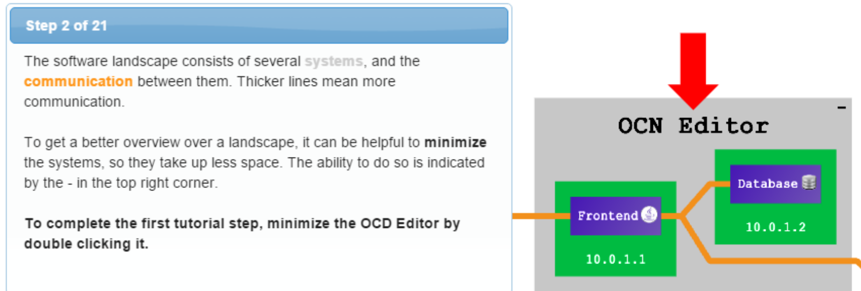


Figure 8.2. One step of our interactive tutorial

before accessing it. This authentication also lays the foundation for future integration of, for instance, access roles (only seeing authorized information) and team-based distributed program comprehension processes.

8.2.5 Based on Dynamic Information

Our approach is not relying on upfront static information, since it would be necessary to conduct a static analysis for every single application in the software landscape. Therefore, we take only the dynamic information into account. Another advantage is that unused classes are not displayed which adds to providing an easier overview of the actual system.

8.2.6 Interactive Tutorial

Instead of showing hints for the usage and semantics in the visualization, we provide an interactive tutorial mode [Finke 2014] upfront for the user. The tutorial is started at the first login of a user. Figure 8.2 shows one step of the tutorial. In addition to the explanatory text, the visualization is enhanced by a marker which points at the entity that should be interacted with.

8.3 Landscape-Level Perspective

This section introduces our landscape-level perspective. At first, the semantics of the visualization are presented. Then, we describe related features and aspects of the landscape-level perspective, i.e., the layout, an event and exception viewer, and how we visualize the results of an architecture conformance check.

8.3.1 Semantics

Figure 8.3 shows the modeled infrastructure of the GEOMAR's Kiel Data Management Infrastructure for ocean science[1] on the landscape level. The large gray boxes with, e.g., PubFlow (❶) [Brauer and Hasselbring 2013a], represent the systems present in the software landscape. They can also be minimized such that only the system and its communication are visible, without their interior. Thus, providing abstraction on the level of systems and only visualizing systems currently in focus.

The smaller green boxes in one system represent the contained node groups (❷) or nodes (❸). Node groups are labeled with a textual representation of their contained nodes, for example, "10.0.0.1 – 10.0.0.7". We introduced node groups because in cloud computing, for instance, nodes are scaled for performance reasons, but typically keep their application configuration. For providing an overview, these nodes are grouped. However, they can be extended with the *plus* symbol near the node group.

A node can contain different applications (❹). The communication between applications is visualized by lines. In accordance to their call count, the line thickness changes, i.e., higher amount of communication leads to thicker communication lines (❺). The user can navigate to the application-level perspective by choosing one application.

As already stated, we feature a time shift mode to analyze specific situations (❻). To provide an indication, when large amounts of calls are processed, the call count of the entire landscape is shown on the y-axis. A configurable time window is shown on the x-axis.

[1] `https://portal.geomar.de`

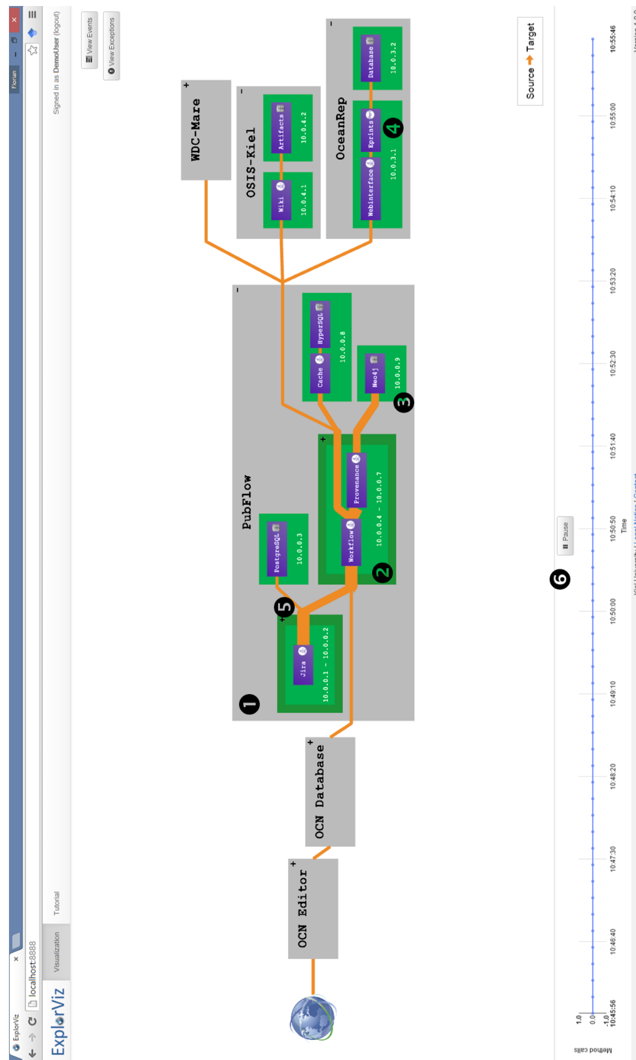


Figure 8.3. Landscape-level perspective modeling the Kiel Data Management Infrastructure for ocean science

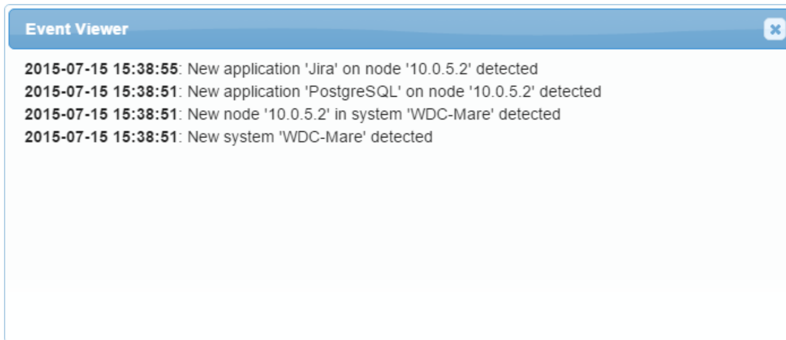


Figure 8.4. Event viewer for the software landscape

8.3.2 Layout

We employ auto-layout algorithms to ensure that the user does not need to manually layout the nodes which can be infeasible in large software landscapes. The employed flow-based auto-layout, named KLay Layered[2] [Schulze et al. 2014], orders the nodes and applications in accordance to our defined communication flow direction, i.e., from left to right. Future work should make the flow direction configurable in the GUI.

8.3.3 Event and Exception Viewer

We provide an event viewer to log changes of the software landscape. An example log is displayed Figure 8.4. When new nodes or applications get detected, an event entry is generated. Therefore, the software engineer can quickly locate changes which happen in the software landscape.

In a similar fashion to the event viewer, exceptions are shown. Since exceptions should point out that some part is misbehaving and should not be the normal behavior of an application, exceptions are visualized at the landscape-level perspective and not on the application-level. Therefore, if any exception occurs in the software landscape, the software engineer is able to quickly notice and view the exceptions.

[2] `http://rtsys.informatik.uni-kiel.de/confluence/x/joAN`

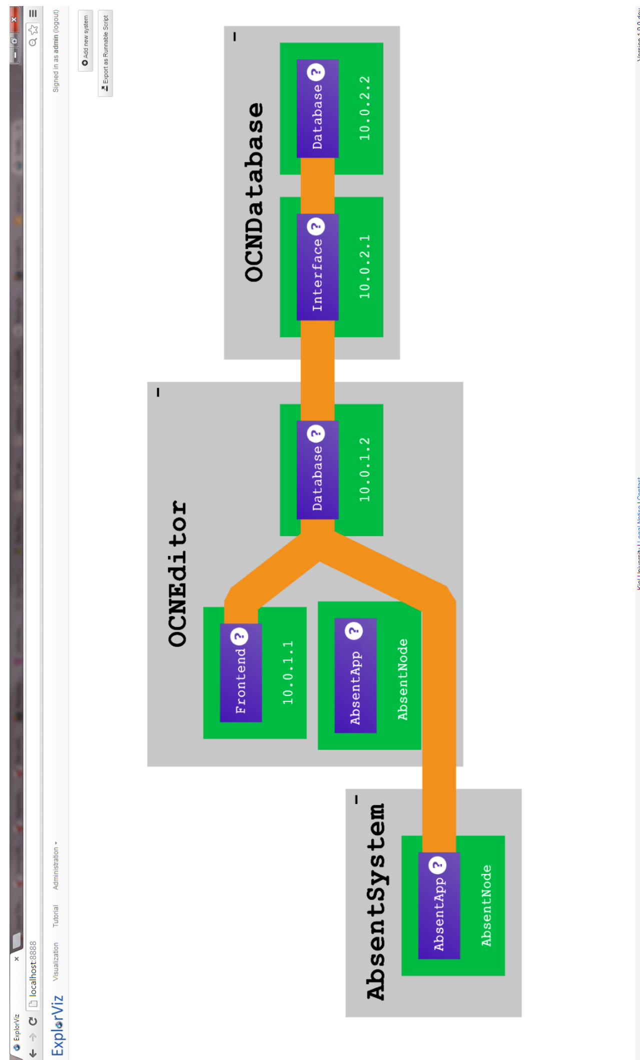


Figure 8.5. Conceptual architecture of the example software landscape in the modelling perspective (taken from [Simolka 2015])

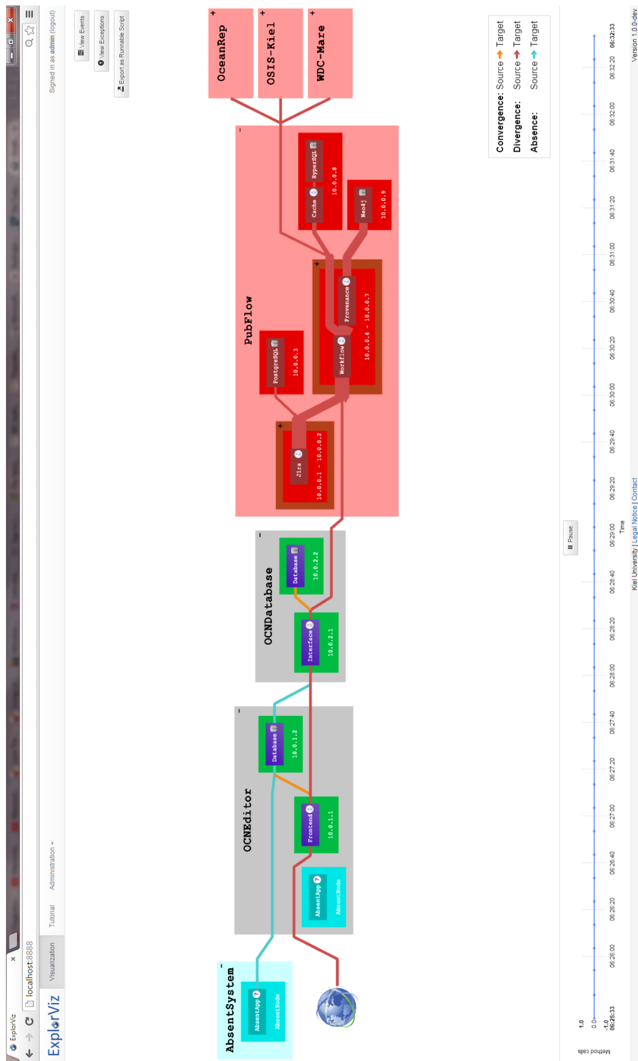


Figure 8.6. Landscape-level perspective showing the differences in the conceptual architecture and actual architecture of the software landscape (taken from [Simolka 2015])

8.3.4 Architecture Conformance Checking

When a software architect intends to check whether the actual software landscape architecture still conforms to the conceptual architecture, she first models the conceptual architecture in the modeling perspective of ExplorViz. Then, she switches to the landscape-level perspective and the differences between the runtime model and the conceptual model are automatically displayed [Simolka 2015].

Figure 8.5 presents an example conceptual architecture and Figure 8.6 shows the resulting landscape-level perspective visualizing the differences of the conceptual and the actual architecture. Again, we model the GEOMAR's Kiel Data Management Infrastructure in the actual architecture. The convergent communication and entities are visualized in their normal color. Absent communication and entities are drawn in shades of blue (for example, the *AbsentSystem* system) and divergent ones are shown in shades of red (e.g., the *PubFlow* system). For easier accessibility, a small legend in the lower right corner presents this semantic to the user. For details about the checking process, we refer to [Simolka 2015].

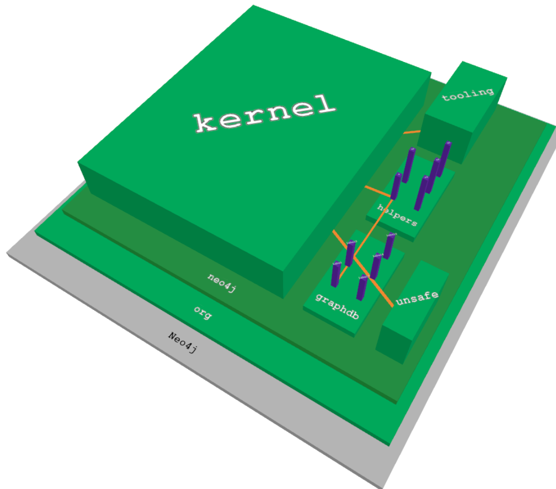
8.4 Application-Level Perspective

Our application-level perspective is based on the *3D city metaphor* [Knight and Munro 2000; Wettel and Lanza 2007]. After introducing the semantics of the application-level perspective, different features and aspects of this perspective are discussed.

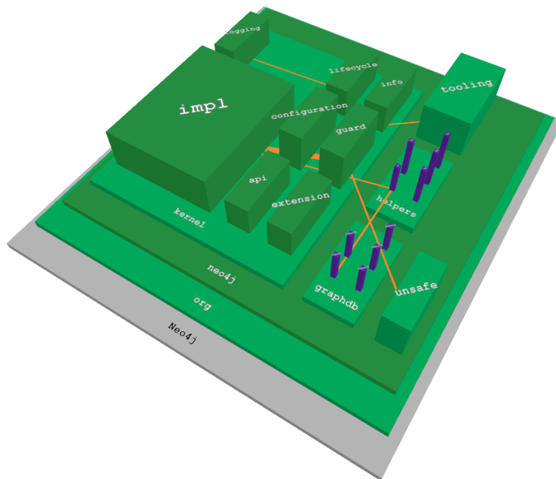
8.4.1 Semantics

In Figure 8.7a, the structure of a mockup of Neo4j[3] is visualized as an example. The smallest, purple boxes represent classes. The height of each class maps to its active instance count in the last monitored interval and the width of a class is defined as one unit.

[3] `http://neo4j.com`



(a) Mockup of Neo4j with closed *Kernel* component



(b) Mockup of Neo4j with opened *Kernel* component

Figure 8.7. Application-level perspective of a mockup of Neo4j

The green boxes represent the components of the application. Notably, to be programming language independent as far as possible, we only specify components to be an organizing unit, for example, packages can be used for Java. However, also folders can be components. The height of a component is the maximum height of its contained classes or components, i.e., the highest instance count. The width property maps roughly to the amount of classes contained in it. The interaction amount of the components or classes is visualized by the thickness of the connecting lines.

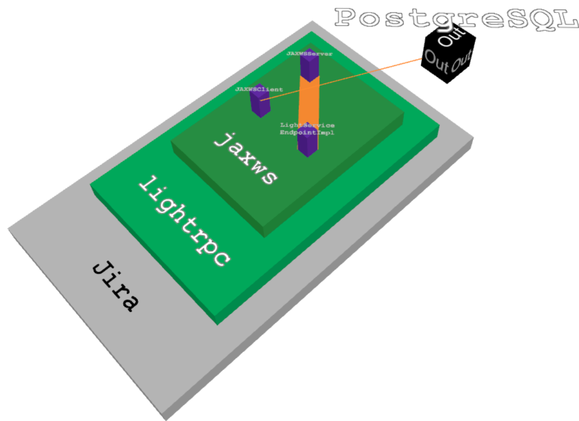
Contrary to other approaches utilizing the city metaphor, we do not visualize the whole application in class-level detail at once. We follow an interactive top-down approach, where only top-level components and their relationships are shown, i.e., hiding internal details of those components. Our navigation concept bases on focusing the entities and interactions that are currently of interest. This concept is visualized by the transition from Figure 8.7a to Figure 8.7b. After analyzing the interaction between the top-level components, we might want to get more information by opening the `Kernel` component to find the root cause of high interaction. Therefore, we provide two kinds of components, i.e., opened and closed components. Opened components (Figure 8.7a) hide their internal details and closed components (Figure 8.7b) show their internals.

At the bottom of the visualization is a foundation platform represented by a gray box. It contains the application name of the visualized program. The platform is important when multiple top-level packages exist.

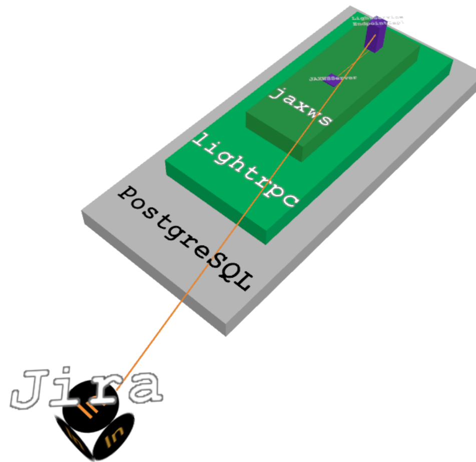
To visualize the connection to other applications, we feature in- and outgoing ports. Notably, a port is only visualized when a connection exists. Figure 8.8a shows an outgoing connection and Figure 8.8b shows an incoming connection on the counterpart application. With these ports, the software engineer is able to follow the trace starting at the class handling the incoming connection and thus an entry point for the analysis is provided.

8.4.2 Layout

Figure 8.9 shows an overview of the used layout. It is inspired by the rectangle packing layout algorithm of CodeCity [Wettel 2010, pp. 35–38] and follows its basic steps. The only difference exists in ordering when two entities have the same size.



(a) Showing an outgoing RPC to *PostgreSQL*



(b) Showing an incoming RPC from *Jira*

Figure 8.8. Visualizing the in- and outgoing communication in the application-level perspective

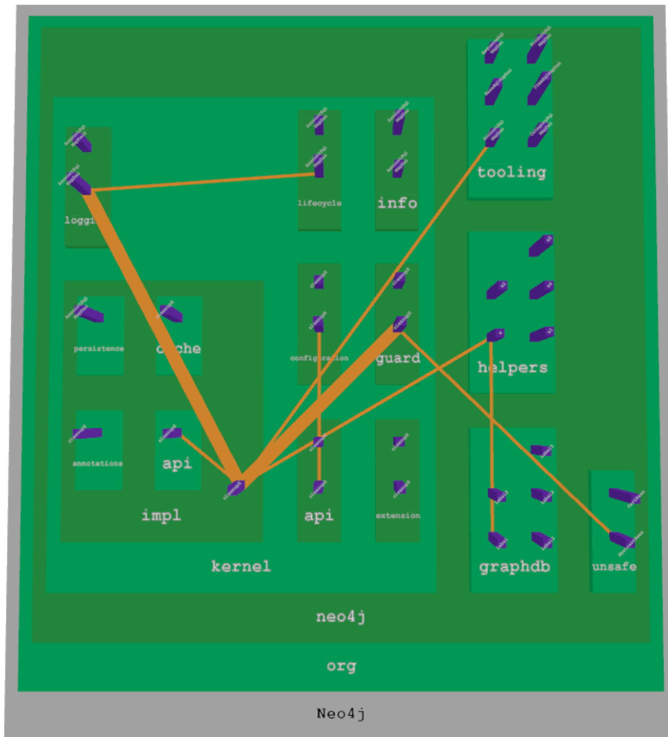


Figure 8.9. Overview of the application-level layout

Notably, the communication lines are not part of the layout process and thus are simply drawn after the entities were arranged, leading to overlaps over other entities. Furthermore, the layout is not stable. When a new component is detected, it might change the whole layout of the application.

Therefore, we developed other layouts but found no satisfying solution. The requirements for our new layout were the following. The layout should be stable, i.e., when a new component is detected, the following layout should only differ slightly from the previous layout without the new component. Furthermore, the layout should be compact, since screen

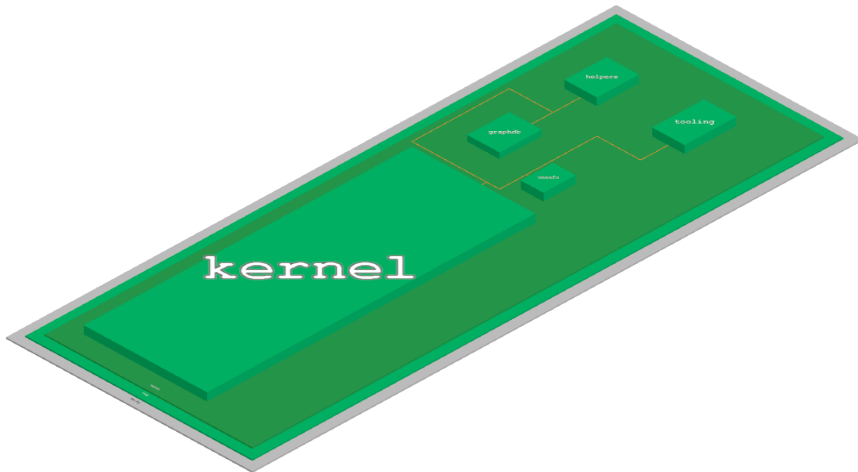


Figure 8.10. Alternative layout using quadtrees and free space is cut off (taken from [Barbie 2014])

space is limited and the user should perceive the visualization using as less scrolling as possible. In addition, it should respect our hierarchical concept of interactive exploration. Due to this requirement, we can not use the stable layout of Steinbrückner [2010]. A further requirement is that the calculation time of the layout should be fast since the visualization is updated every 10th second as default value and the visualization should still be perceived as responsive. Since we are using the third dimension, an optional requirement is that occlusion of the visual entities is minimized.

The best alternative layout for the application-level was developed by Barbie [2014]. He used quadtrees [Finkel and Bentley 1974] to organize the components and classes. A quadtree splits the space in four squares where one square is also a new quadtree. The components and classes are hierarchically inserted into this quadtree structure. After this, the free squares in the quadtree get cut off to spare space and make the layout more compact. The communication lines are routed with 9 pins along one quadtree. Therefore, the communication lines get a square shape.

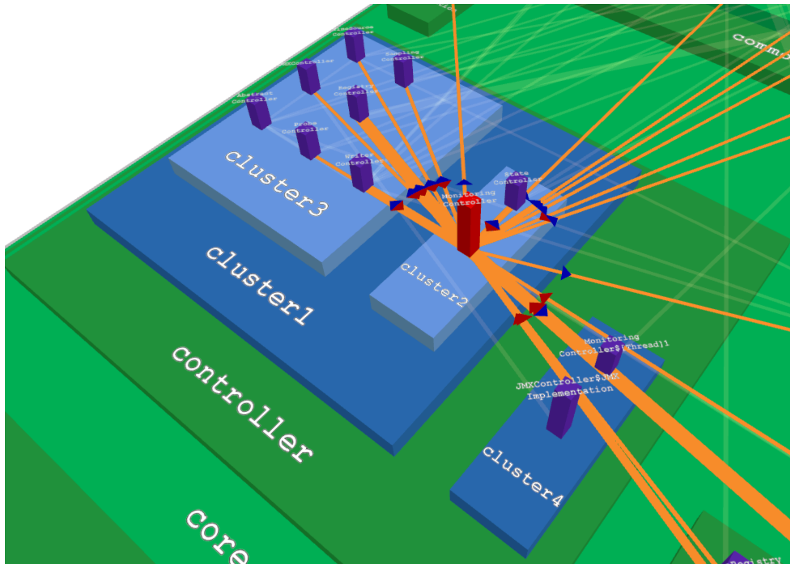


Figure 8.11. Clustered *controller* component from Kieker

In Figure 8.10, the same Neo4j mockup as in Figure 8.9 is visualized except that the developed quadtree layout is used. The layout is more visual appealing since no communication lines overlap other entities. However, it also gets larger and less compact. Stability is reached in some circumstances but not in all situations. Unfortunately, the calculation time for medium-sized applications increases to a few seconds which renders it unusable for those applications. Therefore, we still use the rectangle packing approach by Wettel [2010] and a better application-level layout remains as future work. Details about the quadtree layout can be found in [Barbie 2014].

8.4.3 Clustering

The visual scalability of our approach is provided by the interactive exploration of the displayed application. A prerequisite for being able to explore it are the components. If no components or too large components

exist in the application, our approach does not scale. Therefore, our visualization features clustering techniques to create synthetic components [Barzel 2014]. The resulting synthetic components, when clustering the *controller* component in Kieker, are visualized in Figure 8.11. For clustering, we use a weighting of lexicographic, number of instances, and number of incoming and outgoing communications. For more details about the clustering approach, we refer to [Barzel 2014].

8.4.4 Database Queries

Performance issues often result from inefficient database queries. Therefore, we utilize the monitoring probe of Zirkelbach [2015] to gather the execution time, return value, and SQL statement of each query. The database queries get visualized by a table for each application. Figure 8.12 shows this table for a JPetStore instance. The sortable and searchable table shows the SQL statement, the return value of the query, and the duration of the query in milliseconds.

8.4.5 Trace Replayer

Since we provide a live trace visualization which updates itself every 10th second as the default interval, we do not show single traces in our visualization. However, in some circumstances the execution path matters. For example, when a method call imposes a long response time, the originating path is of interest. For analyzing one trace, we provide a trace replayer which is visualized in Figure 8.13. With the trace replayer, the user is able to conveniently play and pause the execution steps [Kahn 2006] of a trace. In addition to the position, the caller, the callee, and method name, the average execution time is displayed. During playback, the view follows the chosen communication line and then proceeds to the next method call. In addition, the callee, the caller, and the method name are highlighted in the 3D visualization.

Figure 8.12. Dialog showing database queries conducted by JPetStore

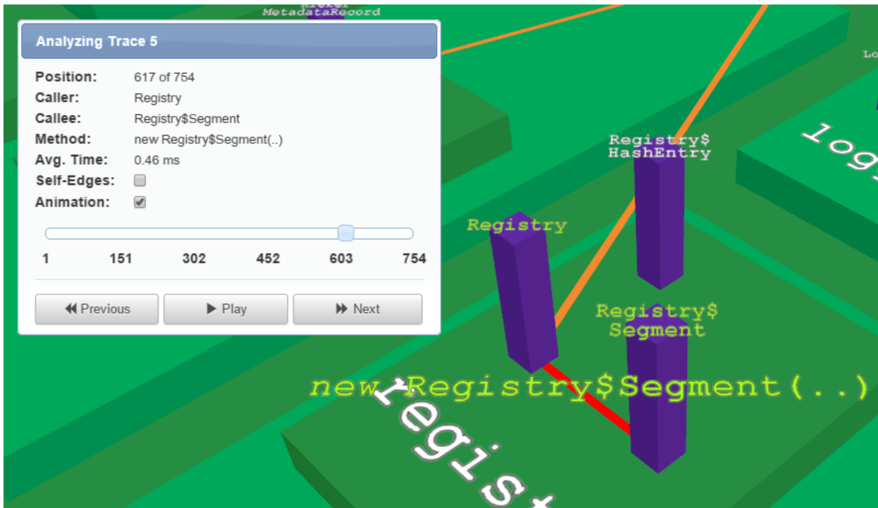


Figure 8.13. Replaying a monitored trace originating from Kieker

Figure 8.14. Analyzing the performance in Kieker

Figure 8.15. Dialog showing the code structure and *Version.java* of Neo4j

8.4.6 Performance Analysis

Since manually searching for the highest response times in a performance analysis is cumbersome, we provide a performance analysis tool for this case [Jähde 2015]. When the user activates the filtering for response times, every communication, where the average response time is below or equal to the input threshold value, is hidden. Therefore, only method calls with an average response time of above the threshold value are shown in the visualization.

Furthermore, the called times and average response time of an adjacent communication get shown and highlighted when the user highlights a class (see Figure 8.14). A further feature of the performance analysis tool is searching for method names. A software engineer might intend to analyze the performance of a specific method. Therefore, she can us the search feature. For further details about the performance analysis tool, we refer to [Jähde 2015].

Figure 8.16. Setup for our VR approach (doing a translation gesture)

8.4.7 Code Viewer

Source code viewing is considered important [Koschke 2003] especially during a program comprehension process. Therefore, our visualization provides the possibility to open a dialog that displays the source code for each class, if available. The code viewer in Figure 8.15 displays the source code structure as a tree and the source code of the *Version.java* of Neo4j.

8.5 Gesture-Controlled Virtual Reality Approach

Although 3D software visualizations can deliver more information compared to 2D visualizations, it is often difficult for users to navigate in 3D spaces using a 2D screen and a 2D input device [Teyseyre and Campo 2009]. As a consequence, users may get disoriented [Herndon et al. 1994] and thus the advantages of a third dimension may be abolished.

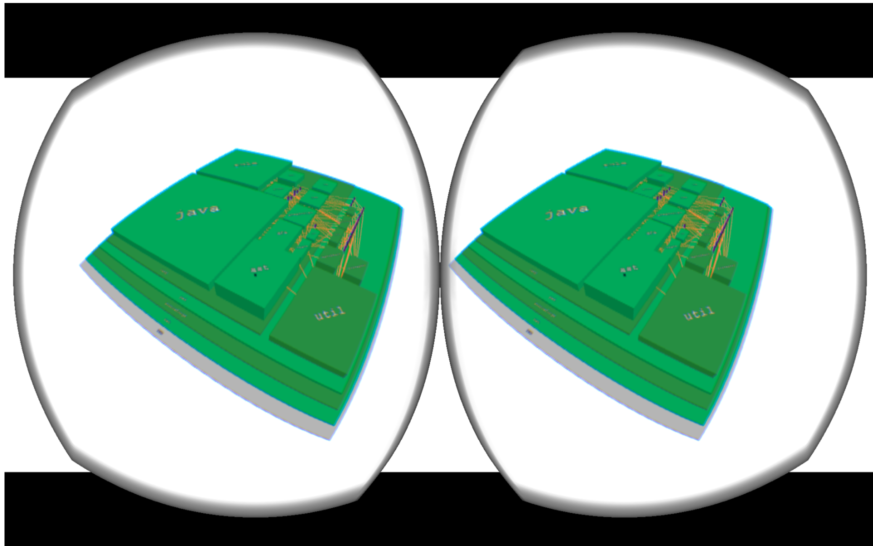


Figure 8.17. View on the city model of PMD through the Oculus Rift DK1

VR can employ the natural perception of spatial locality of users and thus provide advantages for 3D visualization [Elliott et al. 2015; Delimarschi et al. 2014]. In addition to stereoscopic [Ware et al. 1993; Ware and Mitchell 2005] display, natural interaction beyond the 2D mouse provides advantages, e.g., creativity can be enhanced by walking around [Oppezzo and Schwartz 2014]. Therefore, we developed a VR approach for exploring the application-level perspective with a HMD and gesture-based interaction. The basic setup of our VR approach is visualized in Figure 8.16. We utilize an *Oculus Rift*[4] for displaying the city model and use gesture recognition realized with a *Microsoft Kinect v2*.[5] For implementation details, we refer to [Krause 2015].

We start by introducing the used display and the resulting implications for the visualization. Then, we discuss the gesture control and the design of the gestures.

[4] `http://www.oculus.com`

[5] `http://www.microsoft.com/en-us/kinectforwindows`

8.5.1 Display

The Oculus Rift is a HMD. Its rotation sensor and larger field of view provide a more immersive user experience compared to preceding HMDs. For our VR approach, we utilize the Development Kit 1 (DK1) version of the Oculus Rift with an overall display resolution of 1280x800 pixels.

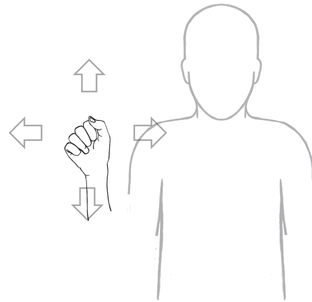
Since the Oculus Rift uses one image for each eye, the city model created on the application-level perspective needs to be rendered twice with a different 3D transformation, i.e., a slight translation between the eyes. Figure 8.17 shows a screenshot of the resulting image sent to the Oculus Rift. The values from the rotation sensor in the Oculus Rift are used to rotate the viewpoint in the virtual space. Hence, users only need to rotate their head to view near model elements which enables a higher immersion experience.

8.5.2 Gestures

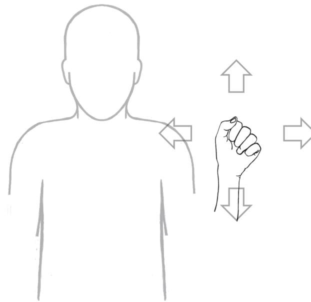
For gesture recognition, we use the *Microsoft Kinect v2*. To enable gesture control for ExplorViz, a C# application developed by Krause [2015], which sends the commands after gesture recognition to the browser, needs to be deployed on every client.

In general, there are two basic concepts for designing gesture-based actions. The first concept is commonly used by control sticks in game controllers. A user performs a gesture and holds the desired position at a boundary of the recognition field. While she holds this position, the movement is conducted continuously into the implied direction. The second concept is a direct mapping between the hand movement and the movement action in the model, similar to how a computer mouse works.

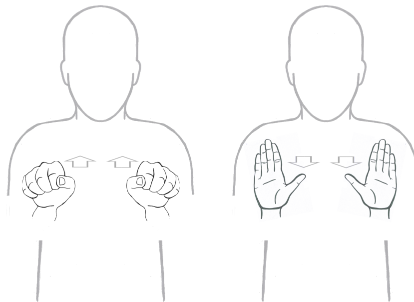
In our prior tests, users familiarized with a direct mapping faster than with the first concept. Furthermore, users working with the continuous movement sometimes tried to manipulate the model as if they would use a direct mapping approach. Thus, we discarded the first concept and designed our gestures with a direct mapping of hand movement to model movement.



(a) Gesture for moving the model



(b) Gesture for rotating the model



(c) Gesture for zooming in and out

Figure 8.18. Gesture concepts for interacting with the city model (taken from [Krause 2015])

In the following, we describe the designed gestures for our VR approach. We discuss different design alternatives and present our final gesture design. For an example of the execution of these gestures, we refer to the video recordings [Fittkau et al. 2015b] of the structured interviews conducted by Krause [2015] for the evaluation of the VR approach.

Translation

Figure 8.18a shows the gesture for translating and thus moving the model. The user raises the right hand, makes a fist, and then moves the hand. This gesture is derived from moving an object in reality by graping and then moving it. Furthermore, the gesture is similar to dragging and swiping on touch-based devices. Since this gesture was quickly understandable in our first tests with test subjects, there is no other design for this gesture.

Rotation

The first design for the rotation gesture was derived from holding and spinning a ball with two hands. A virtual line between the hands formed an axis used for the rotation. Unfortunately, it was not possibly to detect all real-world interactions, e.g., rotation of the hands, due to some restrictions of the Kinect sensor. For example, when the hands overlap in the depth dimension, the hidden hand joints are not detected correctly. Figure 8.18b shows the final design of this gesture. It is very similar to the translation gesture and only differs in using the left hand instead of the right hand.

Zoom

The first design for the zoom gesture used the body tilt as zooming in or out. Leaning forward with the upper body resulted in zooming in and leaning backward led to in zooming out. Tests revealed that users tend to move their head while leaning forward or backward. Hence, they also rotate the viewpoint due to the Oculus Rift rotation sensor. This rotation confused the users during the performance of this gesture. The next design involved walking forward and backward to zoom in and out. Due to the possible lack of space and some users rotating their body while walking,

this gesture was also inappropriate to zoom in or out. Figure 8.18c shows the final design. The gesture is derived from real life interaction similar to rowing. To zoom in, the user raises her hands, closes both hands, and pulls them towards her chest. To zoom out, the user raises her hands and pushes them away from the upper body. Thus, the gesture maps to pulling and pushing the model towards or away from the user.

Selection

For selecting an entity in the city model, the user raises her right hand, then closes and quickly opens it. To open or close a package, users need to do the closing and opening of the hand twice. Since the Kinect might recognize a half-open hand as closed, it is important to fully open the hand for this gesture.

Reset

If users get lost in the 3D space by, for instance, translating too far, they can reset their viewpoint to the origin by performing a jump. The first design of this gesture required the raising and subsequent closing and opening of both hands above the head. However, tests revealed that this design is inappropriate, since users sometimes adjust the wearing of the Oculus Rift with their hands and thus might trigger this gesture. In contrast, jumping is completely different to the other gestures and is therefore unlikely to be triggered unintendedly.

8.6 Physical 3D-Printed City Models

Building architects, but also civil or mechanical engineers often build from their designs physical 3D models for a better presentation, comprehension, and communication among stakeholders. Software engineers usually create visualizations of their software designs as digital objects to be presented on a screen only. 3D software visualization metaphors, such as the employed software city metaphor, provide a basis for exporting those on-screen software visualizations into physical models. From our virtual city models, we

create physical 3D-printed models to transfer the advantages of real, physical, tangible architecture models from traditional engineering disciplines to software engineering.

First, we discuss potential usage scenarios for those models. Afterwards, we describe how we build the physical models from virtual ones and finally the encountered challenges during this process are presented.

8.6.1 Usage Scenarios

We envision several potential usage scenarios for physical models which we will outline in the following.

Program Comprehension in Teams Gestures support in thinking and communication processes [Goldin-Meadow 2005]. Since physical models are more accessible than 2D screens and provide a natural interaction possibility, they might increase the gesticulation of users. This might lead to faster and better understanding when applied in a team-based program comprehension scenario due to its supporting nature. Furthermore, the advantages might increase when applied in larger teams. Since software systems are often changing, the model should only be printed for special occasions, e.g., a new developer team or upcoming major refactorings.

Educational Visualization A further usage scenario is the usage of 3D models for educational purposes. Like an anatomic skeleton model used in a biology course, 3D models of design patterns, architectural styles, or reference architectures could be 3D-printed. Advantages include the possibly increased interest of the students and due to 3D visualization and the possibility to touch the model, there might be a higher chance to remain in long-term memory. Further interaction possibilities with the 3D model could be developed to support the learning process of the students.

Effort Visualization in Customer Dialog A further potential field of application are dialogs with customers. Customers often see the GUI as the program since the actual program logic code is often invisible for them.

Therefore, the – possible large – effort to add a feature or to refactor the code is also often invisible for them. Presenting a physical 3D model of the status quo and another physical 3D model of the desired state, might convince the customer of the effort of the required change. Notably, this could also be achieved with two on-screen software visualizations but a touchable and *solid* 3D model might provide higher conviction.

Saving Digital Heritage We envision physical models to be a step toward saving the digital heritage of 3D software visualizations. Compared to programs, physical models do not depend on the availability of SDKs, library versions, or hardware and thus are less vulnerable to changes of the external environment. Often it is uncertain, whether the code can still be run in thirty years. In contrast, depending on the material (e.g., resin or metal), physical models might last hundreds of years. One might argue that pictures of the 3D visualizations are sufficient. However, they do not provide interaction possibilities and can suffer from occlusion. Contrary, physical models still provide the possibility to interact (e.g., rotate) avoiding possible occlusion.

8.6.2 From Virtual to Physical Models

After creating the on-screen model in ExplorViz by monitoring an application run, we export the model as an OpenSCAD[6] script file. To fit the build platform of our low-budget 3D-printer (a Prusa i3[7] shown in Figure 8.19), we often have to split the exported model into several jigsaw pieces and export each piece as an STL[8] file as input for the 3D-printer. After printing each jigsaw piece, we glue the pieces together. Finally, we have to manually paint the single-colored model.

Figure 8.20 shows a 3D-printed city model of PMD.[9] The fully assembled physical model is 334 mm wide and 354 mm deep. The overall building time of the physical PMD model sums up to 58 hours and the costs of material

[6] http://www.openscad.org

[7] http://reprap.org/wiki/Prusa_i3

[8] http://www.ennex.com/~fabbers/StL.asp

[9] http://pmd.sourceforge.net

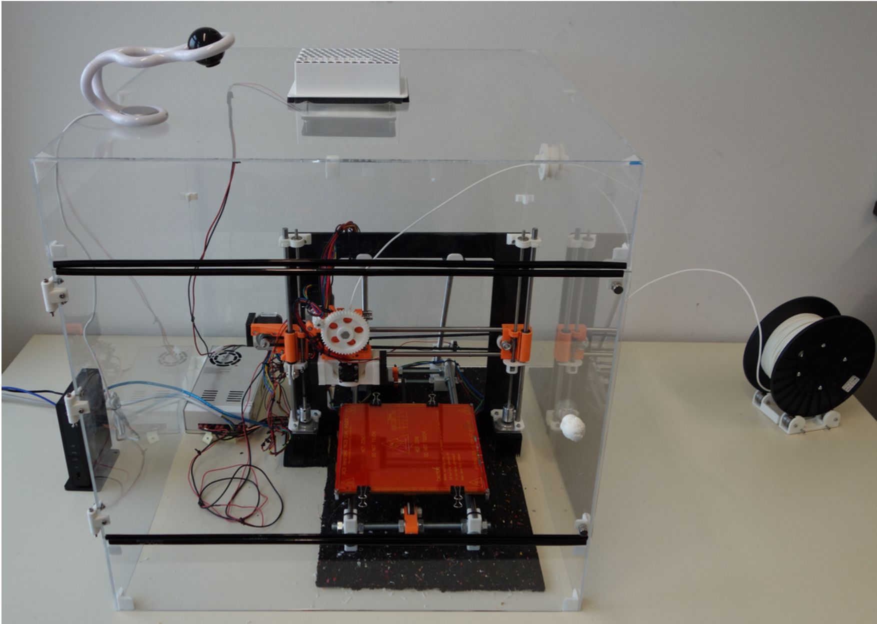


Figure 8.19. Prusa i3 used for printing the physical models

are about 9 Euro on our low-budget, self-built 3D-printer. However, 3D-printing the model on a modern multi-color printer would take only a small fraction of the building time (a few hours) and save several of the described working steps.

8.6.3 Encountered Challenges

During the process of creating a physical model from a virtual city model, we encountered several challenges. In the following, we provide an overview of these and present how we overcame them.

Figure 8.20. Physical 3D-printed model of PMD

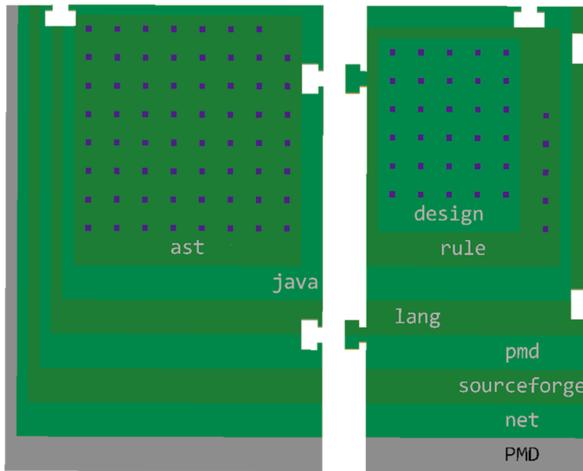


Figure 8.21. Two jigsaw pieces of our PMD model

3D-Printer Input Format

Since being a new technology in the consumer market, the input format for 3D-printers may still change. To mitigate this risk, we export an OpenSCAD script files of the virtual models and not a directly 3D-printable formated file. The script files contain commands how to create entities, for instance, boxes with their position and size. The compiler of OpenSCAD uses these script files to render 3D objects. The rendered objects can be exported to six different file formats, at the time of writing. The currently most popular 3D-printer input format STL is among them.

Mapping Virtual Dimension to Physical Dimension

The virtual city model has its own virtual dimension in ExplorViz but this dimension has no concrete relation to a physical dimension. To create a physical model, we had to find a mapping coefficient.

Figure 8.22. One unpainted jigsaw piece of our physical PMD model

There are two opposing factors. The smallest parts are the buildings and the labels. Both should be printed as large as possible to avoid fragile buildings and to ensure readability of the labels. On the opposite, the overall model should be as small as possible since a huge model would require more time to grasp the model. After a period of prototyping, we found a mapping coefficient which is a trade-off between both factors.

Creating Labels

To provide a useful physical model, the components had to be labeled. Our first approach included a font library for OpenSCAD, which implemented every letter as a grid of pixels, i.e., boxes. The print result was not satisfactory since the underlying font was unreadable at the desired size. Since June of 2014, OpenSCAD directly supports rendering fonts via the `text()` command. However, some fonts are harder to print for the 3D-printer and are less readable at small size. Therefore, we tested some fonts and achieved the best performance with *Consolas*.

Limited Build Volume

We use a Prusa i3 to print our city models due to its rectangular build platform. It can print objects with a size up to 200 mm^3. However, our models can easily get larger than this volume, as in the case of the PMD model presented in Figure 8.20. Therefore, we split the physical models into multiple smaller jigsaw pieces using the PuzzleCut[10] library for OpenSCAD. Figure 8.21 shows two jigsaw piece from our PMD model rendered in OpenSCAD. After the print, the model is assembled and agglutinated.

Monochromacity

Most of the current consumer 3D-printers, which use the fused filament fabrication technique,[11] create only single-colored objects. Three-colored objects are possible but the 3D-printer must be upgraded. Our city metaphor models contain six colors and thus they require a different solution at the moment. We prime the printed model using a white spray can. Afterwards, we use table top colors to manually paint the models. An unpainted model and some used colors can be seen in Figure 8.22.

8.6.4 Meta-Monitoring

A special model created by us is the physical 3D model of the visualization of ExplorViz. To monitor the monitoring (similar to meta-monitoring Kieker with Kicker [Waller 2014]) visualization ExplorViz, we first had to instrument the GWT-generated JavaScript methods. Therefore, we had to write a monitoring component for JavaScript. As already utilized in our Java monitoring component, we utilize aspect-oriented programming to instrument the JavaScript methods of ExplorViz. After gathering the required information, the records are sent to a server which uses Kieker to write the monitoring logs onto the disc. Then, the logs are read in with our monitoring data adapter and visualized in ExplorViz. The resulting visualization showing ExplorViz in ExplorViz is shown in Figure 8.23. From

[10] http://nothinglabs.blogspot.de/2012/11/puzzlecut-openscad-library.html

[11] http://reprap.org/wiki/Fused_filament_fabrication

this visualization, we exported the OpenSCAD script and followed the previously described process for creating the physical model which is depicted in Figure 8.24.

Figure 8.23. Virtual model of ExplorViz in ExplorViz

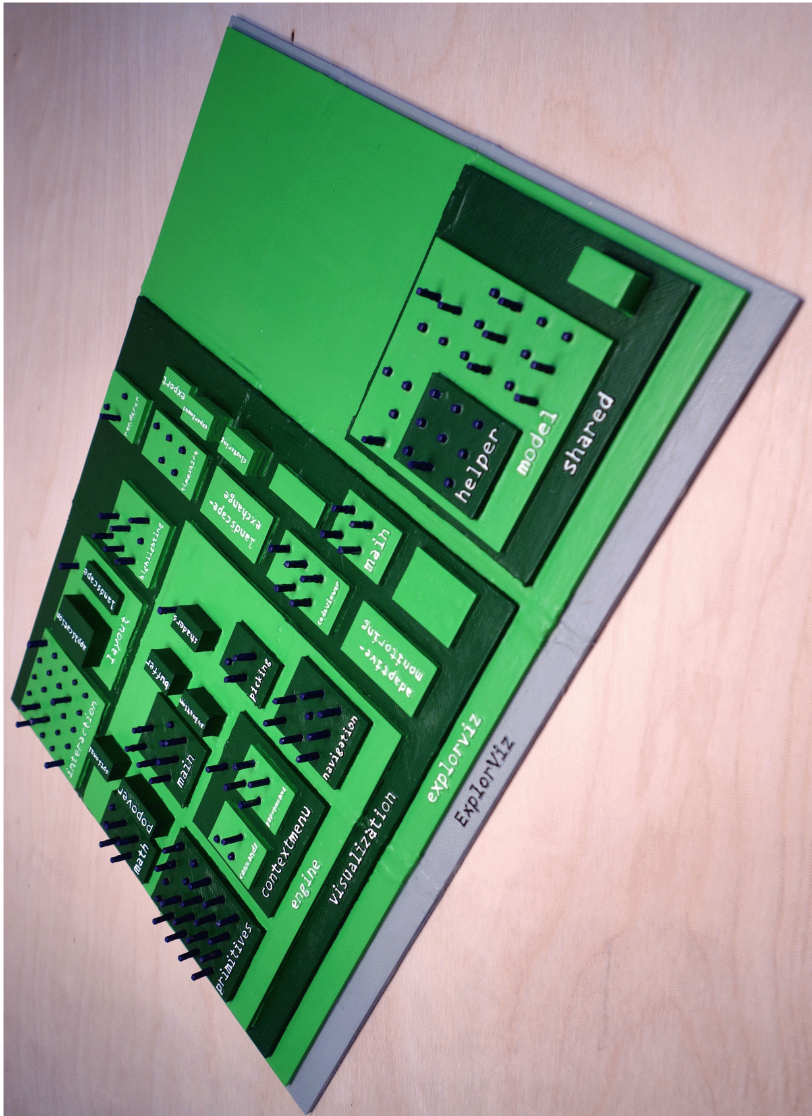


Figure 8.24. Physical 3D-printed model of ExplorViz

Part III

Evaluation

9

ExplorViz Implementation

In order to evaluate our approach in lab experiments and controlled experiments, it has to be implemented first. Therefore, this chapter describes the realized implementation which forms the basis for each evaluation. Due to about 32,000 lines of code of ExplorViz without comments and blank lines – measured with the tool CLOC[1] –, we focus on the main activities and technologies involved. We leave out class diagrams since the contained details would be too excessive. For further details about the implementation and its classes, the source code is freely available at the corresponding Git repository.[2]

We start by describing the overall architecture of ExplorViz (see Section 9.1). Then, the implementation of the *Monitoring* component is presented in Section 9.2. Afterwards, Section 9.3 presents an overview of the implementation of the *Analysis* component. In Section 9.4, we describe the implementation of the *Repository* component. Finally, the components used in the implementation of our web-based visualization are presented (see Section 9.5).

[1] `http://cloc.sourceforge.net`

[2] `https://github.com/ExplorViz`

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013

2. [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

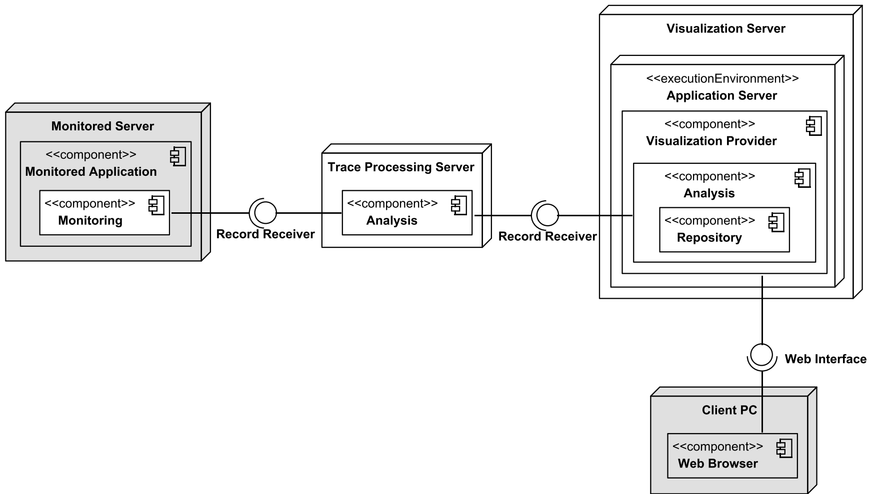


Figure 9.1. Overall architecture of our ExplorViz implementation

9.1 Overall Architecture

Figure 9.1 depicts an overview of the overall architecture of our core components in ExplorViz. To enable a better comprehension, the execution environments and servers are also denoted. It starts with monitoring of the existing application utilizing the *Monitoring* component. The generated monitoring records are sent to an *Analysis* component running on a *Trace Processing Server*. As already described in Section 6.3, this worker is only started if the master analysis node impends to be overutilized.

After preprocessing the traces, they are sent to the *Analysis* component running on the *Visualization Server*. The *Analysis* component is part of the *Visualization Provider*. This provider – deployed with a WAR-file – runs inside an application server, such as Jetty.[3] A part of this *Analysis* component

[3] `http://www.eclipse.org/jetty`

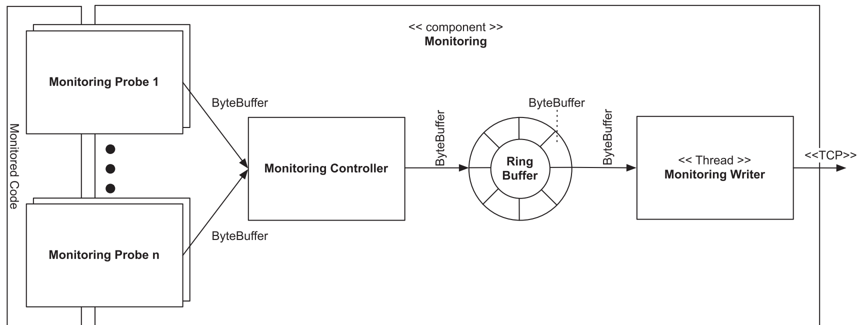


Figure 9.2. Overview of the monitoring component

is the *Repository* component which handles the creation and update of the landscape model. The landscape model is passed to the client, i.e., the *Web Browser*, via HTTP when the client requests it.

9.2 Monitoring

In this section, we describe the implementation of our monitoring component which aims to provide high-throughput monitoring for live monitoring data analysis.

In Figure 9.2, an overview of our *Monitoring* component is presented. The *Monitoring Probes* are integrated into the monitored code by aspect weaving using AspectJ.[4] They collect the method's information and write the gathered information sequentially into an array of bytes realized by the Java native input/output class *ByteBuffer*. The first byte represents an identifier for the class that is later constructed with these information and the following bytes contain information like, for instance, the logging timestamp. Then, the *ByteBuffer*s are sent to the *Monitoring Controller* which in turn puts the *ByteBuffer*s into a *Ring Buffer* for synchronization since

[4]https://eclipse.org/aspectj

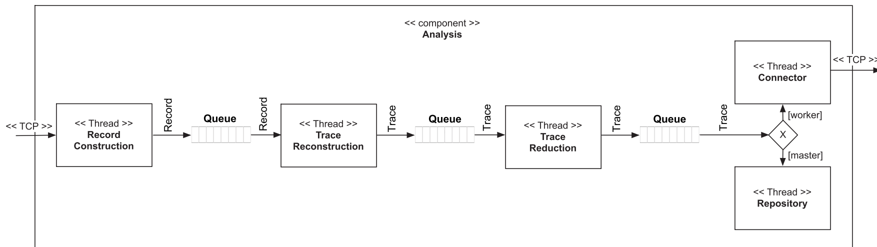


Figure 9.3. Overview of the analysis component

every probe can have different execution threads. For realizing the *Ring Buffer*, we utilize the disruptor framework.[5] The *Monitoring Writer*, running in a separate thread, receives the *ByteBuffers* and writes them to the *Analysis* component utilizing the Java class *java.nio.channels.SocketChannel* and the transfer protocol TCP. We use TCP since Beye [2013] evaluated that TCP is the fastest technology for this purpose.

Contrary to most other application-level monitoring tools, for example, Kieker [van Hoorn et al. 2012], we do not create record objects in the probes, but write the data directly into a *ByteBuffer*. This usually results in less garbage collection and the time for the object creation process is saved. For monitoring the CPU utilization and memory consumption on the servers, we utilize the Java class *com.sun.management.OperatingSystemMXBean* which is available since Java version 1.7.

9.3 Analysis

In Figure 9.3, an overview of our *Analysis* component is presented. Concerning the overall architecture, we follow the pipes and filters pattern.

The sent bytes are received from the monitoring component via TCP. From these bytes, *Monitoring Records* are constructed and passed into the

[5] `http://lmax-exchange.github.io/disruptor/`

first *Queue*. The queue implementation is a bounded single-producer-single-consumer queue provided by the JCTools.[6] It provides the advantage of lower synchronization overhead in comparison to the *Ring Buffer* used in the *Monitoring* component.

The *Trace Reconstruction* filter receives the *Monitoring Records* and reconstructs the execution traces. Then, these traces are forwarded into the second *Queue*.

The *Trace Reduction* filter receives these traces and reduces their amount by utilizing the technique of equivalence class generation also called trace summarization. This technique is used since Weißenfels [2014] evaluated possible trace reduction techniques, and trace summarization was rated as the best technique for our live trace processing purpose. The filter is configured to output reduced traces each four seconds. Each trace is enriched with runtime statistics, e.g., the count of summarized traces or the minimum and maximum execution times. The resulting reduced traces are put into the third *Queue*.

If the *Analysis* component is configured to be a worker, the reduced traces are sent to the *Connector*. This connector then writes them to another chained *Analysis* component, either further workers or the master. If the *Analysis* component is configured to be the master, the reduced traces are passed to the *Repository* component where they are used to create the landscape model.

Each filter runs in a separate thread and is therefore connected by a *Queue*. This design decision is made because every filter has enough work to conduct when millions of records per second must be processed.

9.4 Repository

Figure 9.4 displays an overview of the *Repository* component. For easier deployment, it is embedded in the *Analysis* component if it is configured as a master. It receives the reduced traces from the *Trace Reduction* step and generates a *Landscape Repository Model* from the incoming traces in the *Landscape Repository Sink*. After a defined interval – default is ten seconds –,

[6] `https://github.com/JCTools`

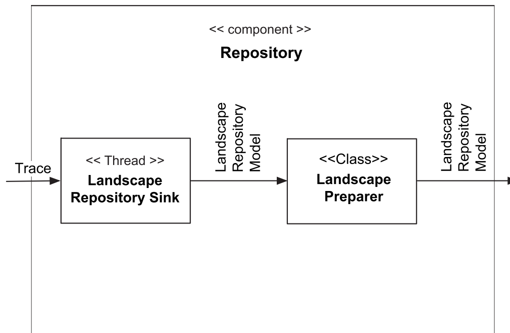


Figure 9.4. Overview of the repository component

the internal model is duplicated and a clone is passed outside of the sink to the *Landscape Preparer*. Only a clone is passed outside since the internal model keeps track of the already discovered entities and just resets the conducted communication calls and number of instances.

The *Landscape Preparer* adds and sets properties of the *Landscape Repository Model* which would slow down the insertion of elements in the *Landscape Repository Sink*. For example, it sets the foundation component for each application and sets the alternating colors of the components. After preparing the landscape model, it is passed to the exchange service. Due to technical reasons, the request to the exchange service for the model actually originates from the client (web browser) which then receives the new or updated landscape model.

9.5 Visualization Provider

After creating or updating the landscape model representing the current software landscape, the next step is visualizing the model. As front end, we utilize web browsers, since live monitoring the landscape should be accessible on different clients without further software installation. Since our application-level perspective is a 3D visualization, we chose Web Graphics

Library (WebGL)[7] as the basic rendering technology. WebGL provides the advantage that no additional plug-ins must be installed and mobile devices can also access our visualization in the same way as desktop PCs do.

Since WebGL requires to write extensive JavaScript code, we use the GWT[8] to be able to write the largest part of our source code in Java. After creating the Java source code, GWT generates the JavaScript code from this source code.

Since GWT utilizes many Java callbacks for the handling of asynchronous events, we furthermore use Xtend[9] to ease the implementation of the callbacks. Xtend is usable in combination with GWT since it actually generates Java classes from the Xtend sources. In summary, the generation chain for most of the implemented classes is: Java source code is generated from the Xtend classes and then this Java source code is translated to JavaScript code.

In Figure 9.5, the components and their relationships taking part in the visualization are shown. The *Main* component triggers the rendering process in the *Engine* component. The *Engine* component sets up the *Landscape Exchange* to periodically poll for new available landscape models. After execution, the *Landscape Exchange* also triggers the *Timeshift* component to fetch the new timeshift history.

When a new landscape model is received, the *Engine* component is responsible for passing the model through several components. At first, the new landscape model is passed to the *Clustering* which generates clusters when packages contain more classes than a configurable threshold value. Then, the *Layout* component calculates the size and position for each drawable model entity. Afterwards, the actual visual entities get created by the *Rendering* component. For example, gray boxes for each system. Finally, the *Interaction* component binds interaction handlers to these visual entities.

Beneath smaller interaction handlers, e.g., opening a package, five larger components can be triggered through interaction means. The first component is the *Code Viewer* which visualizes the source code of the application under study. The second component is the *Database Queries* component. It

[7] http://www.webgl.org

[8] http://www.gwtproject.org

[9] https://eclipse.org/xtend

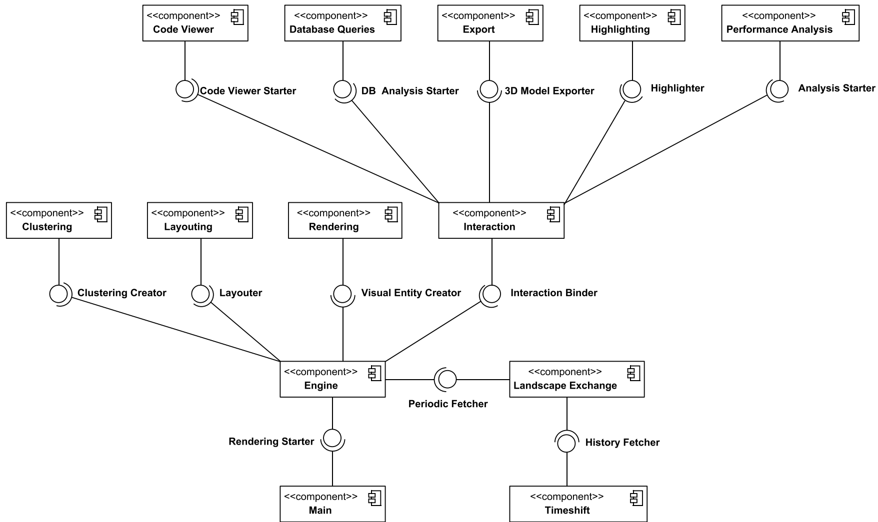


Figure 9.5. Overview of the components involved in the visualization

lists the monitored database SQL queries in a sortable and searchable table. The *Export* component handles the export of the OpenSCAD script used to create the physical 3D models. The *Highlighting* component is responsible for the highlighting of objects, e.g., clicking on a class reveals the in- and outgoing communication lines, and also the replaying of traces is handled in it. The last component is the *Performance Analysis* component which starts the performance analysis mode.

Since all those features are implemented at the client-side and not on the server-side, the browser executes the corresponding JavaScript. Notably this means that all features – also some that might become compute intense such as the layout – are run on the client and not on the server. Thus, the server can focus on analyzing the incoming traces and provides better scalability when more users are connected. A further implication is that the user has the possibility to go offline after receiving the landscape model and is still able to analyze the model.

10

Monitoring and Trace Processing Evaluation: Lab Experiments

As already stated, large software landscapes often consist of hundreds of applications. Monitoring the applications and analyzing the huge amount of resulting data is challenging for the actual monitoring and analysis approach. In this chapter, we evaluate the monitoring and analysis part of our ExplorViz approach with the aim to show their feasibility for monitoring a large software landscape. For this purpose, we split the evaluation into three parts which follow three goals of having a low overhead, providing a live trace processing capability, and being able to scale the monitoring solution with the monitored applications in an elastic fashion using cloud computing.

At first, the goals for the evaluations are defined. Afterwards, three evaluations following the three goals are presented. Finally, we summarize them and evaluate if our approach is feasible for monitoring large software landscapes.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Waller et al. 2014a] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance 2014 (SOSP 2014)*. University of Stuttgart, Nov. 2014
2. [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013
3. [Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015

10.1 Goals

For the evaluation of our monitoring and analysis solution, we define three aspects of our solution as goals which we target to evaluate through several lab experiments. These are detailed in the following.

Mon-G1: Showing Low Overhead

Application-level monitoring imposes an overhead to the actual execution of the program since data, e.g., the called method name, has to be collected during the execution. Since this overhead adds to the response time of an application, it is desirable to keep the monitoring overhead as low as possible.

Mon-G2: Showing Live Trace Processing Capability

Since our approach provides a live trace visualization, the analysis component must provide the capability to analyze the monitoring data on-the-fly and especially should not slow down the actual monitoring phase.

Mon-G3: Showing Scalability and Elasticity

Providing a low overhead monitoring and live trace processing analysis is not sufficient when monitoring large software landscapes. The monitoring and analysis solution still needs to scale with the possible large number of monitored applications. Furthermore, it is desirable to have a cost efficient solution which frees unused resources and therefore provides an elastic solution.

10.2 Monitoring Overhead Evaluation

In [Waller et al. 2014a], we conducted several performance tunings starting on the code base of a Kieker 1.8 nightly release (Kieker 1.8 in the following) and ended with a project written from scratch which imposes the basis of our current monitoring component of ExplorViz.

Table 10.1. Throughput of the ExplorViz′ Monitoring Component (Traces per Second)

	No Instrumentation	Deactivated	Collecting	Writing
Mean	1 190.5k	763.3k	145.1k	141.2k
95% CI	2.0k	4.0k	0.2k	0.3k
Q_1	1 187.4k	747.0k	144.2k	139.4k
Median	1 191.4k	762.5k	146.1k	142.7k
Q_3	1 195.2k	778.4k	146.8k	144.2k

In this section, we show the results of the evaluated monitoring component of ExplorViz in comparison to Kieker 1.8. We utilized MooBench[1] developed by Waller et al. [2014a] as a benchmark to determine the throughput of generated traces per second. MooBench splits the application-level monitoring into four phases and benchmarks each phase. Afterwards, the comparison of the phases reveals which phase adds significant overhead in comparison to the execution without the phase. The gathered data of the evaluation is available online as a dataset package [Waller et al. 2014b].

10.2.1 Throughput of the ExplorViz′ Monitoring Component

Table 10.1 displays the throughput measured by MooBench of our monitoring component. The absolute numbers are tied to the actual hardware configuration which was a X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GB RAM with Solaris 10. Therefore, the comparison between the actual phases is more meaningful than the actual absolute number.

The *Deactivated* phase actually measures the impact of using an instrumentation framework such as AspectJ. The *Collecting* phase adds the actual gathering of information like the called method name. The impact on the overall overhead is rather large in this phase (from 736.3k traces per second

[1] http://kieker-monitoring.net/research/projects/moobench

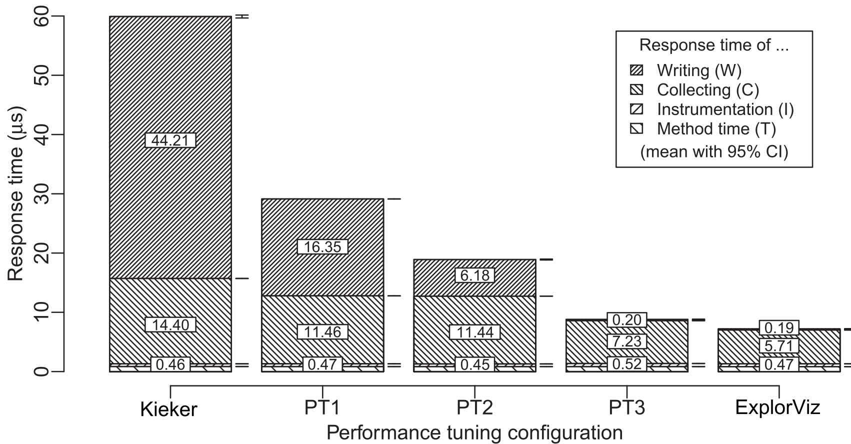


Figure 10.1. Overview of the MooBench benchmark results in response time

down to 145.1k traces per second). The following *Writing* phase, which writes the gathered data to another host using a TCP connection, only negligibly adds to the monitoring overhead of our monitoring component.

10.2.2 Comparison to Kieker

By comparing these values of ExplorViz to the Kieker 1.8 base, we are able to identify differences in the performance of each phase. Furthermore, we can qualitatively evaluate the overhead imposed by the monitoring component of ExplorViz, since other researchers already compared Kieker to other monitoring tools [Eichelberger and Schmid 2014; Waller 2014] and concluded that Kieker imposes a low overhead.

Figure 10.1 shows the results for the conducted performance tunings (PT), the Kieker 1.8 base, and ExplorViz' monitoring component measured in response times. PT1 is similar to the results of Kieker 1.9 [Waller 2014]. PT2 and PT3 show potential versions for future Kieker releases. Since out of scope of the current discussion about monitoring overhead, we refer to [Waller et al. 2014a] for the actual differences in the versions.

In comparison to Kieker 1.8, PT1, and PT2, we achieved a large decrease in the *Collecting* phase with ExplorViz of about 50 % in the response times. Compared to PT3, ExplorViz still provides a decrease of about 20 % in the *Collecting* phase. We attribute the improvements in the *Collecting* phase mainly to our design decision that no Java objects are created for each recorded method call. Instead we directly write the gathered information into a plain array of bytes.

In the performance tunings PT1 and PT2, there is a constant enhancements of the *Writing* phase compared to Kieker 1.8. However, our monitoring component in ExplorViz still outperforms those tunings. In comparison to the base version, we achieved a decrease of about 99.5 % in the response times. Only PT3 also imposes such a large decrease. Therefore, future releases of Kieker might also have such a fast *Writing* phase which was mainly achieved due to enhancing the monitoring with a fast *RingBuffer*.

The total monitoring overhead of Kieker 1.8 is about 59.07 μs. The monitoring component of ExplorViz imposes a monitoring overhead of about 6.37 μs. Therefore, we achieved a decrease of about 89 % in the monitoring overhead. For the throughput, we achieved a speedup of roughly factor 9.

10.3 Live Trace Processing Evaluation

In the former evaluation, we showed that our monitoring component in ExplorViz imposes only low overhead on the monitored application. Since it only involved generating the required information, we add the analysis of this information in this evaluation to show the live trace processing capability of ExplorViz. For replicability and verifiability of our results, the gathered data are available online [Fittkau et al. 2013a]. For reasons of simplicity, when referring to ExplorViz in this section, we mean the monitoring and analysis solution of ExplorViz.

We start by describing our experimental setup. Afterwards, we discuss the results and threats to validity.

10.3.1 Experimental Setup

For our evaluation, we employ an extended version of the monitoring overhead benchmark MooBench [Waller et al. 2014a] which was described in the previous section. In the case of live analysis, we can extend the benchmark's measurement approach for the additional performance overhead of the analysis of each set of monitoring data. Specifically, we can quantify the additional overhead of the phases *Trace Reconstruction* and *Trace Reduction* within our analysis component.

We use two virtual machines (VMs) in our OpenStack[2] private cloud for our experiments. Each physical machine in our private cloud contains two 8-core Intel Xeon E5-2650 (2.8 GHz) processors, 128 GB RAM, and a 500 Mbit network connection. When performing our experiments, we reserve the whole cloud and prevent further access in order to reduce perturbation. The two used VMs are each assigned 32 virtual CPU cores and 120 GB RAM. Thus, both VMs are each fully utilizing a single physical machine. For our software stack, we employ Ubuntu 13.04 as the VMs' operating system and an Oracle Java 64-bit Server VM in version 1.7.0_45 with up to 12 GB of assigned RAM.

MooBench is configured as single-threaded 4 000 000 measured executions with Kieker 1.8. We increased the number of measured executions to 100 000 000 for ExplorViz since it processed the 4 000 000 executions too fast to have reliable results. In each case, we discard the first half of the executions as a warm-up period.

10.3.2 Results and Discussion

First, the throughput results are discussed. Afterwards, the response times are covered.

Throughput The throughput for each phase is visualized in Table 10.2 for Kieker 1.8 and in Table 10.3 for our monitoring and analysis solution in ExplorViz. In both versions, the no instrumentation phase is roughly equal which is to be expected since no monitoring is conducted.

[2] https://www.openstack.org

Table 10.2. Throughput for Kieker 1.8 Monitoring and Analysis (Traces per Second)

	No instr.	Deactiv.	Colle.	Writing	Reconst.	Reduc.
Mean	2 500.0k	1 176.5k	141.8k	39.6k	0.5k	0.5k
95% CI	371.4k	34.3k	2.0k	0.4k	0.001k	0.001k
Q_1	2 655.4k	1 178.0k	140.3k	36.7k	0.4k	0.4k
Median	2 682.5k	1 190.2k	143.9k	39.6k	0.5k	0.5k
Q_3	2 700.4k	1 208.0k	145.8k	42.1k	0.5k	0.5k

Table 10.3. Throughput for our ExplorViz Monitoring and Analysis Solution (Traces per Second)

	No instr.	Deactiv.	Colle.	Writing	Reconst.	Reduc.
Mean	2 688.2k	770.4k	136.5k	115.8k	116.9k	112.6k
95% CI	14.5k	8.4k	0.9k	0.7k	0.7k	0.8k
Q_1	2 713.6k	682.8k	118.5k	102.5k	103.3k	98.4k
Median	2 720.8k	718.1k	125.0k	116.4k	116.6k	114.4k
Q_3	2 726.8k	841.0k	137.4k	131.9k	131.3k	132.4k

ExplorViz manages to generate 770 k traces per second with deactivated monitoring, i. e., the monitoring probe is entered but left immediately. Kieker 1.8 performs significantly better with 1 176 k traces per second. Both versions run the same code for the deactivated phase. We attribute this difference to the change in the number of measured executions with each tool. ExplorViz runs 20 times longer than Kieker 1.8 which might have resulted in different memory utilization. As future work, this observation should be researched by running 100 million method calls with Kieker 1.8 – which takes a few weeks to finish the benchmark.

In the collecting phase, Kieker 1.8 performs 141.8 k traces per second whereby ExplorViz achieves 136.5 k traces per second which is roughly the same with regards to the different number of measured executions of both experiments.

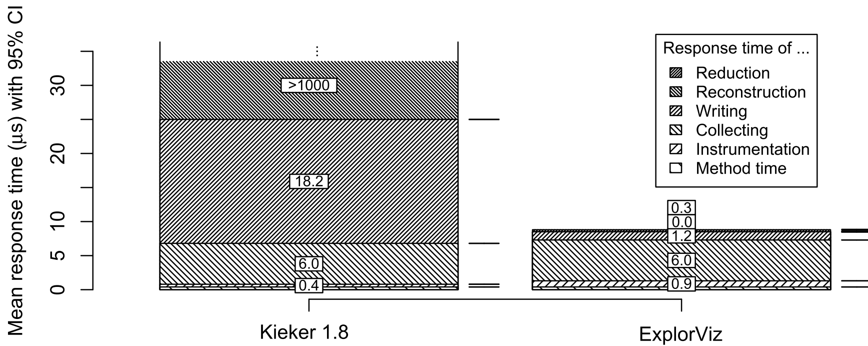


Figure 10.2. Comparison of the resulting response times

ExplorViz reaches 115.8 k traces per second while Kieker 1.8 achieves 39.6 k traces per second in the writing phase. In this phase, our performance tunings take effect. Notably, the trace amount is limited by the network bandwidth in the case of ExplorViz.

In the trace reconstruction phase, Kieker 1.8 performs 0.5 k traces per second and ExplorViz reaches 116.9 k traces per second. We attribute the increase of 1.1 k traces per second in ExplorViz – in comparison to the writing phase – to measuring inaccuracy which is confirmed by the overlapping confidence intervals. In this trace reconstruction phase, ExplorViz performs about 250 times better than Kieker 1.8. We attribute this observation to the old pipes and filters architecture of Kieker 1.8 which has bottlenecks in handling the pipes resulting in poor throughput. Future releases of Kieker will use TeeTime[3] [Wulf et al. 2014] for this purpose. Therefore, the experiment should be conducted again when TeeTime will be integrated in the Kieker releases.

Kieker 1.8 reaches 0.5 k traces per second and ExplorViz achieves 112.6 k traces per second in the reduction phase. Compared to the previous phase, the throughput slightly decreased for ExplorViz which means that the trace summarization slightly slows down the analysis.

[3] http://teetime.sf.net

Response Times In Figure 10.2, the resulting response times are displayed for Kieker 1.8 and ExplorViz in each phase. The response times for the instrumentation is again slightly higher for ExplorViz. In the collecting phase, the response times of both tools are equal (6 µs). Kieker 1.8 has 18.2 µs and ExplorViz achieves 1.2 µs in the writing phase. The comparatively high response times in Kieker 1.8 suggests that the *Monitoring Writer* fails to keep up with the generation of *Monitoring Records* and therefore the buffer to the writer fills up resulting in higher response times. In contrast, in ExplorViz, the writer only needs to send out the *ByteBuffers*, instead of object serialization. In the reconstruction and reduction phases, Kieker 1.8 has over 1 000 µs (in total: 1 714 µs and 1 509 µs), and ExplorViz achieves 0.0 µs and 0.3 µs. The response times of ExplorViz suggest that the filters are efficiently implemented such that the buffers are not filling up. This result is achieved by the utilization of threads for each filter. We attribute the high response times of Kieker 1.8 to garbage collections and the aforementioned bottlenecks in the pipes and filters architecture.

10.3.3 Threats to Validity

We conducted the evaluation only on one type of virtual machine and also only on one specific hardware configuration. To provide higher external validity, other virtual machine types and other hardware configurations should be benchmarked which is future work.

Furthermore, we ran our benchmark on a virtualized cloud environment which might resulted in unfortunate scheduling effects of the virtual machines. We tried to minimize this threat by prohibiting over-provisioning in our OpenStack configuration and assigned 32 virtual CPUs to the instances such that the OpenStack scheduler has to run the virtual machines exclusively on one physical machine.

10.4 Scalability and Elasticity Evaluation

In this section, we present an experiment for evaluating the scalability and elasticity of our application-level monitoring and analysis approach

for monitoring several, scaled JPetStore instances using cloud computing. Stelzer [2014] already conducted a preceding experiment which showed that our worker concept functions for eight JPetStore instances and a firm count of two analysis worker levels. Our evaluation utilizes a dynamic count of worker levels and peeks at 160 JPetStores instances running in parallel.

We start by describing the used workload curve and the experimental setup. Then, the results of the experiment are discussed. At last, we identify threats to validity.

10.4.1 Workload

Our employed workload curve is shown in Figure 10.3. It was derived by monitoring a real web application from a photo service provider [Rohr et al. 2010]. The workload curve represents a day-night-cycle workload pattern which can be considered typical for regional websites. It starts with a rising workload until six o'clock when about 1,000 requests per second are conducted. Then, the load peaks at nine o'clock with about 8,000 requests per second. Afterwards, it slightly decreases to about 7,000 requests per second. In the evening at around eight o'clock p.m., the request count peaks with about 14,000 requests per seconds. Then, it falls to about 1,000 requests per second at midnight and shortly behind this point in time, it drops to no requests for our experiment.

10.4.2 Experimental Setup

We utilize our private cloud running OpenStack containing seven servers. Each server has two Intel Xeon E5-2650 (2.8 GHz, 8 cores) CPUs, 128 GB of RAM, and a 400 GB SSD. Therefore, the total amount of resources are 112 CPU cores, 896 GB of RAM, and 2.8 TB of disc space. Since every core also features Hyper-threading, we configured our cloud to have a maximum of 224 virtual cores (VCPUs).

As object system, we utilize the web application JPetStore[4] written in Java. As the name suggests, it is a software for setting up a small web

[4] http://ibatisjpetstore.sf.net

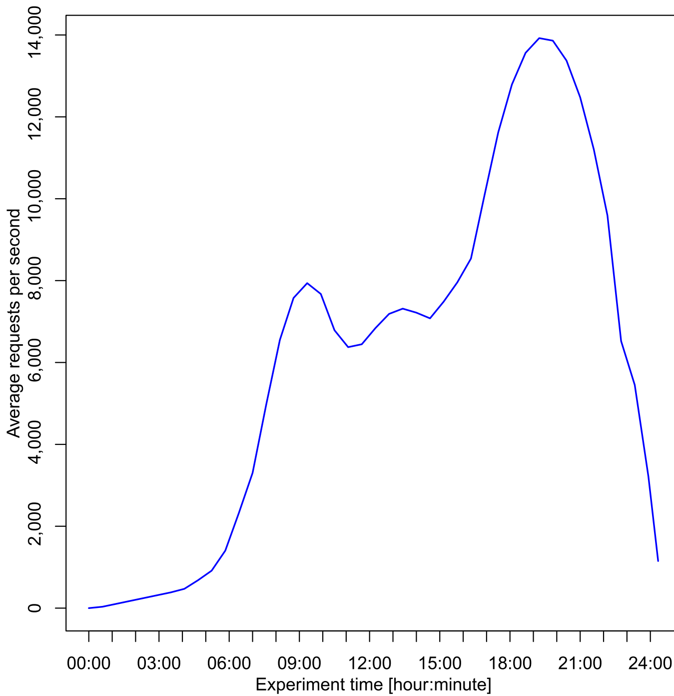


Figure 10.3. Employed workload curve for the evaluation of the sclability and elasticity of our monitoring approach

shop for pets. We monitor all method calls in the *com.ibatis* package which contains source code written by the authors of JPetStore and all method calls in the *org.apache.struts* package which significantly contributes to the generation time of one web page.

Two flavors – resource configurations in OpenStack terms – are used in our experiment. The first one is a small flavor which is used by every dynamically started instance (Master, Worker, and JPetStore nodes). It consists of one virtual CPU (VCPU), 3 GB of RAM, and 10 GB disc space. With this configuration, we are able to start a total count of 224 possible

instances. The second flavor is only used by the capacity manager node. Since this node also contains the `Monitoring LoadBalancer` and generates the workload, it should be guaranteed to have sufficient resources for its tasks. Therefore, the capacity manager node runs with 8 VCPUs, 16 GB of RAM, and 80 GB disc space which reduces the maximum count of dynamically started instances of the small flavor to 216.

A large setup cost for this experiment was imposed by tuning the operating system of the physical server to process the large amount of requests per second. In the default configuration, this request amount is detected as potential denial of service attack and thus the requests are dropped. For example, we had to tune the number of usable TCP ports, TCP state timeouts, the maximum open files, and the NAT connection tracking tables. Some settings also had to be changed on the virtual machine image. For potential replications, our supplementary data package [Fittkau and Hasselbring 2015a] contains the relevant configuration files and all settings we changed. Furthermore, we provide the virtual machine image used for all our instances to reduce the setup costs.

The configuration of our capacity manager `CapMan` contains three scaling groups, i.e., for the Master, the workers (as prototype for dynamically started levels), and the JPetStores. The Master scaling group uses a threshold of 40 % average CPU utilization to trigger the insertion of a new worker level. `CapMan` always calculates the average CPU utilization over a time window of 120 seconds to reduce the impact of short utilization spikes. The prototype of a worker scaling group is configured with a downscaling condition of a value below 15 % average CPU utilization.

A new instance is started if the average CPU utilization is above 45 %. In the JPetStore scaling group, an instance is shut down when the average CPU utilization falls below 27 %. For upscaling, the outstanding requests are counted and when these are above 200, a new instance is started. In contrast to the other scaling groups, the start time of a new instance is not negligible. Therefore, 16 seconds are waited during booting since Jetty must be started and JPetStore must be deployed.

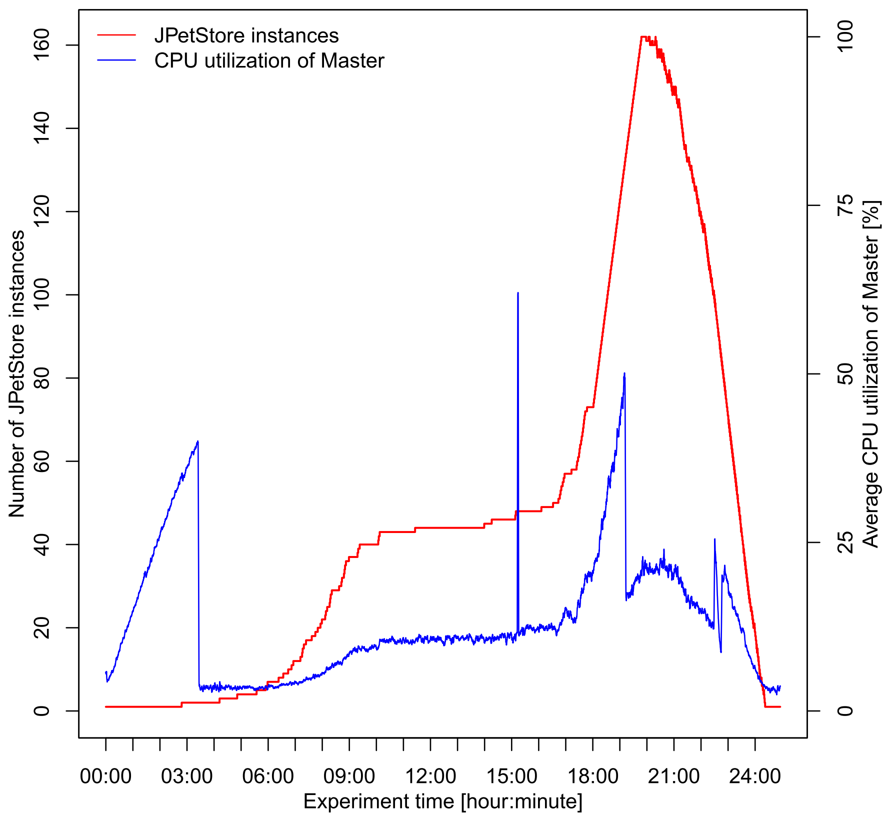


Figure 10.4. JPetStore instance count and average CPU utilization of Master node

10.4.3 Results and Discussion

Figure 10.4 shows the resulting JPetStore instance count and the average CPU utilization of the Master node in our experiment. In general, the count of the JPetStore instances follows the workload curve and peaks at 160 instances. The only exception is the instance count not reducing after the first peak in the workload at hour nine. This is caused by the 27 % average CPU utilization downscaling condition which could be further reduced to also scale down in this situation.

Notably, since the workload curve is reflected in the JPetStore instances, the general scaling in accordance to the imposed workload is functioning. For the evaluation of our elastic monitoring approach, we take a closer look at the average CPU utilization of the Master, the started worker levels, and the monitored method calls per second.

With the constantly rising workload, the CPU utilization of the Master also constantly increases until approximately hour three. At this time, a new worker level is started since the average CPU utilization of the Master rises above 40 %. The started analysis nodes are visualized in Figure 10.5 where this circumstance can also be seen. After the successful insertion of the worker level, the CPU utilization of the Master drops to about 3 %. Notably, at this point in time only two JPetStore instances are started which suffice to utilize our Master node with a 40 % CPU utilization.

After hour three, the CPU utilization of the Master node only rises slightly to 11 % while the JPetStore instance count drastically increases to about 40 instances in hour ten. The work induced by the analysis of the monitoring data is distributed to the workers in the first worker level where the instance count increases to 20 instances till hour ten.

In hour 15, a short peak of about 62 % in the Master CPU utilization can be seen. Since it only occurred for about one minute and has a difference of about 50 % to the previous and following values, this peak is an anomaly. During other runs on our private cloud, we often observed this behavior when another instance is started on the same physical host. Therefore, we implemented a linear anomaly detection algorithm in our capacity manager for this circumstance and thus no new worker level is started in hour 15.

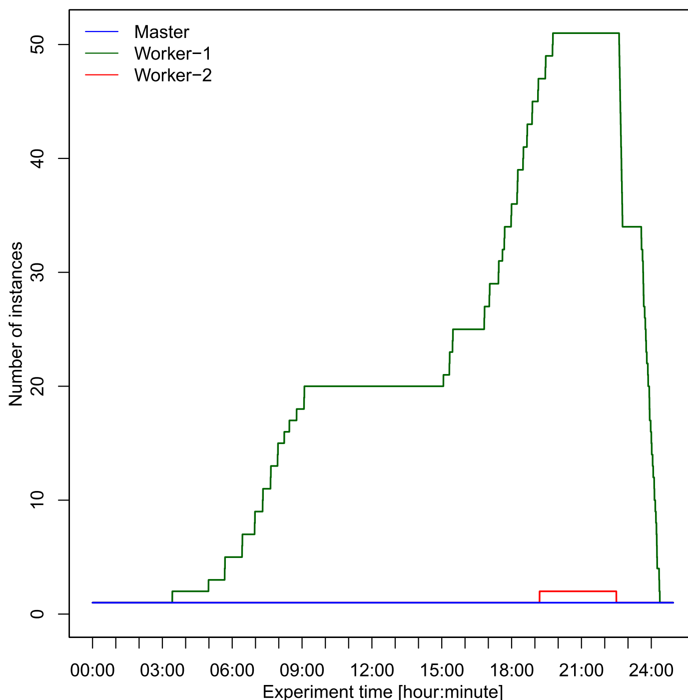


Figure 10.5. Analysis nodes and number of instances in each analysis level

The JPetStore instance count is rising again from hour 15 till hour 20 peaking in 160 instances. Therefore, the instance count of the workers in the first worker level is also increasing which peaks at about 50 instances in hour 20. Since the Master has to receive and merge the traces from those instances, its CPU utilization also rises until hour 19. Then, the CPU utilization is once again above the 40 % threshold which results in a newly inserted worker level. Afterwards, the Master CPU utilization drops to about 17 %. This drop is not as large as the previous one but still it circumvents the overutilization of the Master node.

In hour 20, the workload approximately decreases until hour 24. This leads to a reduction of the JPetStore instances and therefore, also the analysis

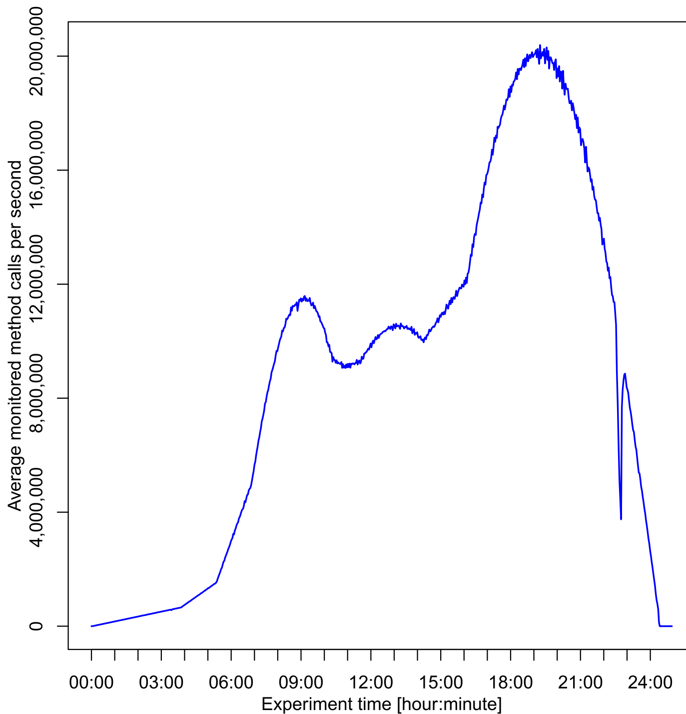


Figure 10.6. Average monitored and analyzed method calls per second

nodes are reduced. At first, the second worker level is completely removed in hour 22 resulting in an increase of the CPU utilization on the Master node. The worker instances in the first worker level are also reduced until hour 24 is reached. Then, also the first worker level is removed resulting in the initial configuration where only the Master node is analyzing the monitored data.

Figure 10.6 visualizes the monitored and analyzed method calls per second. In general, it follows the requests per second of the workload and peaks in about 20 million analyzed method calls per second. The only

exception is a short spike in hour 22. This resulted from a too fast shutdown of one analysis node in the second worker level which can be circumvented by increasing the shutdown delay of analysis nodes in higher worker levels.

10.4.4 Threats to Validity

We conducted our experiment on our private cloud with the scaling of JPetStore instances. For external validity, it should also be evaluated in other environments and with other applications. The same applies to the employed workload curve and the amount of conducted monitoring.

Our experiment involved two worker levels since only 216 VCPUs were available. The results for a third worker level might be different. Further experiments are required to show whether the third worker level still circumvents the overutilization of the Master node.

Furthermore, similar traces are generated by accessing JPetStore. We assume that our monitoring approach will behave differently if this assumption is not satisfied. This should be also investigated in further experiments.

10.5 Summary

From our monitoring overhead evaluation, we conclude that the monitoring component of ExplorViz provides low overhead, since Kieker has already been shown to have low overhead in comparison to other monitoring tools [Eichelberger and Schmid 2014; Waller 2014]. With our monitoring component of ExplorViz, we were able to even speedup the maximal trace monitoring capability by a factor of 9 and decreased the overhead by 89 %.

ExplorViz outperformed the employed Kieker 1.8 version with a speedup of about factor 250 when adding an analysis of the gathered monitoring information. Furthermore, the impact on the throughput when adding analysis steps was negligible small. Therefore, the analysis only negligibly impacts the monitoring component. Thus, we conclude that our monitoring and analysis approach is capable of providing the required live trace processing for our approach.

Summarizing the results of our scalability and elasticity evaluation, our distributed application-level monitoring approach showed feasible to circumvent the overutilization of the Master node in spite of a rising workload. Furthermore, the Master node employs only a single VCPU. Therefore, during low workload on the monitored applications, the minimum costs for monitoring incur. Thus, our monitoring and analysis approach showed its scalability and elasticity capability.

Concluding, we successfully evaluated each requirement for the monitoring and analysis part to enable live trace visualization. Therefore, our approach showed feasible to monitor several applications in a large software landscape and to analyze the resulting traces on-the-fly as required for the visualization.

11

Visualization Evaluation: Controlled Experiments

In this chapter, we present an evaluation of our visualization approach. We split the visualization evaluation into three parts, namely the evaluation of the application perspective, the physical 3D models, and the landscape perspective. If we would have conducted an all-in-one evaluation, we would have required an approach that is comparable as the whole. To the best of our knowledge, there is no such approach combining landscape and application perspective in a similar fashion. Therefore, we would have had to combine other approaches into one tool, which imposes several threats to validity on its own. Thus, we compare each part of our visualization approach to a comparable, already established visualization.

After defining the goals, we start by describing a controlled experiment and its replication where we compare our application perspective to the trace visualization tool Extravis. The next section describes a controlled experiment for evaluating our physical 3D models approach by comparing them to on-screen models. Then, we present a controlled experiment to evaluate the landscape perspective by comparing it with an APM tool visualization. In the last section, we briefly outline the qualitative evaluation of our VR approach and conclude with a summary.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. IEEE, May 2015
2. [Fittkau et al. 2015j] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. Technical report 1507. Department of Computer Science, Kiel University, Germany, June 2015
3. [Fittkau et al. 2015h] F. Fittkau, A. Krause, and W. Hasselbring. Hierarchical software landscape visualization for system comprehension: a controlled experiment. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

11.1 Goals

To evaluate new visualization approaches, they should be compared to already existing ones to show that they provide actual benefits in comparison to the old ones. Important benefits in the context of program and system comprehension are an increased efficiency and/or effectiveness in solving comprehension tasks. Empirical studies, such as controlled experiments, are required to assess them.

As major goal of our evaluations, we target to show that our ExplorViz visualization is more efficient and effective for solving program and system comprehension tasks in comparison to already established approaches. To achieve this goal, each experiment defines hypotheses as subgoals which are described in the corresponding sections. Furthermore, the usability of our ExplorViz tool – including stability of our tool – is implicitly evaluated by the diverse participants in our controlled experiments.

11.2 Application Perspective Evaluation

In this section, we present a controlled experiment comparing our trace visualization ExplorViz on the application perspective with the already established visualization Extravis by Cornelissen et al. [2009] in the context of program comprehension. Therefore, we defined typical program comprehension tasks and used PMD[1] as object system. To validate our results, we replicated [Lindsay and Ehrenberg 1993] our experiment by conducting a second controlled experiment using a different-sized object system named Babsi.[2] We measured the *time spent* and the *correctness* for each task. These measures are typically used in the context of program comprehension [Rajlich and Cowan 1997]. After the experiments, we analyzed the benefits of using Extravis with circular bundling or ExplorViz with the city metaphor on the defined tasks.

To facilitate the verifiability and reproducibility for further replications [Crick et al. 2014], we provide a package containing all our experimen-

[1] `http://pmd.sourceforge.net`

[2] `http://babsi.sourceforge.net`

tal data. It contains the employed version of ExplorViz v0.5-exp (including source code and manual), input files, tutorial materials, questionnaires, R scripts, datasets of the raw data and results, and 80 screen recordings of the user sessions. We explicitly invite other researches to compare their trace visualizations with ExplorViz and we provide as complete material as possible to lower the effort for setting up similar experiments. The package is available online [Fittkau et al. 2015c] with source code under the Apache 2.0 License and the data under a Creative Commons License (CC BY 3.0).

After presenting the used visualizations, we describe both controlled experiments by their design, operation, data collection, analysis, results, discussion, and threats to validity. Afterwards, we share lessons learned and give advice on avoiding some observed challenges.

11.2.1 Used Visualizations

Figure 11.1 displays the used Extravis visualization for our experiments. For the description of the semantics, we refer to Section 3.3.

In Figure 11.2, the employed application perspective visualization of ExplorViz version 0.5 is shown. In contrast to the current visualization, the height of the packages and buildings is slightly smaller. However, this change had cosmetic reasons for the physical 3D models and should not invalidate the results for the current visualization. Furthermore, we have disabled the trace replayer and the time shifting for the experiments since there are no similar features in Extravis. Similarly to the these features, we have disabled the source code viewer for the experiments due to its absence in Extravis. Since we only utilize the application perspective, we have disable the landscape perspective for this experiment.

11.2.2 Experimental Design

In addition to general software engineering experimentation guidelines [Kitchenham et al. 2002; Jedlitschka and Pfahl 2005; Di Lucca and Di Penta 2006; Di Penta et al. 2007; Sensalire et al. 2009], we follow the experimental designs of Wettel et al. [2011] and of Cornelissen et al. [2009]. Similar to these experiments, we use a *between-subjects* design. Thus, each subject only

Figure 11.1. The recorded execution trace of PMD for the first controlled experiment visualized in Extravis

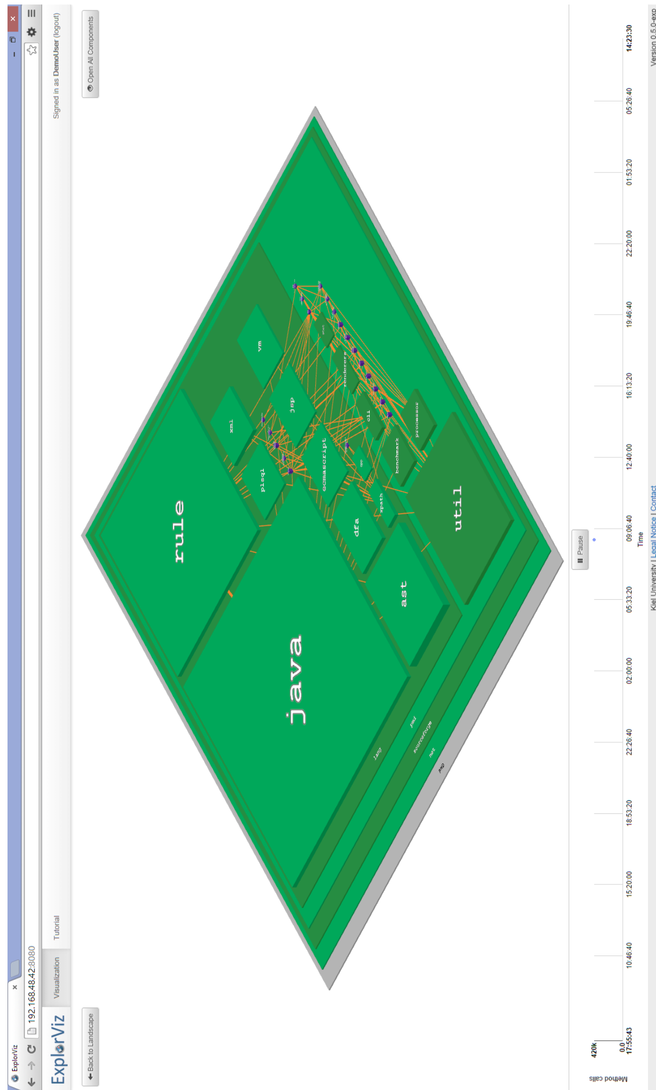


Figure 11.2. The recorded execution trace of PMD for the first controlled experiment represented in ExplorViz v0.5

solves tasks with either Extravis or ExplorViz and therefore, uses one tool only. Following the GQM approach [Basili and Weiss 1984], we define the goal of our experiments as quantifying the impact of using either Extravis or ExplorViz for program comprehension.

Research Questions & Hypotheses

We define three application perspective research questions (App-RQ) for the comparison:

- **App-RQ1:** What is the ratio between Extravis and ExplorViz in the *time required for completing* typical program comprehension tasks?
- **App-RQ2:** What is the ratio between Extravis and ExplorViz in the *correctness of solutions* to typical program comprehension tasks?
- **App-RQ3:** Which typical sources of error exist when solving program comprehension tasks with Extravis or ExplorViz?

Accordingly, we formulate two null hypotheses:

- **App-H1$_0$:** There is no difference between Extravis and ExplorViz in the *time spent* for completing typical program comprehension tasks.
- **App-H2$_0$:** The *correctness* of solutions to typical program comprehension tasks does not differ between Extravis and ExplorViz.

We define the following alternative hypotheses:

- **App-H1** Extravis and ExplorViz require different times for completing typical program comprehension tasks.
- **App-H2** The correctness of solutions to typical program comprehension tasks differs between Extravis and ExplorViz.

For App-RQ3, we conduct an in-depth analysis of the results and analyze the recorded sessions of each subject in detail.

Dependent and Independent Variables

The independent variable in both experiments is the employed tool used for the program comprehension tasks, i.e., Extravis or ExplorViz. We measured the accuracy (*correctness*) and response time (*time spent*) as dependent variables. These are usually investigated in the context of program comprehension [Wettel et al. 2011; Rajlich and Cowan 1997; Cornelissen et al. 2009].

Treatment

The control group used Extravis to solve the given program comprehension tasks. The experimental group solved the tasks with ExplorViz.

Tasks

For our task definitions, we stuck to the framework of Pacione et al. [2004] which describes categories of typical program comprehension tasks. It focuses on dynamic analysis [Wettel et al. 2011] providing a good match for our task definitions. In addition, Cornelissen et al. [2009] used this framework.

For our first experiment, we selected a medium to large-sized object system and adhered to the tasks defined by Cornelissen et al. [2009] as close as possible to prevent bias toward ExplorViz. Preliminary experiments with their object system Checkstyle revealed only a small amount of used classes (41). PMD provides similar functionality to Checkstyle, i.e., reporting rule violations on source code. Analyzing a single source code file (`Simple.java` by Cornelissen et al. [2009]) with the default `design.xml` of PMD version 5.1.2 revealed 279 used classes and 421,239 method calls in the resulting trace.

Table 11.1 shows our tasks including their context and achievable maximum points. We adapted the tasks from Cornelissen et al. [2009] to the context of PMD. Notably, we dismissed two original tasks to restrict our experiment to one hour. However, the dismissed tasks are redundant to the remaining tasks with regard to the program comprehension activity categories which are still completely covered. The categories (A1 to A9)

Table 11.1. Description of the Program Comprehension Tasks for the First Application Perspective Experiment (PMD)

ID	Category	Description	Score
		Context: Identifying refactoring opportunities	
App-T1	A{4,8}	Name three classes (from different packages) that have high fan-in (at least 4 incoming communications) and almost no fan-out (outgoing communication).	3
		Context: Understanding the checking process	
App-T2.1	A{3,4,5}	Write down all constructor/method calls between `RuleChain` and `JavaRuleChainVisitor`.	3
App-T2.2	A{1,2,5,6}	In general terms, describe the lifecycle of `GodClassRule`: Who creates it, what does it do (on a high level)?	3
		Context: Understanding the violation reporting process	
App-T3.1	A{1,5}	Which rules are violated by the input file using the design rule set? Hint: Due to dynamic analysis the violation object is created only for those cases.	2
App-T3.2	A{1,3}	How does the reporting of rule violations work? Where does a rule violation originate and how is it communicated to the user? Write down the classes directly involved in the process. Hint: The output format is set to HTML.	4
		Context: Gaining a general understanding	
App-T4	A{1,7,9}	Starting from the Mainclass `PMD` – On high level, what are the main abstract steps that are conducted during a PMD checking run. Stick to a maximum of five main steps. Hint: This is an exploration task to get an overview of the system. One strategy is to follow the communication between classes/packages. Keep the handout of PMD in mind.	5

Table 11.2. Description of the program comprehension activity categories defined by and taken from [Pacione et al. 2004]

Category	Description
A1	Investigating the functionality of (a part of) the system
A2	Adding to or changing the system's functionality
A3	Investigating the internal structure of an artifact
A4	Investigating dependencies between artifacts
A5	Investigating runtime interactions in the system
A6	Investigating how much an artifact is used
A7	Investigating patterns in the system's execution
A8	Assessing the quality of the system's design
A9	Understanding the domain of the system

are shown in Table 11.2. All tasks were given as open questions to prevent guessing. In addition, we changed the order of the tasks compared to Cornelissen et al. [2009] since in our experiment no source code access was provided. Our task set starts with less complex tasks (identifying fan-in and fan-out) and ends with complex exploration tasks. This enabled users to get familiar with the visualization in the first tasks and raises the level of complexity in each following task.

To validate our results, we conducted a second controlled experiment as replication. It investigated the influence of the object system's size and its design on the results. The visualization of the city metaphor is usually more affected by these factors than using the circular bundling approach. Therefore, we selected a small-sized and not well designed object system. Both criteria are met by Babsi written by undergraduate students. Babsi is an Android app designed to support pharmacists in supervising the prescription of antibiotics. The execution trace generated for our second experiment utilizes all 42 classes and contains 388 method calls.

The tasks for our replication are given in Table 11.3. To enable comparisons of the subjects' performance in our experiments, we kept the tasks as similar as possible. Notably, there is no task similar to App-T3.1 from our PMD experiment and hence we omitted it in the replication.

Population

We used the computer science students' mailing lists of the Kiel University of Applied Sciences (FH Kiel) and Kiel University (CAU Kiel) to recruit subjects for our PMD experiment. 30 students have participated in the experiment (6 students from FH Kiel and 24 students from CAU Kiel). Our replication was conducted with 50 students recruited from the CAU Kiel course "Software Project" in summer term 2014 with no overlapping participants.

As motivation, they participated in a lottery for one of five gift cards of 50 €. Additionally, the best three performances received a certificate. The students in the replication had the additional motivation of supporting their task of understanding the software (Babsi) to be used in their course.

The subjects were assigned to the control or experimental groups by random assignment. To validate the equal distribution of experiences, we asked the subjects to perform a self-assessment on a 5-point Likert Scale [Likert 1932] ranging from 0 (no experience) to 4 (expert with years of experience) before the experiment. The average programming experience in the control group was 2.33 versus 2.46 in the experimental group. Their experience with dynamic analysis was 0.41 and 0.69, respectively. Due to the similarity of the self-assessed results, we conclude that the random assignments resulted in a similarly distributed experience between both groups. The same holds for our replication (Java experience: 1.68 and 1.79; dynamic analysis experience: 0.28 and 0.25).

11.2.3 Operation

In the following, we detail the operation of our experiments.

Generating the Input

We generated the input for ExplorViz directly from the execution of the object systems. ExplorViz persists its data model into files which act as a replay source during the experiments. Extravis requires files conforming to the Rigi Standard Format (RSF). To the best of our knowledge, there were no suitable RSF exporter tool for traces of our Java-based object systems. Therefore, we implemented such an exporter in ExplorViz.

Two traces were generated for PMD. The configuration of PMD is conducted in the first trace while the rule checking is performed in the second trace. Both traces are equally important for program comprehension. However, Extravis is limited to visualize only one trace at a time. Thus, we had to concatenate the two generated traces. Alternatively, the users of Extravis could have manually loaded each trace when needed. However, this would have hindered the comparison between Extravis and ExplorViz. Similar circumstances applied to our replication.

Tutorials

We provided automated tutorials for Extravis and ExplorViz where all features were explained. This enhanced the validity of our experiments by eliminating human influences. For ExplorViz, we integrated a guided and interactive tutorial. Since Extravis is not open-source, we could only provide an illustrated tutorial where the user is not forced to test the functionality. However, we advised the subjects in the control group to interactively test it. Subsequent evaluation of the user recordings showed that all of the subjects have adhered to our advice.

Questionnaire

We used an electronic questionnaire rather than sheets of paper. An electronic version provides several advantages for us. First, the timings for each task are automatically recorded and time cheating is impossible. Second, the user is forced to input valid answers for some fields, e.g., perceived difficulty in the debriefing part. Third, we omit a manual and error-prone digitalization of our results.

Table 11.3. Description of the Program Comprehension Tasks for our Application Perspective Replication (Babsi)

ID	Category	Description	Score
		Context: Identifying refactoring opportunities	
App-RT1	A{4,8}	Name three classes that have high fan-in (at least 3 incoming communications) and almost no fan-out (outgoing communication).	3
		Context: Understanding the login process	
App-RT2.1	A{3,4,5}	Write down all constructor/method calls between `gui.MainActivity` and `comm.Sync`.	3
App-RT2.2	A{1,2,5,6}	In general terms, describe the lifecycle of `data.User`: Who creates it, how is it used? Write down the method calls.	3
		Context: Understanding the antibiotics display process	
App-RT3	A{1,3}	How does the display of antibiotics work? Where and how are they created? Write down the classes directly involved in the process.	6
		Context: Gaining a general understanding	
App-RT4	A{1,7,9}	Starting from the Mainclass `gui.MainActivity` - What are the user actions (e.g., Login and Logout) that are performed during this run of Babsi. Write down the classes of the activities/fragment for each user action. Stick to a maximum of seven main steps (excluding Login and Logout). Hint: This is an exploration task to get an overview of the system. One strategy is to follow the communication between classes.	7

Pilot Study

Before the actual controlled experiment, we conducted a small scale pilot study with experienced colleagues as participants. According to the received feedback, we improved the material and added hints to the tasks which were perceived as too difficult. In addition, a red-green-color-blind impaired colleague used both visualizations to asses any perception difficulties. In the case of ExplorViz, existing arrows in addition to the colors for showing communication directions were sufficient. In the case of EXTRAVIS, we added a tutorial step to change the colors.

Procedure

Both experiments took place at the Kiel University. For the first experiment, each subject had a single session. Therefore, most subjects used the same computer. Only in rare cases, we assigned a second one to deal with time overlaps. In our replication, six to eight computers were concurrently used by the participants in seven sessions. In preliminary experiments, all systems provided similar performance. In all cases, the display resolution was 1920x1080 or 1920x1200.

Each participant received a short written introduction to PMD/Babsi and was given sufficient time for reading before accessing the computer. The subjects were instructed to ask questions in case of encountered challenges at all times. Afterwards, a tutorial for the respective tool was started. Subsequently, the questionnaire part was started with personal questions and experiences, followed by the tasks, and finally debriefing questions.

The less complex tasks (App-T1, App-T2.1, App-T3.1, App-RT1, App-RT2.1) have a time allotment of 5 minutes, while the more complex tasks (App-T2.2, App-T3.2, App-T4, App-RT2.2, App-RT3, App-RT4) have 10 minutes. The elapsed time was displayed during the task and highlighted when reaching overtime. The subjects were instructed to adhere to this timing but were not forced to do so.

11.2.4 Data Collection

In addition to personal information and experience, we collected several data points during our experiments.

Timing and Tracking Information

All our timing information was automatically determined within our electronic questionnaire. Furthermore, we recorded every user session using a screen capture tool (*FastStone Capture*[3]). These recordings enabled us to reconstruct the user behavior and to look for exceptional cases, e.g., technical problems. In the case of such problems, we manually corrected the timing data.

Correctness Information

We conducted a blind review process due to the open questions format. First, we agreed upon sample solutions for each task. A script randomized the order of the answers of the subjects. Thus, no association between answers and group was possible. Then, both reviewers evaluated all solutions independently. Afterwards, the arising small discrepancies of maximal 1 point were discussed.

Qualitative Feedback

The participants were asked to give suggestions to improve their used tool. We restrict ourselves to listing the three most mentioned suggestions for each tool. Ten participants suggested hiding not related communication lines when marking a class in Extravis. Four users missed a textual search feature, which is not available in Extravis and ExplorViz, and four other users suggested that the performance of fetching called methods should be improved. In the case of ExplorViz, ten subjects suggested to resolve the overlapping of communication lines. Seven users found it difficult to see class names due to overlapping. Five users wished for an opening window containing a list of method names when clicking on communication lines.

[3] http://www.faststone.org/FSCaptureDetail.htm

Table 11.4. Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) for PMD Experiment

	PMD			
	Time Spent		**Correctness**	
	EXTRAVIS	ExplorViz	EXTRAVIS	ExplorViz
mean	47.65	34.27	8.42	13.58
difference		-28.06%		+61.28%
sd	9.96	3.14	4.29	2.46
min	23.04	29.43	3	4
median	48.89	33.84	7	14
max	65.07	38.99	16	18
Analyzed users	12	12	12	12
Shapiro-Wilk W	0.8807	0.9459	0.9055	0.9524
Levene F		2.4447		2.0629
Student's t-test				
df		22		22
t		4.4377		-3.6170
p-value		0.0002		0.0015

11.2.5 Analysis and Results

Table 11.4 and Table 11.5 provide descriptive statistics of the overall results related to time spent and correctness for each experiment.

We removed the users with a total score of less than three points from our analysis. This effected five users for our first experiment, i.e., three users from the control group and two users from the experimental group. A single user from the experimental group of our second experiment was effected. In total, three users did not look at the object systems. Hence, they guessed all answers. Two users did not use the "Show full trace" feature in EXTRAVIS, thus analyzing only 0.02% of the trace. One user did not take any look at method names as required for the tasks. For similar reasons, one user for our first experiment and two users in our replication had missing values and are omitted from the overall results but included in the single results.

Table 11.5. Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) for Replication

	Babsi			
	Time Spent		**Correctness**	
	Extravis	ExplorViz	Extravis	ExplorViz
mean	31.55	29.14	9.40	13.04
difference		-7.64%		+38.72%
sd	7.25	6.48	3.60	3.23
min	18.94	19.38	3	6
median	31.27	27.19	9	13.5
max	43.20	41.56	18	18
Analyzed users	24	23	24	23
Shapiro-Wilk W	0.9618	0.9297	0.9738	0.9575
Levene F		0.4642		0.0527
Student's t-test				
df		45		45
t		1.2006		-3.6531
p-value		0.2362		0.0007

In Task App-T3.1, most users searched for a non-existing class *design file* before giving up. This hints at an ambiguous task. Thus, we removed Task App-T3.1 from our overall analysis. We use the two-tailed Student's t-test which assumes normal distribution. To test for normal distribution, we use the Shapiro-Wilk test [Shapiro and Wilk 1965] which is considered more powerful [Razali and Wah 2011] than, for instance, the Kolmogorov-Smirnov test [Pearson and Hartley 1972]. To check for equal or unequal variances, we conduct a Levene test [Levene 1960]. For all our analysis tasks, we used the 64-bit R package in version 3.1.1.[4] In addition to the standard packages, we utilize `gplots` and `lawstat` for drawing bar plots and for importing Levene's test functionality, respectively. Furthermore, we chose $\alpha = .05$ to check for significance in our results. The raw data, the R scripts, and our results are available as part of our experimental data package [Fittkau et al. 2015c].

[4] `http://www.r-project.org`

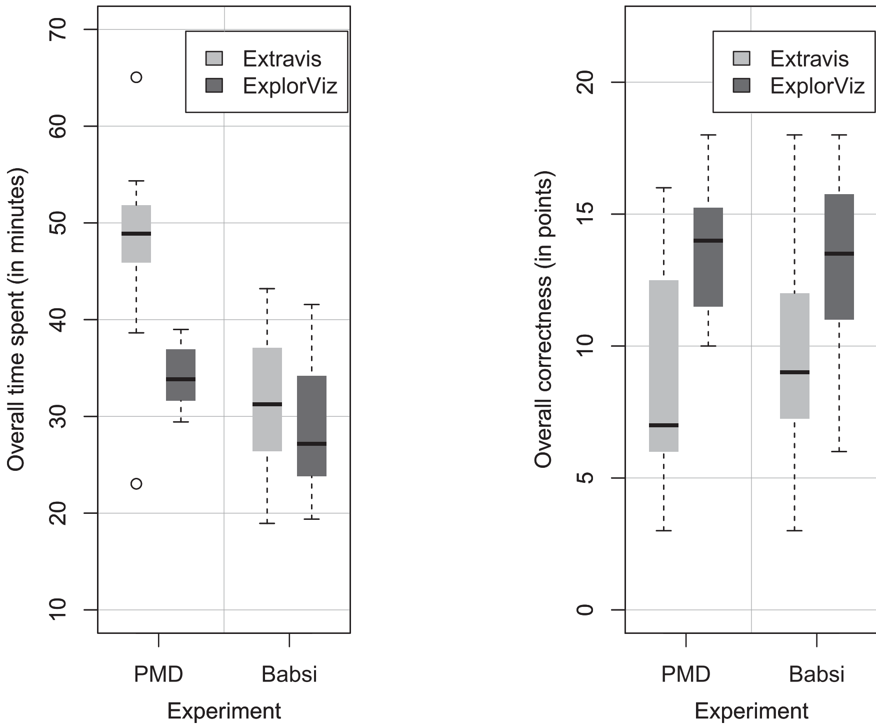


Figure 11.3. Overall time spent and correctness for both experiments

App-RQ1 (Time Spent)

We start by checking the null hypothesis App-$H1_0$ which states that there is no difference in time spent between EXTRAVIS and ExplorViz for completing typical program comprehension tasks. Figure 11.3, left-hand side, displays a box plot for the time spent in both experiments. In Table 11.4 and Table 11.5 the differences between the mean values of EXTRAVIS and ExplorViz for each experiment are shown. For our first experiment, the ExplorViz users required 28.06% less total time for completing the tasks and in our replication they required 7.64% less total time.

The Shapiro-Wilk test for normal distribution in each group and each experiment succeeds and hence we assume normal distribution. The Levene test also succeeds in both experiments and hence we assume equal variances between both groups.

The Student's t-test reveals a probability value of 0.0002 in our first experiment which is lower than the chosen significance level. Therefore, the data allows us to reject the null hypothesis App-$H1_0$ in favor of the alternative hypothesis App-H1 for our first experiment. Thus, there is a significant difference in timings (-28.06%) between Extravis and ExplorViz (t-test t=4.4377, d.f. = 22, P = 0.0002).

In the replication, the Student's t-test reveals a probability value of 0.2362 which is larger than the chosen significance level and we fail to reject the null hypothesis in this case.

App-RQ2 (Correctness)

Next, we check the null hypothesis App-$H2_0$ which states that there is no difference in correctness of solutions between Extravis and ExplorViz in completing typical program comprehension tasks.

Figure 11.3, right-hand side, shows a box plot for the overall correctness in both experiments. Again, Table 11.4 and Table 11.5 shows the differences between the mean values of each group and each experiment. For our first experiment, the ExplorViz users achieve a 61.28% higher score and in our replication they achieve a 38.78% higher score than the users of Extravis.

Similar to App-RQ1, the Shapiro-Wilk and Levene tests succeed for both experiments. The Student's t-test reveals a probability value of 0.0015 for our first experiment and 0.0007 for our replication which is lower than the chosen significance level in both cases. Therefore, the data allows us to reject the null hypothesis App-$H2_0$ in favor of the alternative hypothesis App-H2 for both experiment. Hence, there is a significant difference in correctness (+61.28% and +38.72%) between the Extravis and ExplorViz groups (t-test t=-3.6170, d.f. = 22, P = 0.00015 and t-test t=-3.6531, d.f. = 45, P = 0.0007).

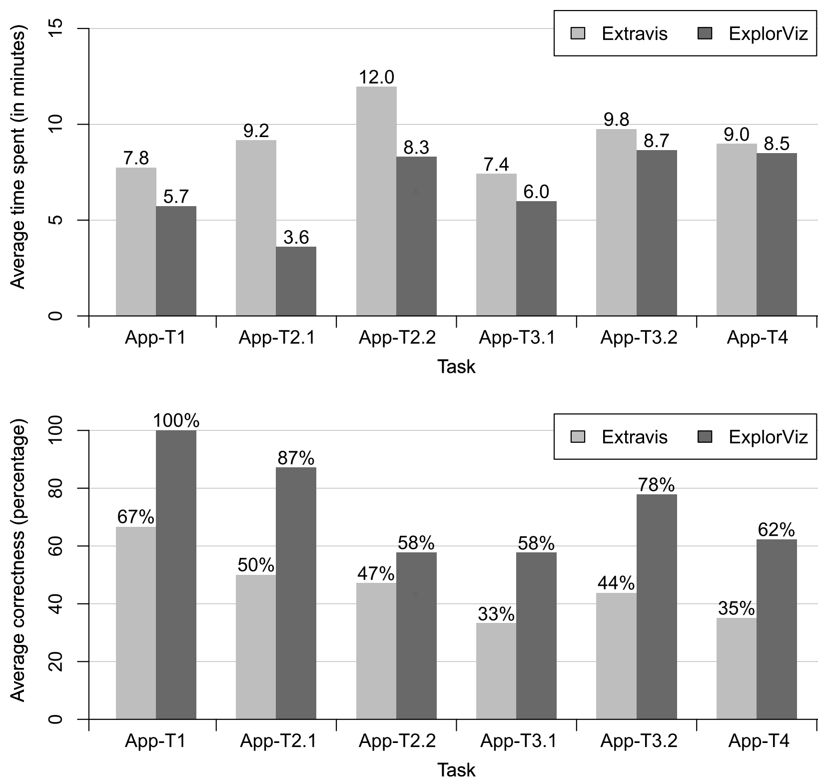


Figure 11.4. Average time spent and correctness per task for PMD experiment

11.2.6 Discussion

The time spent in our first experiment is significantly lower using ExplorViz. The results for the time spent in our replication are not significant, hence there is no statistical evidence for difference in the time spent. However, due to the median of 31 minutes for Extravis and 27 minutes for ExplorViz, the box plot, and the fact that our first experiment had a significant difference in time spent, it is unlikely that the time spent with Extravis is much less than with ExplorViz. Therefore, we can interpret our results such that using

either EXTRAVIS or ExplorViz had only a negligible effect in time spent in the replication. Thus, we focus on the correctness in our following discussion.

The analysis of the results reveals a significant higher correctness for users of ExplorViz in both experiments. We conclude that the effect of using ExplorViz for solving typical program comprehension tasks leads to a significant increase in correctness and in less or similar time spent on the tasks in comparison to using EXTRAVIS.

We conducted an in-depth analysis of each user recording to investigate the reasons for the drawn conclusions. We focus on our PMD experiment and briefly describe any differences in our replication. The results for each task are shown in Figure 11.4. For our replication, the graphics for each task are contained in our provided experimental data package. We omit the discussion of Task App-T3.1 due to its unclear task description.

App-T1

Most EXTRAVIS users investigated the classes with incoming red lines. They evaluated each of the found classes for the amount of incoming connections by counting the method calls. This was hindered by the large amount of displayed method calls and hence they had to restrict the shown trace using the massive sequence view. A source of error was the used color coding. Some users confused the incoming color and the outgoing color, i.e., they searched for green lines. In the smaller object system of our replication, the method call counting was easier.

Most of the ExplorViz users used random searching for a class. However, the amount of incoming and outgoing method calls of a class was directly perceivable.

App-T2.1

The EXTRAVIS users searched for the first class by closing unrelated packages. Then, the second class was searched. Due to the large amount of method calls, the users had to filter them with the massive sequence view. The last call `visitAll` was sometimes missed due to being a thin line in the massive sequence view.

In ExplorViz, the users searched for the first class and marked it. For the second class, the users opened two packages and hovered over the communication to get the method names. With both tools, the users were faster due to a smaller object system in our replication.

App-T2.2, App-T3.2, App-T4

Each of these three tasks required an exploration of the system. For Extravis, the users started at the class from the task assignment. They marked the class to view its communication and used the massive sequence view to filter it. Therefore, they often divided the trace into arbitrary sequences. Then, the users investigated the method names in each sequence. During this process, some users skipped sequences since Extravis provides no hints on already viewed sequences. This resulted in misses of method calls and called classes.

The ExplorViz users started with the class described in the task. They looked at the incoming and outgoing method calls. Upon finding a method name of interest, they marked the called class and investigated further communication.

Summary of the Discussion

In our experiments, the massive sequence view of Extravis led to missing method calls, if the trace is large. This is caused by missing hints on already viewed sequences and by single method calls being visualized by thin lines. Furthermore, the color coding of directions became a source of error. We attribute this circumstance to easily forgettable semantics of color coding while concentrating on a task. Thus, some users had to regularly look up the semantics.

Some users of ExplorViz had difficulties in hovering the communication lines. Some users tried to hover the triangles used for displaying the communication direction instead of the required lines. Furthermore, the overlapping method names and communication lines resulted in taking more time to get the required information.

11.2.7 Threats to Validity

In this section, we discuss the threats to internal and external validity [Wohlin et al. 2012; Shadish et al. 2002; Juristo and Moreno 2010] that might have influenced our results.

Internal Validity

Subjects The subjects' experience might not have been fairly distributed across the control group and the experimental group. To alleviate this threat, we randomly assigned the subjects to the groups which resulted in a fair self-assessed experience distribution as described in Section 11.2.2.

Another threat is that the subjects might not have been sufficiently competent. Most participants stated regular or advanced programming experience in our first experiment involving PMD. For the replication, most subjects stated beginner or regular programming experience which should be sufficient for the small-sized object system Babsi.

The subjects might have been biased towards ExplorViz since the experiments took place at the university where it is developed. To mitigate this threat, the subjects did not know the goal of the experiments and we did not tell them who developed which tool.

A further threat is that the subjects might not have been properly motivated. In our first experiment, the participants took part voluntarily. Furthermore, we encountered no unmotivated user behavior in the screen recordings – except for the subjects scoring below three points that we have excluded from our analysis (Section 11.2.5). Therefore, we assume that the remaining subjects were properly motivated. In our replication, the student subjects had the additional motivation to understand the object system. Its maintenance and the extension of Babsi was the main goal of their upcoming course. Hence, we assume that they were properly motivated. This is also confirmed by the screen recordings.

Tasks Since we – as the authors of ExplorViz – created the tasks, they might have been biased towards our tool. We mitigated this threat by adapting the tasks defined by Cornelissen et al. [2009] as similar as possible.

The tasks might have been too difficult. This threat is reduced by the fact that users from both groups achieved the maximum score in each task except for the last. Furthermore, the average perceived task difficulty, shown in Table 11.6, is between easy (2) and difficult (4), and never too difficult (5).

To reduce the threat of incorrect or biased rating of the solutions, we used a blind review process where two reviewers independently reviewed each solution. The seldom discrepancies in the ratings were at most one point which suggests a high inter-rater reliability.

Omitting the unclear Task App-T3.1 might have biased the results. As can be seen in Figure 11.4, this task had similar results as the others tasks. Thus, not omitting Task App-T3.1 would result in only a small difference of one percent in time spent (27 percent) and two percent in correctness (63 percent).

Miscellaneous The results might have been influenced by time constraints that were too loose or too strict. This threat is reduced by the average perceived time pressure which are near an appropriate (3) rating for both groups.

The tutorials might have differed in terms of quality. This might have led to slower subject performance with one tool. In both groups, the subjects had the possibility to continue to use the tutorial until they felt confident in their understanding.

Users of Extravis had to switch between the questionnaire application and Extravis. This could have let to a disadvantage of the Extravis group. This threat is mitigated by the fact that the users had only to switch to enter answers and both applications were tiled on the screen. In addition, the users of ExplorViz had also to click into the input field.

For reasons of fairness, we disabled some features in ExplorViz for the experiment. This might have influenced the results since the participants had less features to learn. The impact of the disabled features should be investigated in a further experiment.

External Validity

To counteract the threat of limited representativeness of a single object system, we conducted a replication and varied the system. We chose an object system that differed in size and design quality. However, there might be more attributes of the object system that impacted the results. Therefore, further experiments are required to test the impact of other systems.

Our subjects were only made up of students. Thus, they might have acted different to professional software engineers. Further replications are planned with professionals to quantify the impact of this threat.

The short duration of the tool's usage endangers the generalizability of the results. To investigate the impact of this threat, longer experiments should be conducted.

Another external validity threat concerns the program comprehension tasks, which might not reflect real tasks. We adapted our tasks from Cornelissen et al. [2009] who used the framework of [Pacione et al. 2004]. Thus, this threat is mitigated.

11.2.8 Lessons Learned and Challenges Occurred

We consider the user recordings very useful. They enabled us to analyze the users' strategies in detail. Furthermore, we can investigate unsuitable answers as in the case of the users, who did not access the visualization of the correct object system. Thus, the user recordings are a method of quality assurance resulting in more confidence in the data set.

However, we also experienced some challenges during our experiments. Implementing a generator for the input files of tools should be superfluous for using the tool. Therefore, we advise other visualization tool developers, who use externally generated input files, to bundle a working input file generator and a manual with their distribution (if the license permits this). External tools might change or become unavailable, rendering it hard for others to employ the tool.

Furthermore, tutorial material and source code for Extravis were unfortunately not available. Therefore, we had to create our own tutorial materials which might have influenced the experiment. With the source

Table 11.6. Debriefing Questionnaire Results for our PMD Experiment (1 is better – 5 is worse)

	Extravis		ExplorViz	
	mean	stdev.	mean	stdev.
Time pressure (1-5)	3.67	0.78	2.62	0.50
Tool speed (1-5)	3.08	0.90	2.15	0.99
Tutorial helpfulness (1-5)	2.75	1.14	1.92	0.86
Tutorial length (1-5)	3.58	0.67	3.00	0.41
Achieved PMD comprehension (1-5)	3.42	0.79	3.15	0.90
Perceived task difficulty (1-5)				
App-T1	2.33	0.65	2.42	1.00
App-T2.1	3.17	0.72	2.31	0.75
App-T2.2	3.58	0.67	3.46	1.05
App-T3.1	3.83	0.58	3.54	0.88
App-T3.2	3.75	0.75	3.38	0.65
App-T4	3.75	0.87	3.54	0.88

code, we could have, for instance, integrated an interactive tutorial into Extravis. To facilitate a better and easier comparison of visualization tools, *research* prototypes should be developed as open source.

Tool developers should contribute to overcome those challenges and to lower the hurdle for comparison experiments.

11.2.9 Summary of the Application Perspective Evaluation

In this evaluation, we presented two controlled experiments with different-sized object systems to compare our ExplorViz application perspective visualization and the trace visualization tool Extravis in typical program comprehension tasks. We performed the first experiment with PMD and conducted a replication as our second experiment using the Babsi system.

Our first experiment resulted in a significant decrease of 28 percent of time spent and increase in correctness by 61 percent using ExplorViz for PMD. In our replication with the smaller-sized system, the time spent using

either Extravis or ExplorViz was similar. However, the use of ExplorViz significantly increased the correctness by 39 percent.

Our in-depth analysis of the used strategies revealed sources of error in solving the tasks caused by, for instance, color coding or overlapping communication lines. In addition, we identified common challenges for controlled experiments to compare software visualization techniques. For instance, available visualization tools miss sufficient tutorial material.

Our results provide guidance towards ExplorViz for new users who search for a trace visualization tool and have similar tasks as we examined. Since our experiments investigated first time use, the results might be different in long term usage. This should be addressed in further experiments.

11.3 3D-Printed City Model Evaluation

In this section, we present a first preliminary evaluation of our physical models. We compare the impact of using either an on-screen model or a physical model to solve typical program comprehension tasks. Gestures support in thinking and communication processes [Goldin-Meadow 2005] and thus can enhance program comprehension in groups. We hypothesize that physical models support in gesticulation and thus the program comprehension tasks shall be solved in a team-based scenario (pairs of two participants). As object system we use PMD again and measure the *time spent* and *correctness* for each task. Afterwards, we analyze the employed strategies and possible differences between both groups.

As in the previous evaluation, we provide a package [Fittkau et al. 2015e] containing all our experimental data to facilitate the verifiability and reproducibility for replications. It contains the employed version of ExplorViz v0.6-exp (including source code and manual), input files, STL files for 3D-printing the used models, tutorial materials, questionnaires, R scripts, dataset of the raw data and results, and 112 screen and camera recordings of the participant sessions.

After presenting the employed visualizations, we describe the design of our controlled experiment, its operation, data collection, analysis, results, discussion, and threats to validity.

11.3.1 Used Visualizations

Figure 11.5 displays the on-screen visualization of our application perspective used by the control group in the experiment. In contrast to our current visualization, we disabled some features in the on-screen visualization since we wanted to investigate the impact of using a 3D model instead of an on-screen visualization. Otherwise, we would only have tested the current limitations of the physical models.

For the experiment, we hide class names and communication lines, since the physical model cannot provide this information at the moment. We also disabled the highlighting feature, since the physical model does not provide such capabilities. The interactive opening and closing of packages to show or hide their internal details is also disabled. Again the time shifting and the source code viewer feature are disabled since the physical model is static and does not provide such capabilities.

The physical model of PMD is depicted in Figure 11.6. Since we wanted to investigate the impact of using a physical 3D model compared to an on-screen visualization, we kept the semantics and appearance of the physical model as close as possible to the on-screen visualization.

Since we disabled some interaction possibility of the on-screen model, the comparison seems unfair at first sight. However, our goal of the preliminary evaluation is to investigate if there *is* any difference in using physical models compared to on-screen visualization. If we would not have disabled some interaction possibilities, we would have tested for the current shortcomings of our physical model rather than investigating if physical model provide any benefits.

11.3.2 Experimental Design

In addition to general software engineering experimentation guidelines [Kitchenham et al. 2002; Jedlitschka and Pfahl 2005; Di Lucca and Di Penta 2006; Di Penta et al. 2007; Sensalire et al. 2009], we again follow the experimental designs of Wettel et al. [2011] and of Cornelissen et al. [2009]. Similar to these experiments, we use a *between-subjects* design. Thus, each subject only solves tasks with either using the on-screen or the physical model.

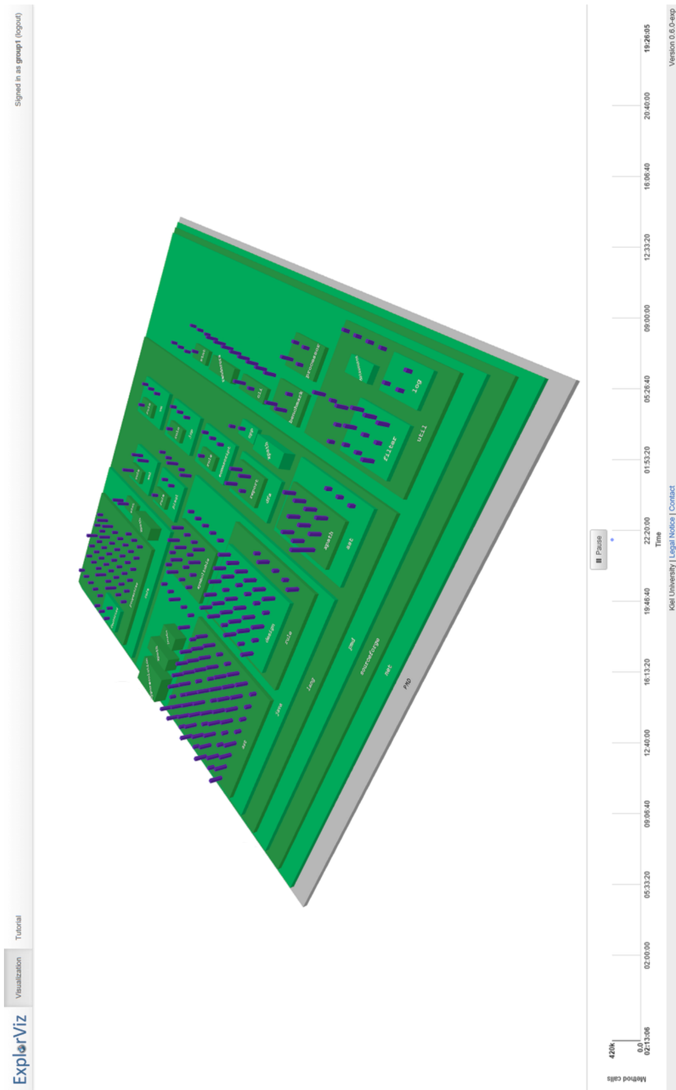


Figure 11.5. On-screenapplication-level perspective of ExplorViz v0.6 visualizing PMD

Figure 11.6. Physical 3D-printed and manually painted city metaphor model of PMD (334mm wide and 354mm deep)

Following the GQM approach [Basili and Weiss 1984], we define the goal of our experiment as quantifying the impact of using either the on-screen model or the physical model for program comprehension.

We name the control group On-Screen Model and the experimental group Physical Model. Due to space constraints, we abbreviate the groups as On-Screen and Physical in figures and tables. To circumvent confusion with the control group and experimental group, we name each group of two participants (a pair) as a team and not as a group.

Research Questions & Hypotheses

We define three physical model related research questions (RQ) for this evaluation:

- ▻ **Phy-RQ1:** What is the ratio between On-Screen Model and Physical Model in the *time required for completing* typical program comprehension tasks?
- ▻ **Phy-RQ2:** What is the ratio between On-Screen Model and Physical Model in the *correctness of solutions* to typical program comprehension tasks?
- ▻ **Phy-RQ3:** Which typical program comprehension tasks benefit more from using the on-screen model and which benefit more from using the physical model?

Accordingly, we formulate two null hypotheses:

- ▻ **Phy-H1$_0$:** There is no difference between On-Screen Model and Physical Model in the *time spent* for completing typical program comprehension tasks.
- ▻ **Phy-H2$_0$:** The *correctness* of solutions to typical program comprehension tasks does not differ between On-Screen Model and Physical Model.

We define the following alternative hypotheses:

▻ **Phy-H1** On-Screen Model and Physical Model require different times for completing typical program comprehension tasks.

▻ **Phy-H2** The correctness of solutions to typical program comprehension tasks differs between On-Screen Model and Physical Model.

For Phy-RQ3, we conduct an in-depth analysis of the results and analyze the recorded sessions of each team in detail.

Dependent and Independent Variables

The independent variable is the model employed for the program comprehension tasks, i.e., on-screen model or physical model. We measured the accuracy (*correctness*) and response time (*time spent*) as dependent variables. These are usually investigated in the context of program comprehension [Wettel et al. 2011; Rajlich and Cowan 1997; Cornelissen et al. 2009].

Treatment

The control group used the on-screen model to solve the given program comprehension tasks. The experimental group solved the tasks utilizing the physical model.

Tasks

Again, we used the framework of Pacione et al. [2004] to create our task set and selected a medium to large-sized object system (PMD) for our experiment since it provides a well designed software architecture. The function of PMD is reporting rule violations on source code. Therefore, it takes source code and a rule configuration as input parameters. To generate our city metaphor visualization of PMD, we monitor the analysis run of PMD on a simple source code file (`Simple.java` by Cornelissen et al. [2009]) and the default `design.xml` of PMD version 5.1.2. The resulting model visualizes 279 used classes of PMD.

Table 11.7. Description of the Program Comprehension Tasks for the Physical Model Experiment

ID	Category	Description	Score
		Context: Metric-Based Analysis	
Phy-T1	A{3,5,6,8}	Find the package containing the one class having the most instances in the application. How is the package named? How many classes (and subpackages if existing) does it contain? Please write down the full package path.	2
		Context: Structural Understanding	
Phy-T2	A{6,8}	What are the names of the three packages directly containing the most classes (without their subpackages)? Please order your answer by beginning with the package containing the most classes and write down the full path.	4
		Context: Concept Location	
Phy-T3	A{1,3,7}	Assuming a good design, which package could contain the *Main* class of the application? Give reasons for your answer.	2
		Context: Structural Understanding	
Phy-T4	A{3,4}	Which package name occurs the most in the application? In addition, shortly describe the distribution of these packages in the system. Hint: Have a look at the different levels of the packages. There are exactly two types of distribution.	3
		Context: Design Understanding	
Phy-T5	A{1,2,3,9}	What is the purpose of the `lang` package and what can you say about its content regarding PMD? Are there any special packages? Do they differ by size? Ignore the `xpath` and `dfa` packages and name three facts in your answer. Hint: Remember the received paper about the introduction to PMD.	3

In Table 11.7, our defined tasks including their context and achievable maximum points is displayed. Furthermore, the covered categories of the framework of Pacione et al. [2004] are shown. To prevent guessing, all tasks were given as open questions. Our task set starts with less complex tasks (identifying the highest class) and ends with a more complex design understanding task. This enabled each team to get familiar with the model in the first tasks and raises the level of complexity in each following task.

Population

The participants were students from the course “Software Engineering”. They received one exercise point for successfully solving each task. For the exercise points, they received bonus points for their exam at the end of the term. To further motivate the participants for the experiment, they could win one of ten gift cards over 10 € for the sole participation. Furthermore, the best eight teams each received a gift card over 20 €.

The teams were assigned to the control or experimental groups by random assignment. To validate the equal distribution of experiences, we asked the teams to perform a self-assessment on a 5-point Likert Scale [Likert 1932] ranging from 0 (no experience) to 4 (expert with years of experience) before the experiment. The average programming experience in the control group was 1.89 versus 1.8 in the experimental group. The average software architecture experience was 1.11 in the control group and 1.00 in the experimental group. Since the experience was self-assessed and both values are within a close range, we assume that random assignment succeeded.

11.3.3 Operation

Next, the operation of our experiment is detailed.

Generating the Input

We generated the input for the control group directly from the execution of PMD. ExplorViz persists its data model into files which act as a replay source during the experiment. How we build the physical model from this on-screen model, was already detailed in Section 8.6.

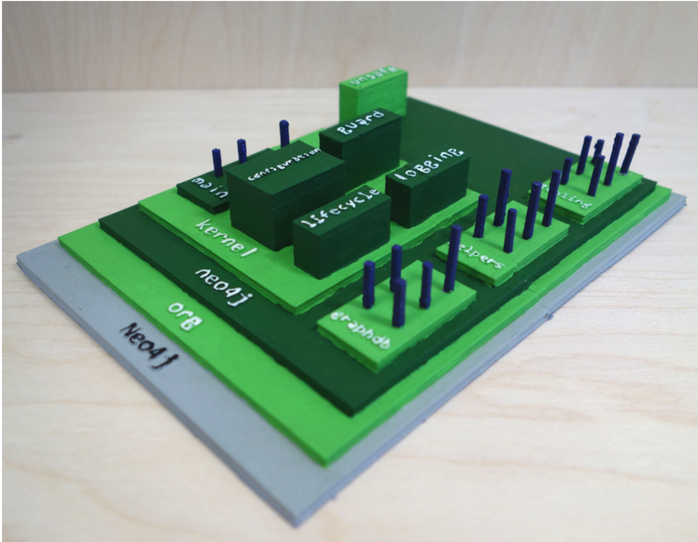


Figure 11.7. Physical model of a mockup of Neo4J used during the tutorial

Tutorials

We provided automated tutorials for both groups of the experiment. This enhanced the validity of our experiments by eliminating human influences. For the tutorial system, we used a small mockup of Neo4J to make the teams familiar with the visualization. For the control group, we integrated an interactive tutorial such that the teams can directly test the functionality on-screen. As tutorial, the experimental group received a physical tutorial model which can be seen in Figure 11.7. Their explanation text was shown on a screen in addition to a photo of the model.

Both groups had the same explanation text for the tutorial except how the control group could interact with the on-screen model, i.e., rotation, panning, and zooming.

Questionnaire

As in the last evaluation, we utilized an electronic questionnaire for both experiment groups. An electronic version provides three advantages over using sheets of paper for us. First, the teams are forced to input valid answers for category fields, e.g., existing experience. Second, manual digitalization can be error-prone in itself and we avoid this by direct electronic capture. Lastly, the timings are automatically recorded for each task and thus time cheating by the teams is impossible.

Pilot Study

Before the controlled experiment took place, we conducted a pilot study with four experienced colleagues as participants forming two teams. The received feedback helped us in improving the material. Furthermore, we added hints to the tasks which were perceived as too difficult.

Procedure

Our experiment took place at the Kiel University. Each team had a single session of up to one hour. Therefore, all teams used the same computer. The display resolution was set to 1920x1200 and prior benchmarking had shown that ExplorViz runs smoothly on this computer.

In order to have recordings of the conducted gesticulation, a Full HD camera recorded all team sessions. At the beginning, the control and the experimental group were told that the table is recorded such that only hands are visible on the video. Furthermore, we explained them that their faces – if they came into the recording area – get pixelated. Afterwards, each team received a sheet of paper containing a short introduction to PMD and its features. They were given sufficient time for reading before accessing the computer.

After telling the teams that they can ask questions at all times, a tutorial for the respective model was started and the Physical Model group were given the physical tutorial model. Subsequently, the questionnaire part was started with personal questions and experiences. Afterwards, the program

Table 11.8. Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points)

	Time Spent		**Correctness**	
	On-Screen	Physical	On-Screen	Physical
mean	28.11	29.39	11.28	11.62
difference		+4.55%		+3.01%
sd	5.94	8.46	1.88	1.47
min	16.24	14.54	7	8.5
median	28.46	27.72	11	12
max	40.97	53.48	14	14
Shapiro-Wilk W	0.9894	0.9436	0.9326	0.9566
Levene F		0.6387		0.7095
Student's t-test				
df		50		50
t		-0.6326		-0.7283
p-value		0.5299		0.4698

comprehension tasks part begun and the experimental group were handed the physical PMD model. The session ended with the debriefing questions.

The less complex tasks (Phy-T1, Phy-T2, Phy-T3) had a time allotment of 5 minutes and the more complex tasks (Phy-T4, Phy-T5) had 10 minutes. During the task, the elapsed time was displayed beneath the task description. The teams were instructed to adhere to this timing but were not forced to end the task. If they reached overtime, the timer was only highlighted.

11.3.4 Data Collection

We collected several data points during our experiment.

Timing and Tracking Information

Our electronic questionnaire automatically determined the timing information for each task. In addition to the camera recordings, we directly

capture the screen of every team using a screen capture tool. The screen recordings enabled us to reconstruct the behavior of the teams and to look for exceptional cases, for instance, technical problems. If we found such a case, we manually corrected the timing data.

Correctness Information

Since we chose an open question format for each task, we conducted a blind review process to rate the answers. After agreeing upon sample solutions, a script randomized the order of the solutions. Thus during the review process, no association between the answers and a team was possible for the reviewers. After both reviewers evaluated all solutions independently, any discrepancies in the ratings were discussed and resolved.

Qualitative Feedback

The participants were asked to give suggestions to improve the model they used for solving the tasks. We only list the top 3 of each experiment group.

Three teams from the control group missed a feature to highlight entities. Furthermore, three teams would like to have more readable labels at a higher zoom distance. One team disliked that the camera was not resetable.

In the experimental group, two teams wanted to have an always visible legend explaining the meaning of the entities. One team would like to have more readable labels. Another team proposed that the amount of instances of a class should be color-coded.

11.3.5 Analysis and Results

Table 11.8 provides descriptive statistics of the overall results related to time spent and correctness for our experiment.

We removed the teams with a total score of less than six points from our analysis. This effected four teams, i.e., one team from the control group and three teams from the experimental group. After watching the recorded sessions, we came to the conclusion that the teams did not understand the semantics of the visualization and thus the given answers to the tasks seem guessed.

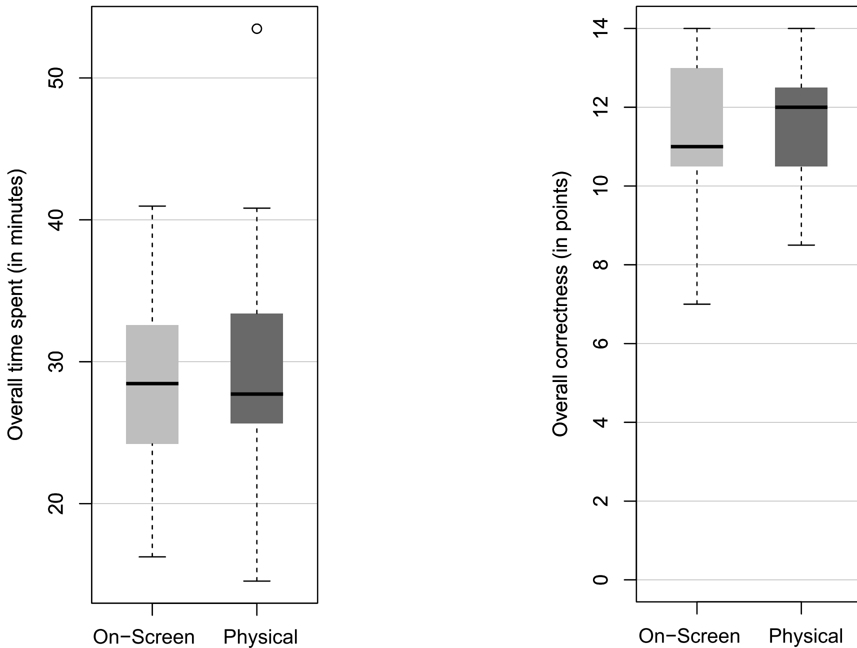


Figure 11.8. Overall time spent and correctness for our physical model experiment

For our analysis, we use the two-tailed Student's t-test which assumes normal distribution. To test for normal distribution, we use the Shapiro-Wilk test [Shapiro and Wilk 1965] which is considered more powerful [Razali and Wah 2011] than, for instance, the Kolmogorov-Smirnov test [Pearson and Hartley 1972]. We conduct a Levene test [Levene 1960] to check for equal or unequal variances. We used the 64-bit R package in version 3.1.2. for the analysis. In addition to the standard packages, we utilize `gplots` and `lawstat` for drawing bar plots and for importing Levene's test functionality, respectively. Furthermore, we chose $\alpha = .05$ to check for significance. The raw data, the R scripts, and our results are available as part of our experimental data package [Fittkau et al. 2015e].

Phy-RQ1 (Time Spent)

We start by checking the null hypothesis Phy-$H1_0$ which states that there is no difference in time spent between On-Screen Model and Physical Model for completing typical program comprehension tasks.

On the left side of Figure 11.8 a box plot for the time spent is displayed. In Table 11.8 the differences between the mean values of On-Screen Model and Physical Model are shown. The Physical Model group required 4.55% more total time for completing the tasks on average.

The Shapiro-Wilk test for normal distribution in each group succeeds and hence we assume normal distribution. The Levene test also succeeds and hence we assume equal variances between both groups.

The Student's t-test reveals a probability value of 0.5299 which is larger than the chosen significance level and we fail to reject the null hypothesis Phy-$H1_0$.

Phy-RQ2 (Correctness)

Next, we check the null hypothesis Phy-$H2_0$ which states that there is no difference in correctness of solutions between On-Screen Model and Physical Model in completing typical program comprehension tasks.

On the right side of Figure 11.8 a box plot for the overall correctness is displayed. The Physical Model group achieved a 3.01% higher average correctness score.

Similar to Phy-RQ1, the Shapiro-Wilk tests and Levene test succeed. The Student's t-test reveals a probability value of 0.4698 which is larger than the chosen significance level and we fail to reject the null hypothesis Phy-$H2_0$.

11.3.6 Discussion

The time spent in the experimental group is slightly higher than the time spent in the control group. However, since we failed to reject Phy-$H1_0$, this could also be caused by chance. The analysis also reveals a slightly higher correctness in the experimental group. However, since we failed to reject Phy-$H2_0$, this could also be caused by chance.

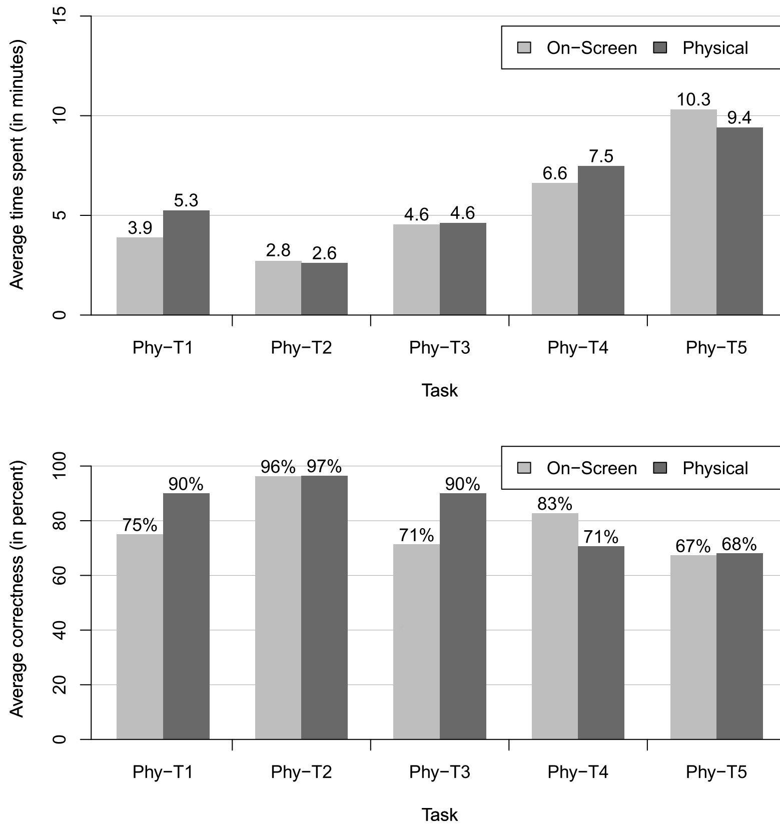


Figure 11.9. Average time spent per task and average correctness per task

Since time spent and correctness of the solutions are in a close range, we conclude that there is only a non-determining overall impact when using physical models for our task set. However, analyzing each task reveals that there is an actual impact in four tasks as can be seen in Figure 11.9. However, the effects of each task compensate each other for our chosen task set resulting in a close ranged overall result.

To investigate the impact of using physical models in each task and the reasons for this impact, we conducted an in-depth analysis of the camera and screen recordings. In the following, used strategies and possible differences between the two groups are described.

Phy-T1

The control group often used the rotation feature to find the highest class. However, they sometimes rotated the model longer than required for the task. Since the experimental group has never seen anything similar to our physical models, the participants often first investigated the whole model. We assume that this is the reason for the increased time needed.

The correctness in the experimental group is increased by 15%. We attribute this fact to an easier navigation to find the highest class. Furthermore, the control group often missed the second part of the task, i.e., counting the classes in the contained package. In comparison, the experimental group was more attentive to the task description.

Phy-T2

Often the On-Screen Model group turned the model to have a bird's-eye view for a better view of the packages and their contained classes. Furthermore, often the gestures (if any were conducted) were only implied such that the monitor screen was not touched.

In the Physical Model group, the gesticulation was more pronounced. One team partner often touched the physical model and showed his teammate the package he was talking about. Furthermore, the package path was often dictated to the partner for writing down the answer, while going through the path having the fingers on the labels (see, for instance, camera

recording of Team 6). However, the task might have been too small or easy to show any effect of the gesticulation in the experimental group.

Phy-T3

We often observed decisions via exclusion procedure in both groups. However, the experimental group outperformed the control group by 19% in average correctness while using roughly the same time. In the control group, the team partner often had to search for the package name that the teammate was talking about.

In contrast, the experiment group often used gestures to show the package under investigation. Furthermore, all packages could be clearly seen without scrolling or rotating. A further reason for the outperformance might be that the package hierarchies were more clear to perceive with the physical model.

Phy-T4

Finding the most occurring package name was more difficult for the experimental group, which needed more time for a lower correctness. We assume that the uneven size of the labels on the physical model causes problems in reading. Especially, the solution package name "rule" is harder to read on the physical model. In contrast, the labels in the on-screen model all have the same size.

Phy-T5

The control group required more time for roughly the same correctness score as the experimental group. We often observed that both team partners used gestures during discussion in the Physical Model group – also in parallel. In contrast, it was harder for the On-Screen Model group to use gestures in parallel since one arm often occludes the screen. These parallel gestures could have increased the efficiency to solve the task in the experimental group.

Summary of the Discussion

In summary, we observed a higher amount of gestures in the experimental group compared to the control group. These were used to communicate with the team partner and reading the package paths. Difficulties with the physical model were encountered due to less readable labels.

To summarize the impact of using physical models in our task set: Two tasks (Phy-T3 and Phy-T5) were positively influenced by the physical model. In contrast, Phy-T4 was negatively influenced. Phy-T2 did not show any differences in using either the on-screen or the physical model. The achieved correctness in Phy-T1 increased with the physical model but also the time spent increased, leading to no clear statement of the impact.

11.3.7 Threats to Validity

In this section, we discuss the threats to internal and external validity [Shadish et al. 2002; Juristo and Moreno 2010; Wohlin et al. 2012] that might have influenced our results.

Internal Validity

We split the internal validity into three parts for our experiment: threats concerning the subjects, the tasks, and miscellaneous threats.

Subjects One threat is that the subjects' experience might not have been fairly distributed across the control group and the experimental group. To mitigate this threat, we randomly assigned the subjects to the groups. These random assignment resulted in a fair self-assessed experience distribution as described in Section 11.3.2.

Another threat is that the subjects might not have been properly motivated. In addition to the lottery, the students received only bonus points for the participation and thus were not forced to take part. Furthermore, we encountered no unmotivated user behavior in the screen recordings with the exception of the four teams scoring below six points that we have excluded from our analysis (Section 11.3.5). Hence, we assume that the remaining teams were properly motivated.

Table 11.9. Debriefing Questionnaire Results for our Physical Model Experiment (1 is better – 5 is worse)

	On-Screen		Physical	
	mean	stdev.	mean	stdev.
Time pressure (1-5)	2.04	0.65	2.04	0.79
Tool speed (1-5)	1.48	0.64	–	–
Tutorial helpfulness (1-5)	2.11	0.85	2.20	1.08
Tutorial length (1-5)	3.30	0.47	3.20	0.50
Achieved PMD comprehension (1-5)	3.00	0.83	3.08	0.76
Perceived task difficulty (1-5)				
Phy-T1	1.74	0.66	2.12	0.73
Phy-T2	2.07	0.73	2.08	0.57
Phy-T3	2.74	0.71	2.88	0.53
Phy-T4	3.56	0.70	3.68	0.48
Phy-T5	3.41	0.69	3.60	0.65

A further threat is that the subjects might not have been sufficiently competent. At the beginning of the questionnaire, most teams stated beginner or regular programming experience. This should be sufficient for our rather easy task set.

Tasks The tasks might have been biased towards one kind of model. We mitigated this threat by disabling some features in the on-screen model such that both models provide equal functionality.

A further threat is the incorrect or biased rating of the solutions. We reduced this threat by conducting a blind review where two reviewers independently reviewed each solution. There were seldom discrepancies in the ratings with at most one point suggesting a high inter-rater reliability.

Another threat is that the tasks might have been too difficult. Teams from both groups achieved the maximum score in each task. Furthermore, the average perceived task difficulty, shown in Table 11.9, is never between difficult (4) and too difficult (5).

Miscellaneous The results might have been influenced by time constraints that were too loose or too strict. In contrast, the average perceived time pressure was slightly above little (2) for both groups and thus the time pressure was not too loose or too strict.

The experimental group had to switch between the physical model and the keyboard. This could have led to a disadvantage of this group. This threat is mitigated by the fact that one team member could stay at the keyboard and the other one could tell him what he should write.

Another threat concerns the possible different quality of the tutorials. This might have led to slower performance with one model. In both groups, the teams had the possibility to continue to use the tutorial until they felt confident in their understanding of the semantics. Furthermore, both groups had the same tutorial text except the explanation of interaction with the on-screen model.

External Validity

Experimenting with only one single object system is not representative for all available systems. Therefore, further experiments with different object systems should be conducted. Another threat concerns the program comprehension tasks, which might not reflect real tasks. We used the framework of Pacione et al. [2004] to define the task set and to mitigate this threat.

Our subjects were made up of students. Therefore, they might have acted differently to professional software engineers. Replications with professionals should be conducted to quantify this impact. Furthermore, the teams were made up of only two persons. The impact of using physical models in larger teams should be investigated in further experiments.

11.3.8 Summary of the Physical Model Evaluation

In this section, we evaluated the impact of using our physical models on program comprehension in a team-based scenario by conducting a controlled experiment.

We identified two tasks that benefit from using physical models in comparison to using on-screen models. However, our experiment resulted in no overall impact neither on time spent nor on correctness of solutions to the program comprehension tasks, since the effects of each task compensate each other for our chosen task set.

Our in-depth analysis of the used strategies supports our hypothesis that physical models provide an appropriate, complementary communication basis and increase interaction when solving comprehension tasks in small teams. In the analysis, we observed an increase in the amount of performed gestures when using the physical model.

11.4 Landscape Perspective Evaluation

In this section, we present our controlled experiment for evaluating the landscape perspective. In the experiment, we compare the usage of a flat, state-of-the-art landscape visualization to our hierarchical landscape visualization in typical system comprehension tasks. As object landscape we use a model of the technical IT infrastructure of the Kiel University and measure the *time spent* and *correctness* for each task. Afterwards, we analyze the employed strategies and possible differences between both groups.

To facilitate the verifiability and reproducibility for replications and further experiments, we again provide a package [Fittkau et al. 2015d] containing all our experimental data. Included are the employed version of ExplorViz v1.0-exp (including source code and manual), input files, tutorial materials, questionnaires, R scripts, dataset of the raw data and results, and 29 screen recordings of the participant sessions.

After presenting the employed visualizations, we describe the design of our controlled experiment, its operation, data collection, analysis, results, discussion (including reasoning about the different performances in each task), and threats to validity.

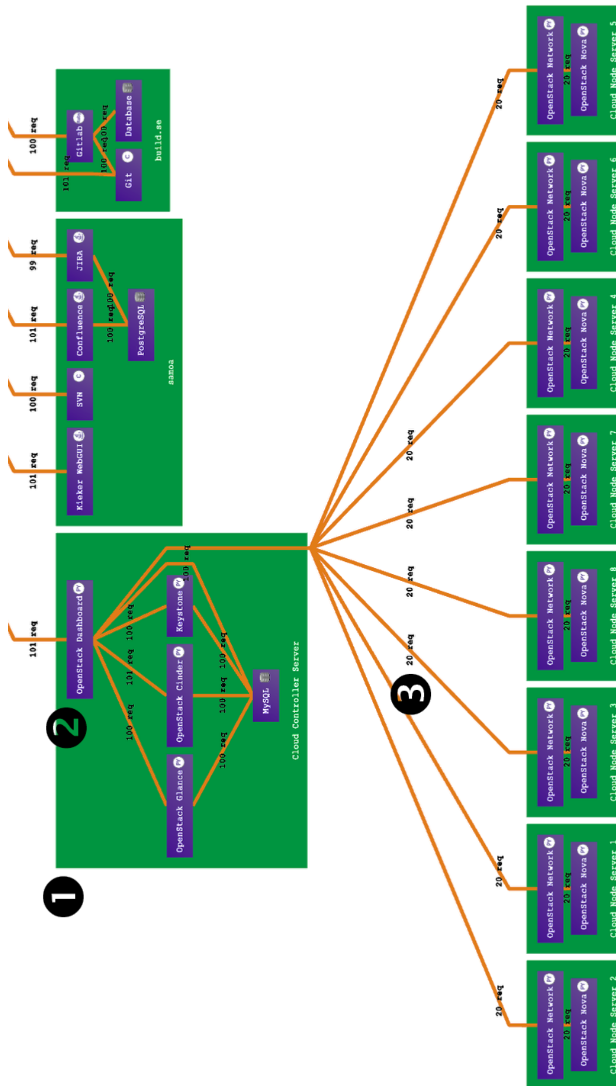


Figure 11.10. An excerpt (29 of 140 applications) of the model of the technical IT infrastructure of the Kiel University in the flat landscape visualization

11.4.1 Used Visualizations

This section introduces the flat software landscape visualization used in the experiment by the control group to solve system comprehension tasks.

Current landscape visualizations can mostly be found in application performance management (APM) tools, for instance, AppDynamics,[5] Foglight,[6] or Dynatrace.[7] Those tools are often driven by commercial interest and thus are not free to use or – if they have an evaluation phase – it is explicitly prohibited to conduct studies with these versions. Therefore, we had to create our own implementation of the visualization which follows the concepts of current landscape visualizations available in APM tools. By implementing the visualization into our ExplorViz tool, we assure that interaction capabilities are the same between both groups in the experiment. This leads to a higher reliability of the presented results.

After surveying the available visualizations, we implemented the visualization depicted in Figure 11.10 which is a mixture of the concepts we rated as best suitable for system comprehension. The figure shows an excerpt of the used object landscape, i.e., a model of the technical IT infrastructure of the Kiel University. Nodes are visualized as green boxes (❶) with white labels representing the hostname of each node at the bottom. The applications running on the nodes are visualized by purple boxes (❷). A white label shows the application name at the center. Besides the label, the programming language or – in the special case of a database – a database symbol is depicted. The communication between applications is represented by orange lines (❸). The conducted request count is shown next to a line in black letters in the abbreviated form of, e.g., *10 req*.

Figure 11.11 displays the same excerpt as it is present in Figure 11.10 but visualized in our current landscape perspective visualization.

[5] `http://www.appdynamics.com`
[6] `http://www.foglight.com`
[7] `http://www.dynatrace.com`

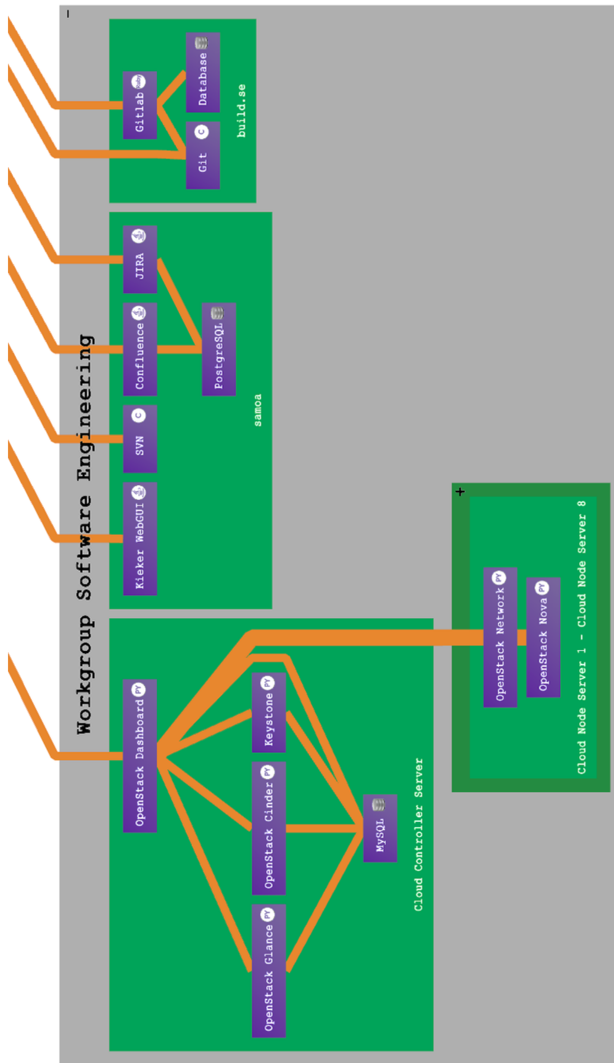


Figure 11.11. An excerpt of the model of the technical IT infrastructure of the Kiel University in our hierarchical landscape visualization

11.4.2 Experimental Design

In addition to general software engineering experimentation guidelines [Kitchenham et al. 2002; Jedlitschka and Pfahl 2005; Di Lucca and Di Penta 2006; Di Penta et al. 2007; Sensalire et al. 2009], we follow the designs of Wettel et al. [2011] and of Cornelissen et al. [2009]. Similar to them, we use a *between-subjects* design. Thus, each subject solves tasks either using the flat or the hierarchical visualization. Following GQM [Basili and Weiss 1984], we define the goal of our experiment as quantifying the impact of using either the flat visualization or the hierarchical one for system comprehension.

We name the control group *Flat Group* and the experimental group *Hierarchical Group*. Due to space constraints, we abbreviate the groups as *Flat* and *Hierarchical* in figures and tables.

Research Questions & Hypotheses

We define three landscape perspective related research questions (Land-RQ) for our evaluation:

▻ **Land-RQ1:** What is the ratio between Flat Group and Hierarchical Group in the *time required for completing*
typical system comprehension tasks?

▻ **Land-RQ2:** What is the ratio between Flat Group and Hierarchical Group in the *correctness of solutions* to typical system comprehension tasks?

▻ **Land-RQ3:** Which typical sources of error exist when solving system comprehension tasks with either of the two visualization types?

Accordingly, we formulate two hypotheses:

▻ **Land-H1** Flat Group and Hierarchical Group require different times for completing typical system comprehension tasks.

▻ **Land-H2** The correctness of solutions to typical system comprehension tasks differs between Flat Group and Hierarchical Group.

The null hypotheses Land-$H1_0$ and Land-$H2_0$ follow analogously. For Land-RQ3, we conduct an in-depth analysis of the results and analyze the recorded sessions of each participant in detail.

Dependent and Independent Variables

The independent variable is the visualization used for the system comprehension tasks, i.e., flat or hierarchical visualization. We measured the accuracy (*correctness*) and response time (*time spent*) as dependent variables. These are usually investigated in the context of program comprehension [Wettel et al. 2011; Rajlich and Cowan 1997; Cornelissen et al. 2009] and thus should also be applicable in the context of system comprehension.

Treatment

The control group used the flat visualization to solve the given system comprehension tasks. The experimental group solved the tasks utilizing the hierarchical visualization which includes the abstractions of systems, node groups, and the communication lines representing the conducted requests by the line's thickness.

Tasks

We selected a medium-sized software landscape (140 applications) for our experiment. Our model of the technical IT infrastructure of the Kiel University landscape represents services of our working group, computing center services, examination services, information services, system operating group services, and management services. We modeled the landscape by available information from the Internet and prior knowledge, and thus the model might not reflect the real deployment in detail. However, this is unimportant for the actual tasks.

In Table 11.10, our defined tasks including their context and achievable maximum points are displayed. To prevent guessing, all tasks were given as open questions. Our task set starts with less complex tasks (identifying applications with a high fan-in) and ends with a more complex risk management task. This enabled each subject to get familiar with the visualization in the first tasks and raises the level of complexity in the following ones. We chose only five tasks since we aimed to stay in a one hour time slot and prevent exhaustion issues.

Table 11.10. Description of the System Comprehension Tasks for the Landscape Experiment

ID	Description	Score
	Context: Identification of Critical Dependencies	
Land-T1	Name three applications that have a high fan-in (at least two incoming communication lines). The two incoming communication lines should be on one node and not distributed over multiple nodes.	3
	Context: Potential Bottleneck Detection	
Land-T2	Name the Top 3 communications with the highest request count in descending order. Write down the start application and the end application.	4
	Context: Scalability Evaluation	
Land-T3	Which applications are duplicated on multiple nodes? The answer should contain all 8 duplicated applications which are all named differently. Hint: The hostname of the nodes, where the applications are running, are numbered, e.g., Server 1, Server 2,...	4
	Context: Service Analysis	
Land-T4	What is the purpose of the WWWPRINT application in your opinion? How does the process might work to achieve the functionality for the user?	4
	Context: Risk Management	
Land-T5	What are the consequences of a failure of the LDAP application? Name all affected applications and briefly describe their purposes. Hint: Remember the received paper about the introduction to the university landscape.	7

Population

The subjects were students from the master course "Software Engineering for Parallel and Distributed Systems". For successfully solving one task, they received bonus points for the final exam of the course. As further motivation, the students could win one of ten gift cards over 10 € for the sole participation and the best five subjects each received a gift card over 30 €.

The subjects were assigned to the Flat Group or Hierarchical Group by random assignment. To validate the equal distribution of experiences, we asked the participants to perform a self-assessment on a 5-point Likert Scale [Likert 1932] ranging from 0 (no experience) to 4 (expert with years of experience) before the experiment. The average programming experience in the control group was 2.5 versus 2.6 in the experimental group. The average dynamic analysis experience was between no experience and beginner in both group. Since the experience was self-assessed, we assume that random assignment succeeded.

11.4.3 Operation

Generating the Input

We generated the input for the experiment by modeling the object landscape in ExplorViz by means of a modeling editor using our visualization as Domain-Specific Language (DSL). Afterwards, we exported the model as a script file. This file contains one entry for each application that should be started. We have written a small configurable RPC application which acts as a server and connects to different servers configurable on the command line. This small application can pass off as the application names from the modeled landscapes which is also a part of one entry in the script. Therefore, the script imitates the real object landscape without having the need to instrument the productive applications. After executing the script and receiving the monitored data of the remote procedure calls, ExplorViz persists its created landscape model into a file which acts as a replay source during the experiment.

Tutorials

We provided automated tutorials for both groups of the experiment. This enhanced the validity of our experiments by eliminating human influences. For the tutorial system, we used a small-sized model of the Kiel Data Management Infrastructure for ocean science [Fittkau et al. 2015g] to make the subjects familiar with the visualization. Both groups had the same explanation text for the tutorial except information about the abstractions in the hierarchical visualization which were only available to the associated group.

Questionnaire

Like in the other evaluations, both groups answered the questions on an electronic questionnaire. An electronic version provides three advantages over using sheets of paper for us. First, time cheating by the subjects is impossible since the timings are automatically recorded. Second, we avoid a possible error-prone manual digitalization by direct electronic capture. Lastly, the participants are forced to input valid answers for category fields, e.g., their experience.

Pilot Study

To check whether the material and questions are understandable for the target population, we conducted a pilot study with two master students as participants before the actual experiment. After this study, we improved the materials based on the feedback. In addition, we added hints to the tasks which were perceived as too difficult or which were misunderstood. While the hint for Land-T3 might hinder the visual query in the Hierarchical Group, the hint for Land-T5 does not favor any group.

Procedure

Our experiment took place at the Kiel University. Each participant had a single session of up to 45 minutes. All subjects used the same computer which had a display resolution of 1920x1200. Before the experiment took place, we benchmarked the computer to ensure that both types of visualization run smoothly.

At the beginning of each session, each subject received a sheet of paper containing an introduction to the object landscape and a description of selected applications which might be unknown. We gave the subjects sufficient time for reading before they could access the computer. After telling the participants that they can ask questions at all times, a tutorial for the respective visualization type was started. Then, the questionnaire was started with personal questions and experiences. Afterwards, the system comprehension tasks begun. The session ended with debriefing questions.

The less complex tasks (Land-T1, Land-T2, Land-T3, Land-T4) had a time allotment of 5 minutes. The more complex task Land-T5 had an allotment of 10 minutes. The elapsed time was displayed beneath the task description during each task. The subjects were instructed to adhere to this timing. However, if they reached overtime, the timer was only highlighted in red and they were not forced to end the task.

11.4.4 Data Collection

Timing and Tracking Information

The timing information for each task is automatically determined by our electronic questionnaire. In addition, the computer screen of every session is captured using a screen capture tool. With the screen recordings, we could analyze the behavior of each participant. Furthermore, it enabled us to look for exceptional cases, for instance, technical problems encountered by the participant. The recordings become important in the case of technical problems since the timing data must manually be corrected and it must be reconstructed how long the subject actually worked on the task.

Correctness Information

The open question format implies to conduct a blind review for rating the given answers. The two reviewers first agreed upon sample solutions and a maximum score for each task. A script randomized the order of the solutions so that no association between the answers and the originating group could be drawn. Then, both reviewers evaluated all solutions independently. Afterwards, any discrepancies in the ratings were discussed and resolved.

Table 11.11. Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points)

	Time Spent		**Correctness**	
	Flat	Hierarchical	Flat	Hierarchical
mean	23.49	23.45	17.07	19.5
difference		-0.17 %		+14.24 %
sd	3.87	5.29	3.27	2.93
min	15.03	15.93	9	11
median	24.64	23.14	17.25	20.5
max	29.68	33.16	22	22
Shapiro-Wilk W	0.9232	0.9605	0.9156	0.7933
Levene F		2.1048		1.2307
Student's t-test				
df		27		27
t		0.0251		-2.4102
p-value		0.9802		0.02303

Qualitative Feedback

The participants were asked to give suggestions to improve the visualization they used for solving the tasks. We only list the Top 3 for each group.

In the Flat Group, five users noted that some labels representing the request count overlapped such that they were forced to get the count by hovering over the communication line. Two users suggested to implement a sortable table for Task Land-T2. Furthermore, two subjects disliked that the font size is not increasing when zooming out.

In the Hierarchical Group, three subjects suggested to use animations for opening and closing the systems or node groups. Two users would like to be able to highlight nodes or connections. As in the flat visualization group, one subject disliked that the font size is not increasing when zooming out.

11.4.5 Analysis and Results

Descriptive statistics for the results of our experiment are shown in Table 11.11. Although we had three outliers, we did not remove any subjects from our analysis since the errors were comprehensible and did not result from exceptional cases as also advised by Wohlin et al. [2012]. We use the two-tailed Student's t-test for our analysis which assumes normal distribution and depends on equal or unequal variances. To test for normal distribution, we use the Shapiro-Wilk test [Shapiro and Wilk 1965] which is considered more powerful [Razali and Wah 2011] than, for instance, the Kolmogorov-Smirnov test [Pearson and Hartley 1972]. We conduct a Levene test [Levene 1960] to check for equal or unequal variances.

We used the 64-bit R package in version 3.1.3. for the analysis. As supplementary packages, we utilize `gplots` and `lawstat` for drawing bar plots and for importing Levene's test functionality, respectively. Furthermore, we chose $\alpha = .05$ to check for significance. The raw data, R scripts, and results are available as part of our experimental data package [Fittkau et al. 2015d].

Land-RQ1 (Time Spent)

We start by checking the null hypothesis Land-$H1_0$ which states that there is no difference between the flat and the hierarchical visualization in respect to the time spent on the system comprehension tasks. The box plot for the time spent is displayed on the left side of Figure 11.12. Table 11.11 shows the differences between the mean values of Flat Group and Hierarchical Group.

The Shapiro-Wilk test for normal distribution in each group succeeds and hence we assume normal distribution of our data in each group. The Levene test also succeeds and thus we assume equal variances between the Flat Group and the Hierarchical Group. The Student's t-test reveals a probability value of 0.98 which is above our chosen significance level of 0.05. Therefore, we fail to reject the null hypothesis Land-$H1_0$.

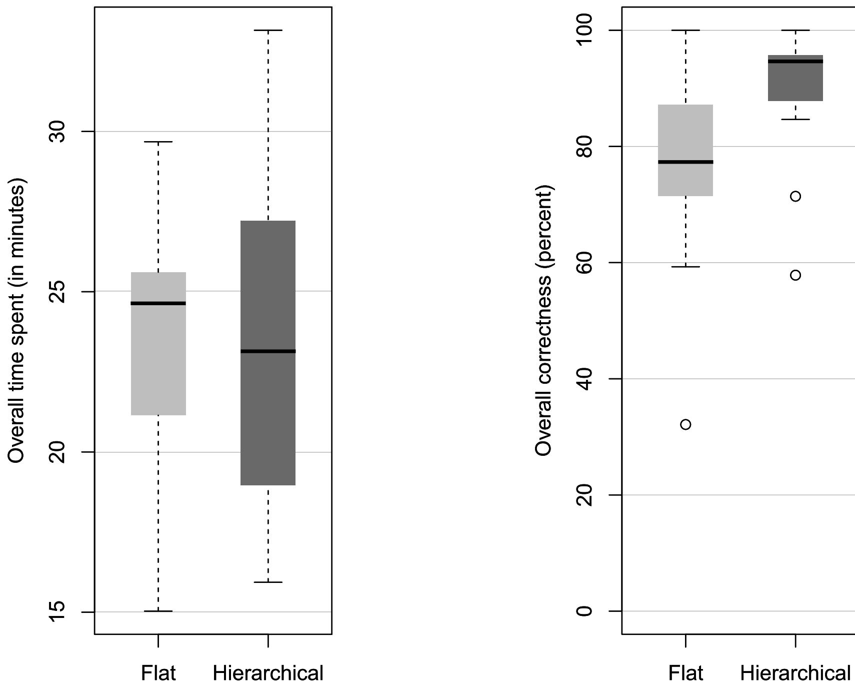


Figure 11.12. Overall time spent and correctness for our landscape experiment

Land-RQ2 (Correctness)

Next, we check the null hypothesis Land-$H2_0$ which states that there is no difference between the two groups in respect to correctness of the solutions. A box plot for the overall correctness in each group is shown on the right side of Figure 11.12.

The Shapiro-Wilk test for the Flat Group succeeds and hence we assume normal distribution in this group. The test fails for the Hierarchical Group. Therefore, we plotted a histogram and looked at the actual distribution. Most points are near 100 % and the rest follows a normal distribution to

the left side. Since 100 % imposes a natural cutoff for the task correctness and the rest of the values are normally distributed, we also assume normal distribution for this group. The Levene test succeeds and thus we assume equal variances between both groups. The Student's t-test reveals a probability value of 0.02 which is below our chosen significance level of 0.05. As a result, we reject Land-$H2_0$ in favor of the alternative hypothesis Land-H2 (t-test t = -2.4102, df = 27, p = 0.02303).

11.4.6 Discussion

The results for the time spent are not statistically significant. Hence, there is no statistical evidence for a difference in the time spent meaning it could be equal or even be different. However, it is likely that the impact of using a flat or hierarchical visualization is negligible in terms of time spent. Whether one group took a few seconds less, is typically out of interest. In terms of task correctness, the Hierarchical Group outperformed the Flat Group by 14 %. This difference is statistically significant in our experiment.

Since the time spent is negligibly different or equal, and the correctness of the solutions are higher in the hierarchical visualization, we conclude that using the hierarchical visualization provides more benefits than the flat visualization. To investigate the reasons for this circumstance, we conducted an in-depth analysis of the recorded user sessions and looked for the employed strategies and typical sources of error. In the following, these findings are described.

Land-T1

Both groups used the same strategy to find the three applications with a high fan-in. At first, the subjects got a general idea of the software landscape looking at its coarse-grained structure. Then, they zoomed in such that they can read the application labels and moved the view until they discovered the wanted applications. Some of the participants began their search from one side (left or right) such that they only needed to go over the landscape once. Others started at a random position and therefore, had to go over the landscape twice if they did not find the wanted applications.

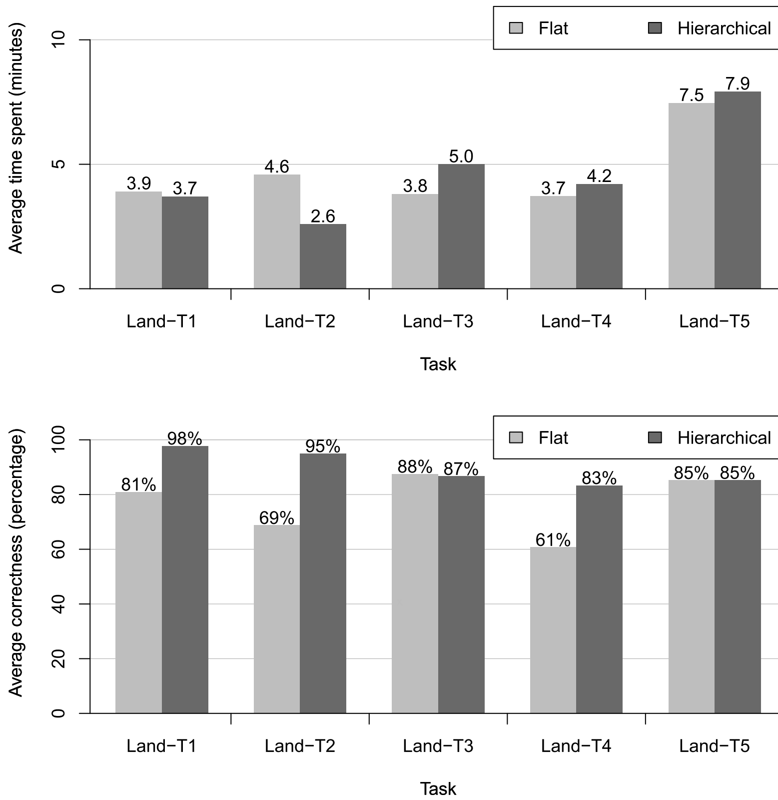


Figure 11.13. Average time spent per task and average correctness per task

A source of error in this task was the distinction between applications and nodes. We observed this confusion more in the Flat Group than in the Hierarchical Group which could be a reason for the 17 % higher task correctness in the hierarchical visualization group. Since the hierarchical visualization group uses more hierarchies, the participants in this group might be more aware of the differences between each abstraction level.

A further possible reason for the higher task correctness might be that the hierarchical landscape visualization is more compact since node groups are closed and thus take less space.

Land-T2

Since the presentation of request labels are different in the groups, each group used a different strategy. Subjects in the Flat Group again started from one side of the landscape visualization searching for the label with the highest request count. Sometimes the labels overlapped and the participants hovered over the communication line to get the number as popup.

The Hierarchical Group zoomed out to get an overview where the thickest communication lines are located. They hovered over these lines to get the actual request count and to form the descending order. Interestingly, the subjects often only distinguished between small and larger lines (4 steps of line thickness were visualized). Therefore, they also looked at medium sized lines instead of only looking at the largest lines.

In respect to time spent and task correctness, the Hierarchical Group outperformed the Flat Group in both metrics. One reason for this circumstance might be that in the flat visualization the manual search for the highest request count can be error-prone in respect to finding and in respect to the descending order.

Land-T3

In this task, both groups used the same strategy to find duplicated applications at the beginning. Participants from both groups formed the visual query for applications that are named equally and run on different nodes. The Flat Group succeeded with this query since the visualization only con-

tains nodes and applications and no closed node group entities. In contrast, the Hierarchical Group did not find such applications since the node groups are closed by default. Often they realized this circumstance after going through the whole landscape without finding any duplicate applications and then looked for node groups. Only a few subjects in the hierarchical visualization looked for the node group entity right from the start.

From our expectations, the Hierarchical Group should have outperformed the Flat Group. However, the opposite happened. While the task correctness is roughly equal, the time spent was larger due to the wrong visual query in the beginning. Therefore, the introduced node groups abstraction confused the subjects in this task. We attribute this to a first time use and properly this behavior changes in long-term usage.

A further reason for the good performance of the Flat Group originates from our layout which visually grouped nodes running duplicated applications instead of distributing the nodes over the whole landscape. Otherwise, the comparison of applications would have been much harder in this group.

Land-T4

Both groups followed the same strategy for describing the purpose of the `WWWPrint` application. First, the subjects had to search the application. After finding it, they looked at the communication lines and the connected applications. Then, they reasoned about the purpose on the basis of the application names and their connections. Additionally, the introduction sheet provided hints about the meaning of, e.g., `LDAP`.

n average, the Hierarchical Group required 30 seconds more time for this task. Since the visualization of the `WWWPrint` node is similar – except communication lines –, we expected an equal timing for this task. Therefore, we also looked at the median which actually reveals an roughly equal time spent. The average is influenced by two outliers (User 5 and User 25 – both taking around six minutes). One source of error in this task was overlooking the connection to `LDAP` and thus not detected that `WWWPrint` requires authentication. We observed this behavior more often in the Flat Group than in the Hierarchical Group which might be a reason for the higher task correctness.

Land-T5

Again, both groups had the same strategy. First, they searched for the `LDAP` application. Afterwards, they followed the communication lines backward to find the services which would fail when `LDAP` fails.

Similar to Task Land-T4, we expected an equal or lower time spent in this task since the layout is more compact in the Hierarchical Group. However, the time spent is 25 seconds higher in average. In the median, it is actually 25 seconds *lower* than the time spent by the Flat Group, again influenced by User 25 who took about 16 minutes.

A typical source of error in this task was not describing the purpose of the potentially failing services. We did not observe any difference in the occurrence of this behavior between the two groups which possibly led to the similar task correctness.

Summary of the Discussion

In summary, we observed three issues leading to a higher time spent or lower task correctness in the Flat Group. The subjects mistook nodes for applications. This happened also in the Hierarchical Group but less frequently. Furthermore, when space became narrow, the request labels overlapped. This led to manually hovering over the connection to get the actual request count. The third issue is related to the layout that was inherently larger due to the absence of abstractions, i.e., especially a node group abstraction. Therefore, the Flat Group often required more time to find entities.

Subjects in the Hierarchical Group often did not utilize the node group abstraction efficiently right from the start. Therefore, this abstraction imposes a non-zero learning curve.

One general issue which affected both groups was that some participants mixed up the direction of the communication which goes from top to bottom in our layout. They sometimes thought it would go from bottom to top. This issue could probably be solved by an always visible legend.

11.4.7 Threats to Validity

In this section, we discuss the threats to internal and external validity [Wohlin et al. 2012; Shadish et al. 2002; Juristo and Moreno 2010] that might have influenced our results.

Internal Validity

We split the internal validity into three parts for our experiment: threats concerning the subjects, the tasks, and miscellaneous threats.

Subjects The subjects might not have been sufficiently competent. Most participants rated themselves as having regular programming experience which should be sufficient for our task set.

A further threat is that the experience of the subjects might not have been fairly distributed across the Flat Group and the Hierarchical Group. This threat is mitigated by randomly assigning the participants to each group. We checked that the random assignment resulted in a fairly distributed self-assessed experience. The concrete numbers were already described in Section 11.4.2.

The subjects might not have been properly motivated which imposes another threat to validity of our experiment. The students were not forced to take part in the experimenter since in addition to the lottery, the students received only bonus points which are not required to pass the exam. Furthermore, while watching the recorded user sessions, we did not encounter any unmotivated user behavior.

Tasks One task-related threat is that the solutions were incorrectly rated or a reviewer might have been biased towards one experiment group. We mitigated this threat by employing a blind review process. Before the actual reviewing process took place, the solutions were mixed by a script such that no trace to the originating group was possible for the reviewers. Then, two reviewers independently reviewed each solution. Afterwards, the seldom discrepancies in the ratings were discussed. These discrepancies were at most one point suggesting a high inter-rater reliability.

Table 11.12. Debriefing Questionnaire Results for our Landscape Experiment (1 is better – 5 is worse)

	Flat		**Hierarchical**	
	mean	stdev.	mean	stdev.
Time pressure (1-5)	2.14	0.77	2.20	0.94
Tool speed (1-5)	2.07	1.00	1.60	0.83
Tutorial helpfulness (1-5)	2.21	0.58	1.6	0.51
Tutorial length (1-5)	3.21	0.70	3.00	0.65
Achieved comprehension (1-5)	2.50	0.85	2.20	0.68
Perceived task difficulty (1-5)				
Land-T1	2.36	0.84	2.20	0.77
Land-T2	2.64	0.93	2.00	0.53
Land-T3	2.64	0.63	3.00	0.76
Land-T4	3.00	0.78	2.93	0.70
Land-T5	2.93	0.73	3.00	0.53

The tasks might have been too difficult which imposes another threat to validity. However, subjects from both groups achieved the maximum score in each task. The average perceived task difficulty is shown in Table 11.12. Since the average rating of each task is never difficult (4) or too difficult (5), we conclude that the difficulty of each task was appropriate.

Another threat is that the tasks might have been biased towards one type of visualization. Since the average perceived task difficulty only differs significantly in Land-T2 and Land-T3 between both groups, at least the other tasks are not biased towards one type of visualization. Task Land-T2 was perceived easier in the Hierarchical Group and Task Land-T3 was perceived harder in this group. Therefore, we conclude that this potential bias is fairly distributed between the two experiment groups.

Miscellaneous A possible different quality of the tutorials impose another threat to validity. In both groups, the teams had the possibility to continue to use the tutorial until they felt confident in their understanding of the

semantics. In addition, both groups had the same tutorial text except the hierarchical abstractions in the Hierarchical Group.

Too loose or strict time constraints might have influenced the results of our experiment. However, the average perceived time pressure was slightly above little (2) for both groups. Therefore, we assume that the time pressure was well fitted for the tasks.

External Validity

Our experiment only involved one single object landscape. Since this is typically not representative for all available software landscapes, further experiments with different object landscapes should be conducted.

Another threat concerns the system comprehension tasks, which might not reflect real tasks. Unfortunately, we did not find any task frameworks for composing system comprehension tasks for software landscapes. We also took a look at program comprehension task frameworks, e.g., the framework by Pacione et al. [2004]. However, we could not adapt the tasks in a reasonable way. Therefore, we used our present knowledge about software landscapes to made up tasks in interesting contexts from real usage scenarios.

Only students participated in our experiment. Professionals might act differently which could result in a different outcome of our experiment. To investigate the impact of this threat, further controlled experiments should be conducted. To lower the setup effort for such experiments, our experimental design can be reused.

11.4.8 Summary of the Landscape Perspective Evaluation

In this section, we presented a controlled experiment to investigate which landscape visualization, i.e., current APM or ExplorViz, supports solving typical system comprehension tasks more effectively or efficiently.

Our experiment resulted in a statistically significant increase of 14 % task correctness in the hierarchical visualization group for system comprehension tasks. The time used by the participants on the tasks did not differ significantly. Since the time spent is approximately equal and the task cor-

rectness is improved by our hierarchical visualization, it provides a valuable enhancement to the current state of the art in landscape visualizations in the context of system comprehension.

During our analysis, we identified some challenges encountered by the participants in both visualizations types. Some subjects mistook nodes for applications. This happened more frequently in the Flat Group than in the Hierarchical Group. Furthermore, some participants from the Hierarchical Group did not utilize the node group abstraction efficiently right from the start. A further challenge was imposed by the flow-based layout. Some participants from both groups sometimes mixed up the direction of the communication.

11.5 Summary

In this section, we conclude the results of the evaluations for our visualization approach.

In the application perspective evaluation, our city metaphor-based application visualization was more efficient and effective than the established Extravis tool. The replication confirmed that our application perspective provides benefits in solution correctness but the time spent on the task was roughly the same for both visualizations. Notably, all participant could finish the tasks since no tool crashes occurred and no bugs hindered the subjects from solving the given tasks. Therefore, the evaluation also showed that our tool provides good usability in the context of program comprehension tasks. We conclude that our application-level perspective provides benefits over a current established trace visualization. However, a comparison to other trace visualization techniques – such as the one of Trümper et al. [2010] – should also be conducted in future work.

Due to our task set, the preliminary evaluation of the physical 3D-printed models resulted in no significant overall effect since the impact on the tasks compensated each other. However, two tasks – which involved the most team discussion – were positively affected by using the physical models. Therefore, the evaluation confirmed that physical models posses

the potential to enhance the team-based program comprehension process and can provide a beneficial future research direction.

The experiment comparing our landscape perspective and a state-of-the-art visualization of a mixture of APM tools resulted in an increase of solution correctness while the time spent on the tasks was roughly the same. Therefore, our addition of further hierarchical abstractions to current landscape visualizations showed beneficial and usable.

The evaluation of the VR approach is not detailed in this chapter since it was conducted in a bachelor's thesis by Krause [2015]. Therefore, we only briefly summarize the qualitative results of the eleven structured interviews and refer to [Krause 2015] for details. The participants of the interviews rated the developed gestures for translation, rotation, and selection as highly usable. However, the zooming gesture was less favored. In general, the subjects see potential for virtual reality in program comprehension. Therefore, also the VR approach imposes a potential beneficial research direction to utilize the advantages of immersive VR experience in software visualization.

In respect to our approach and its implementation, the object systems (PMD and Babsi) and the object landscape (the IT infrastructure of the Kiel University) were generated from program executions and were not artificially modeled without actual monitoring. Therefore, the evaluations also show that our approach and implementation is functioning correctly starting from the monitoring information gathering until the actual visualization of the information. Notably, this includes the correct generation of our landscape meta-model presented in Chapter 7 which was therefore not explicitly addressed in a dedicated evaluation.

Since all parts of our visualization approach were evaluated and either showed as beneficial research direction (VR and physical 3D-printed models), or they proved more effective or efficient than the current state-of-the-art (application and landscape perspective), our visualization approach shows feasible, usable, and beneficial in a program and system comprehension context as a whole. Therefore, the visualization approach proved as a valid addition to and advancement of the current research body.

12

Extensibility Evaluation: Integrating a Control Center Concept

After evaluating the external quality attributes of ExplorViz in the last two chapters, we now focus on the internal quality attributes of our implementation. For this purpose, we developed a semi-automatic control center concept [Fittkau et al. 2014b] and let project-external developers integrate this concept into ExplorViz and thus extend it. This approach tests several internal quality attributes of our ExplorViz tool.

First, we define the goals. Then, we describe the envisioned control center concept in Section 12.2. Afterwards, Section 12.3 presents the case study where project-foreign developers integrated this concept into ExplorViz. Finally, we summarize the results of our extensibility evaluation.

Previous Publications

Parts of this chapter are already published in the following work:

1. [Fittkau et al. 2014b] F. Fittkau, A. van Hoorn, and W. Hasselbring. Towards a dependability control center for large software landscapes. In: *Proceedings of the 10th European Dependable Computing Conference (EDCC 2014)*. IEEE, May 2014

12.1 Goals

To enable other researchers to modify or extend our ExplorViz approach and tool, a high internal quality is crucial. To evaluate the internal quality, we let project-external developers integrate a control center concept. With this approach we aim to show that our ExplorViz implementation has high internal quality aspects such as its software architecture, documentation, interface definitions, and coding style.

12.2 Control Center Concept

From our experience, users often mistrust fully-automatic capacity management tools. Building trust is an open research challenge [Salehie and Tahvildari 2009] in this area. Users are missing the control of the automatic changes conducted to the software landscape. To tackle this problem, we envision a semi-automatic control center.

This section introduces our semi-automatic control center concept based on a common threat and management strategy for system dependability: software aging and rejuvenation. Software aging denotes the phenomenon that software components, when executed over a longer period of time, tend to show degradation effects [Avizienis et al. 2004]. These can manifest themselves in slightly increasing response times or memory consumption. Possible causes are, for instance, unreleased resources, whose impact accumulates over time. A common reactive or proactive resolution strategy, known as software rejuvenation, is the restart of selected system components.

As an example software landscape to demonstrate our concept in a usage scenario, we utilize a system that provides publication workflows for scientific data, called PubFlow[1] [Brauer and Hasselbring 2013b]. It comprises a heterogeneous architecture of distributed applications and hence provides a well-fitting example system for our usage scenario. We assume that the response times of one PubFlow service increase over time and threaten the fulfillment of SLAs. The responsible application needs

[1] http://www.pubflow.de

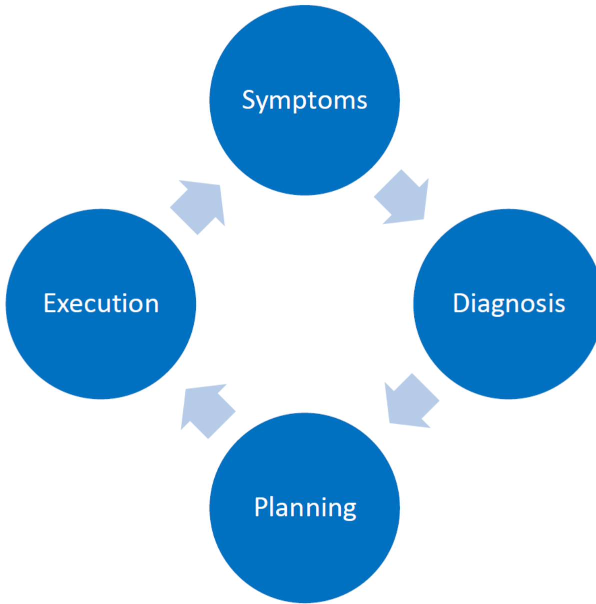


Figure 12.1. Cycle of the four activities in our control center

to be determined, and a restart of this application needs to be planned and executed. In the remainder of this section, we describe our envisioned control center concept according to the activities involved in maintaining the dependability of software landscapes (see Figure 12.1), i.e., viewing the symptoms (*symptoms perspective*), determining the root cause (*diagnosis perspective*), planning a countermeasure (*planning perspective*), and executing it (*execution perspective*).

12.2.1 Anomaly Detection

The *symptoms perspective* is the control center's starting perspective. It provides an overall view on the software landscape, including notifications provided by control center plug-ins. The user can observe the execution of the monitored applications and monitor their status with respect to

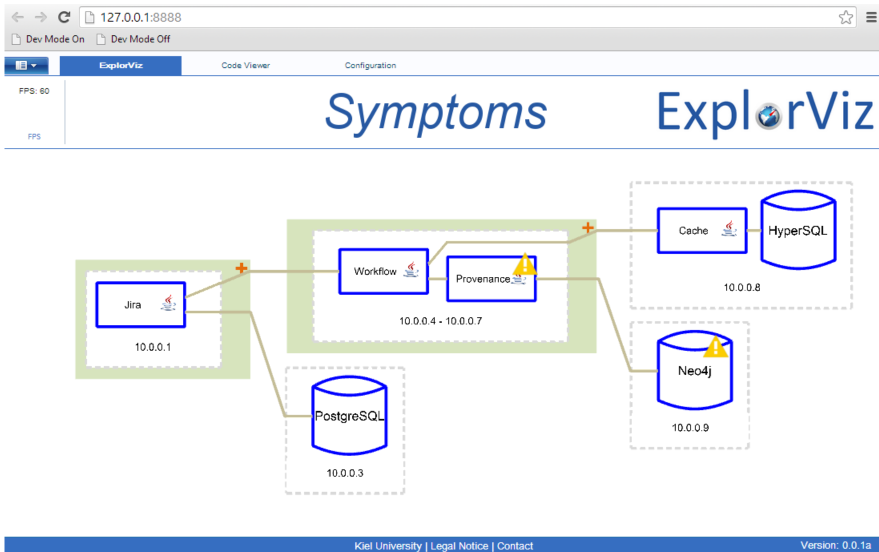


Figure 12.2. Mockup of the symptoms perspective visualizing PubFlow

dependability. Using ExplorViz' interactive visualization approach, it is possible to dive into single applications – similar to [Ehlers et al. 2011] – to get an application-level view of its components. This will be exemplified in the following diagnosis perspective.

Figure 12.2 shows a mockup of the symptoms perspective including a visualization of the PubFlow system. When an abnormal state is detected by a plug-in, the corresponding application is marked with a warning symbol. In this case, we see that both the Neo4J and the Provenance application are marked with a warning sign as a hint for further investigation.

12.2.2 Root Cause Analysis

As we want to find the root cause of this warning, we open the control center's *diagnosis perspective*. Here, the user is guided by automatic tools to find the root cause of a dependability problem – as opposed to the

previously described perspective which only showed its symptoms. Similar to the symptoms perspective, notifications are provided by respective plug-ins.

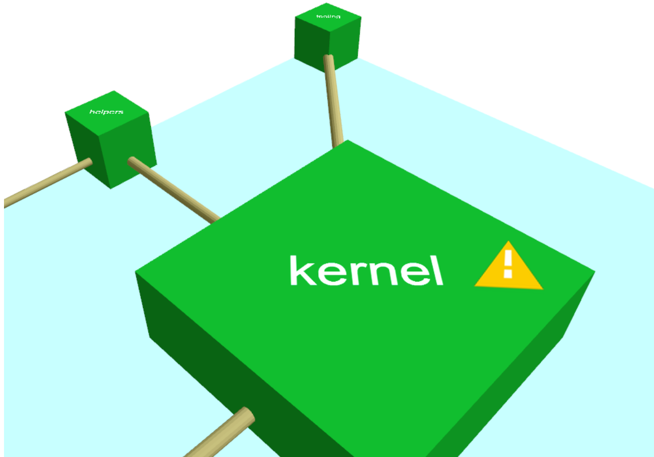
In our scenario, a diagnosis tool marks the Neo4J application as the root cause of the abnormal behavior. We want to further investigate this circumstance and thus jump into the application to get a detailed view which component of Neo4J is causing the abnormal response times. Figure 12.3 shows a mockup of the application-level perspective of Neo4J. The component `kernel` is marked with a warning sign. By jumping further into the `kernel`, we can see that the subcomponent `impl` is responsible for the warning.

To get further insights, we analyze the average response time of the `impl` component. The average response time and its corresponding anomaly score are sketched in Figure 12.4. The gray bars represent the average response times in time windows of five minutes. The green series in the plot represents the predicted response time basing on past behavior. The anomaly score is shown by a blue line chart above the bar chart. The thresholds for a warning and for an error are displayed in yellow and red. From the current rising response times and the normal nearly constant response times – assuming other parameters like the workload intensity to be constant – we conclude that a software aging problem exists and thus we have to trigger proactive countermeasures to ensure the Quality of Service (QoS).

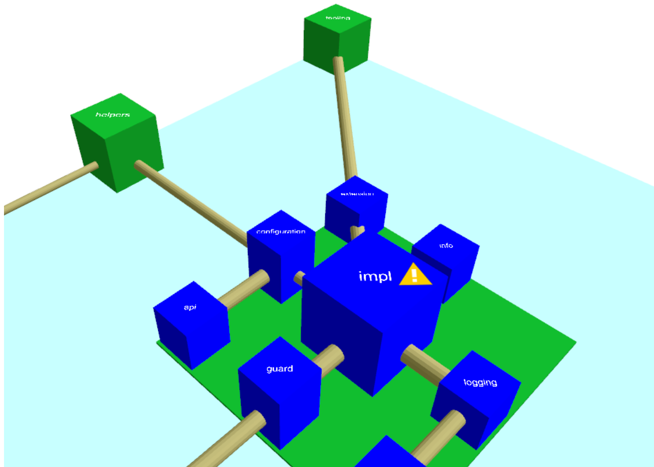
12.2.3 Resource Adaption Planning

After the diagnosis, we want to act proactively to ensure that a failure will not occur. For this purpose, we envision a semi-automatic *planning perspective* for defining or changing an adaptation plan.

Two possibilities for the opening of the planning perspective will exist. The first one is by manually creating a reconfiguration plan. The second one is an automatic suggestion by a plug-in on how to reconfigure the software landscape if needed. When the plug-in has a suggestion after detecting a low QoS, a dialog is displayed to the user. It first shows the low QoS results and a short summary of how it is suggested to adapt the landscape. As a



(a) Application-level perspective of Neo4J with warning sign on component `kernel`



(b) Opened `kernel` component with warning sign on component `impl`

Figure 12.3. Mockup of application-level perspective of Neo4J in the diagnosis perspective

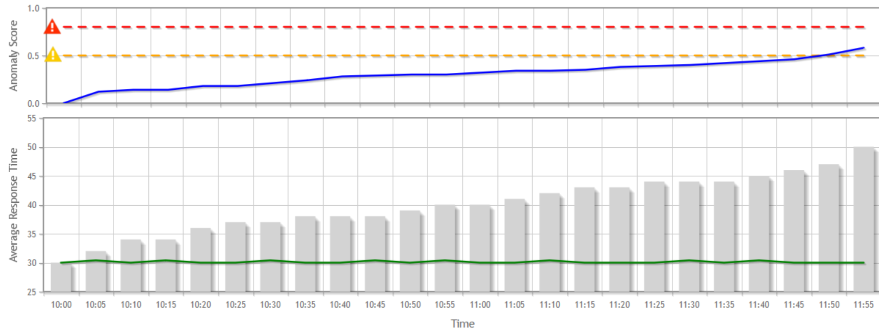


Figure 12.4. Mockup of average response times (bars) and corresponding anomaly scores (upper part) for the `impl` component

further indicator, the new QoS results and the resulting costs are displayed, e.g., by using model-based software performance prediction techniques [Cortellessa et al. 2011]. The user has the ability to directly execute this plan or to manually refine it.

In our scenario, a dialog is shown reading *"The software landscape violates its requirements for response times. It is suggested to start a new node of type 'm1.small' with the application 'Neo4J' on it. After the change, the response time is improved and the operating costs increase by 5 Euro per hour"*. In our usage scenario, we choose to manually adapt the plan which opens the planning perspective. After opening the planning perspective, the reconfiguration plan that was computed by a plug-in is visualized (see Figure 12.5).

In the planning perspective, the user has the possibility to manually refine the reconfiguration plan. The possibilities in the planning perspective are sketched in Figure 12.6. The user can, for instance, restart, terminate, or replicate applications. In our scenario, we have expert knowledge about the situation such that we know that the existing Neo4J application can simply be restarted to act against the software aging. Thus, we rely on our knowledge that the Neo4J application is not critical and can be restarted without violating the SLAs.

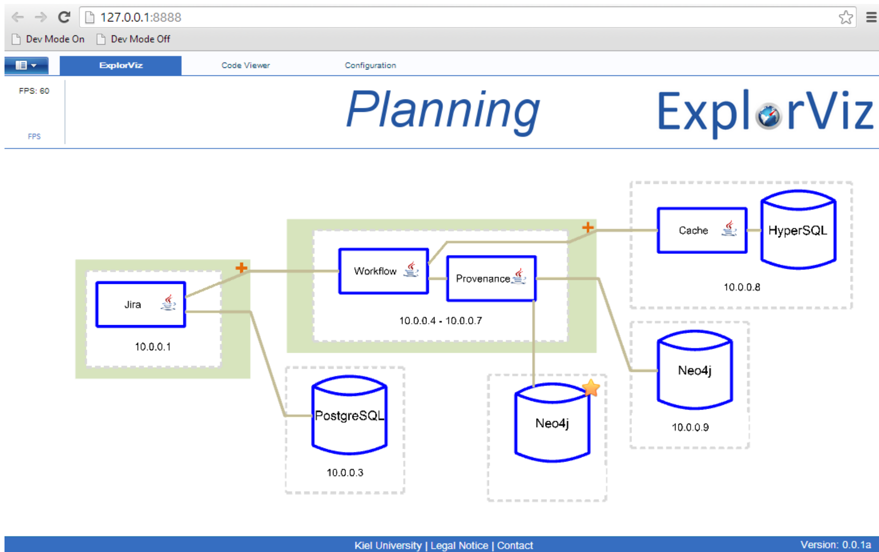
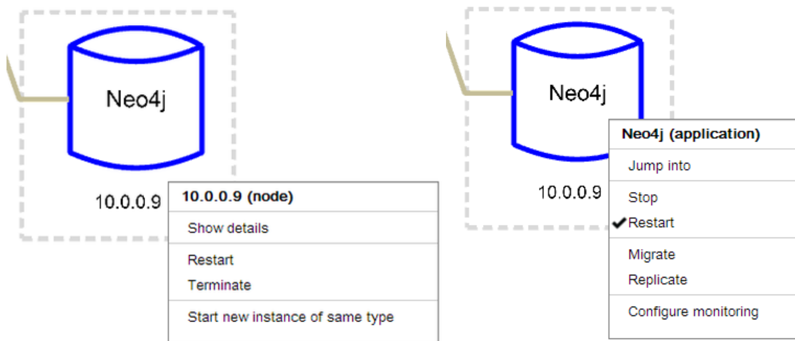


Figure 12.5. Mockup of planning perspective



(a) Node context menu

(b) Application context menu

Figure 12.6. Mockup of reconfiguration options in the planning perspective

12.2.4 Adaption Plan Execution

The actual execution of the reconfiguration plan is triggered by pressing an *execute* button in the planning perspective. The plug-in used for this execution should act in a transactional fashion for this step. If the transaction fails, it should rollback the transaction and a message to the control center should be triggered.

After triggering the execution, the *execution perspective* is opened automatically. During the actual execution of the reconfiguration plan, the execution perspective shows what is planned and what steps of the plan have already been conducted. In our example, a restart of the Neo4J application is triggered and during the restart of the application, the perspective shows a blinking Neo4J application indicating that the application is currently starting.

12.3 Case Study

After presenting the concept, we now describe a case study where project-external developers integrated this concept into ExplorViz. At first, we describe the setup. Afterwards, the results are discussed.

12.3.1 Setup

The task to integrate the previously described control center concept was given by us to three bachelor students and four master students. The task had to be solved in the context of a bachelor's project in the case of the bachelor students and in the context of a master's project in the case of the master students. Both projects had to solve the task as joint work.

The success of the project and the created plug-ins form the basis of our evaluation. In addition, we employed a debriefing questionnaire after the project took place to rate the students' experiences with extending ExplorViz. The questionnaire was given to the students after their final presentation. A research assistant collected the filled-in questionnaires and also created the average ratings from the results such that the students from the project could give their ratings and comments anonymously.

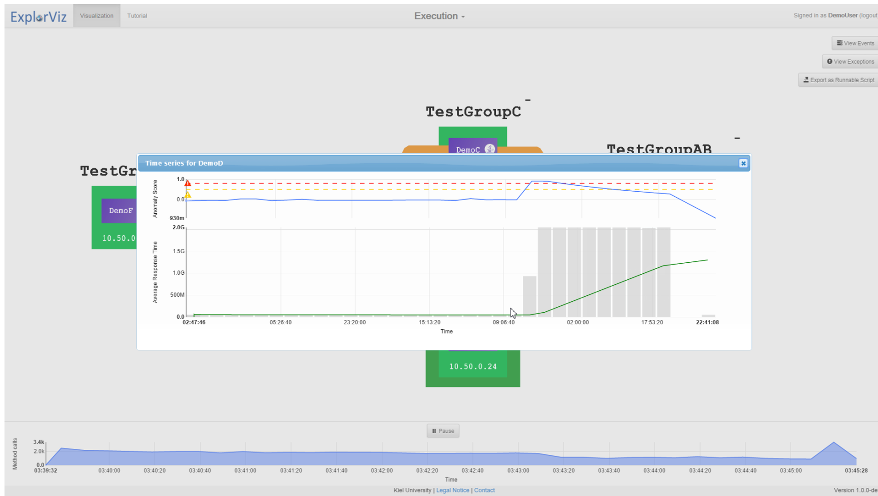


Figure 12.7. Time series showing a detected anomaly

12.3.2 Results and Discussion

There are two artifacts that provide crucial information for our extensibility evaluation. At first, these are the created plug-ins of the students and secondly, the results from the debriefing questionnaire.

Created Plug-Ins

In this section, we describe the created plug-ins. For verifiability, they are available in the “control center”-branch of the ExplorViz’ Git repository.

For detecting anomalies in the symptoms perspective, the students used an adapted version of OPADx [Frotscher 2013] and integrated the source code of it into ExplorViz. Furthermore, they eliminated the R script server dependency for reasons of end user convenience. Detected anomalies are visualized by a warning or error sign as was already presented in the concept (see Figure 12.2). Figure 12.7 shows a time series over the response times of the application *DemoD*. The anomaly score is represented by the blue line in the upper part of the screenshot. It further visualizes a detected

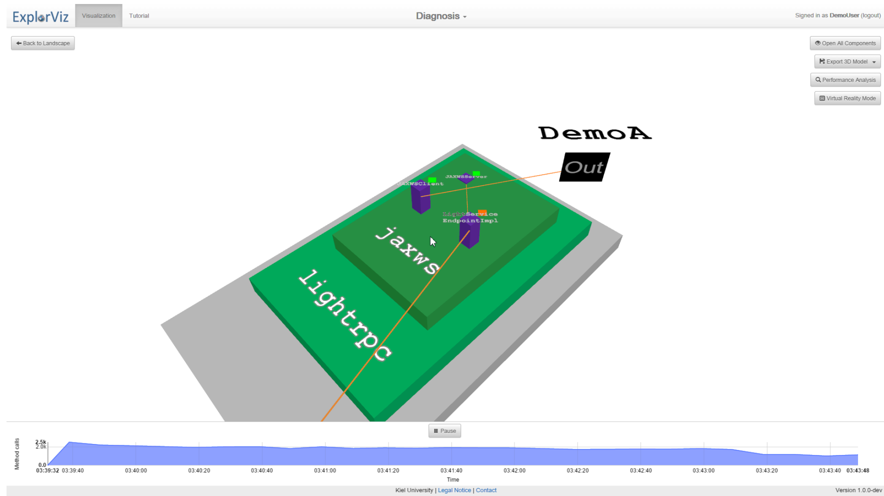


Figure 12.8. Diagnosis perspective on the application level

anomaly at the start of the abrupt rising response times. Details about the actual used algorithms can be found in [Mannstedt 2015].

After detecting the anomalies, the diagnosis perspective provides details about the root cause. For this root cause detection, the students utilized RanCorr [Marwede et al. 2009] and also integrated its source code into ExplorViz. Figure 12.8 displays the marking of a potential root cause. It is marked with an addition of a color gradient field from green to red (less probable and highly probable for causing the anomaly). For details about the root cause detection algorithm, we refer to [Michaelis 2015].

After analyzing the root cause, an automatic capacity plan suggestion dialog is shown (see Figure 12.9). The suggestion originates from our capacity manager *CapMan* which was integrated and extended to first suggest the plan and not automatically conducting it. For details about this integration and extension, we refer to [Gill 2015]. After the suggestion dialog, the plan can be manually refined and executed. The execution perspective opens up automatically and shows what is currently executed and what is still planed by blinking when a server is started or restarted.

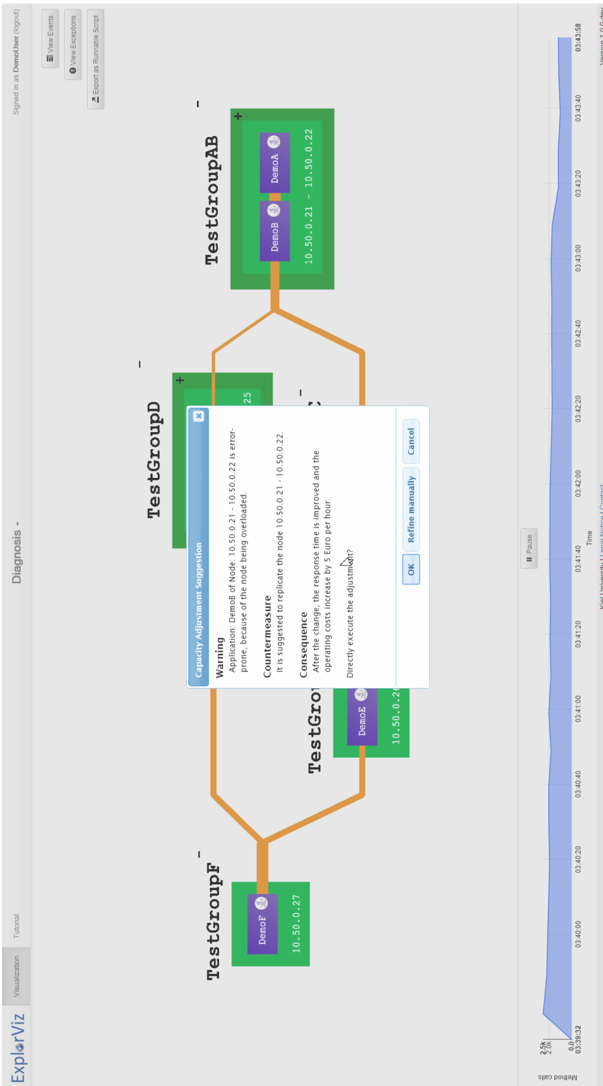


Figure 12.9. Automatic capacity adjustment suggestion dialog

Table 12.1. Debriefing Questionnaire Results for our Extensibility Evaluation

	mean	stdev.
Java experience (1-5)	2.63	0.48
Software architecture experience (1-5)	2.13	0.60
ExplorViz experience (1-5)	0.13	0.33
How much changing interfaces/classes (1-5)	2.63	0.86
Difficulty in changing interfaces/classes (1-5)	2.00	0.71
Difficulty integrating features (1-5)	2.63	0.99
Overall project impression (1-5)	1.50	0.50

Debriefing Questionnaire Results

The results of the debriefing questionnaire are shown in Table 12.1. The questionnaire is provided in Appendix A.1. The average self-rated programming experience with Java and the average self-rated experience with software architectures was between *Intermediate* and *Advanced*. All students had no or only little experience with ExplorViz. From our personal experience, we know that none of them had programmed in ExplorViz before. The tendency of how much they had to change the interfaces/classes was slightly towards *none* from the neutral center. The difficulty for changing the interfaces/classes was rated with *easy*. Whereas the difficulty for integrating the features was between *easy* and the neutral center. The overall impression of the project was between *very good* and *good*. As the only qualitative feedback, the students wished for more comments in the source code.

12.4 Summary

In summary, the students succeeded to implement the control center concept. One large challenge during the project was the comprehension of the existing tools (OPADx, RanCorr, and CapMan) for each phase which should be integrated into ExplorViz. Furthermore, since the students had only

little knowledge about cloud computing before the project, the execution of a demo scenario took a large amount of time. In respect to ExplorViz, they only had few questions and often only an attribute name of a data class was misleading. Since the students succeeded – in spite of the two aforementioned challenges –, this shows that the actual integration and extension of ExplorViz did not take too much time.

Furthermore, the students self-rated that they did not have to change many interfaces or classes of ExplorViz and when they had to, this was easy for them. In general, the integration of features into ExplorViz was rated as rather easy and the students enjoyed the project altogether. If the internal quality of ExplorViz would be poor, this would probably not let to an enjoyable project.

Due to the aforementioned points, we conclude that ExplorViz is extensible and provides a good internal quality which is important when other researchers intend to modify or extend our tool.

13 Related Work

In this chapter, we discuss related work for our thesis. We start by describing related work to our monitoring and trace processing approach (Section 13.1). Then, related work to our visualization approach is presented (Section 13.2). Finally, we discuss related work to our evaluations (Section 13.3).

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013

2. [Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013

3. [Fittkau et al. 2014a] F. Fittkau, P. Stelzer, and W. Hasselbring. Live visualization of large software landscapes for ensuring architecture conformance. In: *Proceedings of the 2nd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2014)*. ACM, Aug. 2014

4. [Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. IEEE, May 2015

5. [Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015

6. [Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015

7. [Fittkau et al. 2015f] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

13.1 Monitoring and Trace Processing

The related work to our monitoring and trace processing approach are detailed in the following.

13.1.1 Monitoring

Kieker [van Hoorn et al. 2012] provides application-level monitoring and separates the process into monitoring and analysis. It is developed as a framework and therefore, provides, e.g., several different methods for writing the monitoring logs to the analysis component. In contrast, ExplorViz only provides one TCP writer since we target out-of-the-box usage of our monitoring solution.

SPASS-meter [Eichelberger and Schmid 2014] also provides application-level monitoring. A special feature of SPASS-meter is the monitoring of resource consumption of self-defined components or services. Contrary, we focus on the conducted method calls to support in the program and system comprehension process.

Dapper [Sigelman et al. 2010] is used for Google's production distributed systems tracing infrastructure and provides scalability and application-level tracing in addition to remote procedure call tracing. Instead of using sampling techniques, our ExplorViz approach uses entire invocation monitoring. Furthermore, contrary to us, they did not present any detailed description of how they actually provide scalability.

A further distributed tracing tool is *Magpie* [Barham et al. 2003]. It collects traces from multiple distributed machines and extracts user specific traces. Then, a probabilistic model of the behavior of the user is constructed. Again, we did not find a detailed description of their architecture how they monitor and process traces.

X-Trace [Fonseca et al. 2007] provides capabilities to monitor different networks, including, for instance, VPNs, tunnels, or NATs. It can correlate the information gathered from one layer to the other layer. In contrast to network monitoring, we target application-level monitoring.

APM tools, such as *AppDynamics*, *dynaTrace*, or *CA Wily Introscope*, inherently also provide monitoring capabilities. In contrast to our approach,

they often feature sampling techniques to reduce and adapt the monitoring overhead. Furthermore, since most tools are commercial, the employed techniques for the monitoring are often not described.

13.1.2 Trace Processing

Brunst and Nagel [2003] present a parallel analysis infrastructure. They focus on massive parallel systems with thousands of processor cores. In contrast, we focus on the monitoring and analysis of applications running on typical business servers or in cloud environments.

Meng et al. [2010] propose a Monitoring-as-a-Service solution for monitoring cloud infrastructures. To monitor the complex infrastructure of Cloud data centers, they developed a scalable and flexible monitoring topology consisting of different services. Compared to our trace processing approach, they focus on monitoring the virtualized data center environment.

Hilbrich and Muller-Pfefferkorn [2012] describe a concept of a scalable job centric monitoring infrastructure. Their approach features multiple layers of short and long time storage of the monitored data. Contrary, we directly analyze the monitored data after gathering it without a persistent storage.

Vierhauser et al. [2014] target Systems-of-Systems architectures with their flexible monitoring and analysis framework. In contrast to our monitoring and analysis approach, they do not dynamically insert or remove preprocessing levels in the analysis.

The *ECoWare Infrastructure* [Baresi and Guinea 2013] consists of three types of components, i.e., the execution environment, processors, and a dashboard. In contrast to our trace processing approach, they use a message bus for their analysis and do not provide multiple analysis levels.

In general, our trace processing approach exhibits similarities to the *MapReduce pattern* [Dean and Ghemawat 2010] and approaches using this pattern for data processing such as [Aher and Lambhate 2015]. However, in contrast, it does not dynamically insert or remove preprocessing levels according to the actual workload.

Capacity management approaches utilizing the monitored data for their scaling decisions, e.g., *SLAstic* [van Hoorn et al. 2009a], are also related

to our approach, since they must analyze the monitored data just after it was observed. To the best of our knowledge, none of these approaches utilizes dynamically inserted or removed worker levels as used by our trace processing approach.

13.2 Visualization

In this section, related work concerning our visualization is discussed. We start by describing related work to our landscape-level perspective and our application-level perspective. Then, related work to our VR approach is presented. Finally, related work in the domain of Enterprise Architecture (EA) and Enterprise Model (EM) are shown.

13.2.1 Landscape-Level Perspective

Web Services Navigator [De Pauw et al. 2005] provides 2D graph visualizations of the communication of web services. *Streamsight* [De Pauw et al. 2008] visualizes cooperating distributed components of streaming applications. In contrast to both tools, our approach aims at general software landscapes and introduces interactive explorable hierarchies to provide a higher visual scalability.

APM tools also often provide a visualization of a software landscape. Most APM tools do not provide different abstraction levels at the software landscape visualization. Furthermore, if the APM tool allows detailed analysis of one application, the used visualization often is a tree-based viewer, which can hinder the analysis of traces with thousands of events.

Briand et al. [2006] utilize UML sequence diagrams to visualize RPCs by adding a hostname to the object representation. Single method calls are shown in the diagrams. In contrast, we provide a single relation entity between communicating applications and therefore a higher visual scalability.

RanCorr [Marwede et al. 2009] visualizes the dependencies between applications in a root cause analysis scenario. The root cause probability of

each application is visualized by a color-coding. In contrast to them, our approach uses hierarchies to provide a higher visual scalability.

VisuSniff [Oechsle et al. 2002] shows the communication between servers. In contrast to our approach, they visualize every communication path on each port and protocol. Therefore, communicating servers often have several connected communication lines and thus the visualization does not scale to large software landscapes.

13.2.2 Application-Level Perspective

Since a large amount of approaches for the visualization of single application exist, we focus on the 2D-based and 3D-based visualization of execution traces.

2D-Based Visualization of Execution Traces

Jive and *Jove* [Reiss and Tarvo 2012] visualize Java applications during their execution. They use a 2D visualization to achieve live trace visualization. Contrary to these tools, our approach utilizes the city metaphor to visualize execution traces.

EXTRAVIS developed by Cornelissen et al. [2007] visualizes single execution traces in two synchronized views, namely a circular bundle view and a massive sequence view. The former view utilizes hierarchical edge bundles to display the interaction of the execution trace. Trümper et al. [2012] visualize traces in a sequence visualization with sub ranges for details. In contrast to both approaches, ExplorViz uses an exploration-based approach for displaying execution traces by utilizing the software city metaphor.

A large part of trace visualization approaches are similar to UML sequence diagrams [Briand et al. 2004]. Inherent to UML diagrams, they often do not provide a large visual scalability. Therefore, we use the software city metaphor for visualizing the gathered execution traces in our approach.

3D-Based Visualization of Execution Traces

Balzer and Deussen [2004] provide a visualization of relations in 3D using a landscape metaphor based on hemispheres. They define the concept of a

Hierarchical Net which substitutes a group of entities with a single entity to display relations. *Call Graph Analyzer* [Bohnet and Döllner 2006] combines the static structure and dynamic properties of a software system in a single 3D view. Traces are visualized in a live fashion during the runtime of the software system. *TraceCrawler* [Greevy et al. 2006] visualizes prerecorded execution traces for one feature based upon a 3D graph metaphor. Caserta et al. [2011] utilize the hierarchical edge bundling technique for visualizing relations in the city metaphor of a single software system. In contrast to the former approaches, ExplorViz substitutes groups of objects by single objects for interactively exploring relations and structures.

EvoSpaces [Alam and Dugerdil 2007] represents the underlying application utilizing a 3D city metaphor. An important feature is the day view for static analysis and the night view for dynamic analysis. We also use two perspectives to visualize different properties. However, we visualize landscape and application-level issues.

DyVis [Wulf 2010] uses the city metaphor to visualize the structure and an execution trace of an application. In contrast to our ExplorViz approach, DyVis requires an upfront static analysis of the application and it only visualizes one trace at a time.

SynchroVis [Waller et al. 2013] developed by Döhring [2012] aims for visualizing concurrent behavior of an application. Similar to DyVis, it uses the city metaphor to visualize the static structure of a program. In addition, it uses special buildings to show, for instance, locked objects. Contrary to our approach, it requires an upfront static analysis and does not use interactivity to provide visual scalability.

13.2.3 Virtual Reality for Software Visualization

In this subsection, we describe related work of VR and augmented reality approaches for software visualization.

Imsovision [Maletic et al. 2001] represents object-oriented software in a VR environment. Electromagnetic sensors, which are attached to the shutter glasses, track the user and a *wand* is used for 3D navigation. In contrast to them, we use a hands-free gesture recognition in our VR approach.

SykscrapAR [Souza et al. 2012] aims to represent software evolution by utilizing an augmented reality approach employing the city metaphor. The user can interact with a physical marker platform in an intuitive way. Contrary to our VR approach, the user only sees the 3D model on a monitor.

Delimarschi et al. [2014] introduce a concept of natural user interfaces for IDEs by using voice commands and gesture-based interaction with a Microsoft Kinect. In contrast, they do not utilize HMDs to create an immersive VR experience.

Young and Munro [1998] developed *FileVis* which visualizes files and provides an overview of the system. Although they aim for virtual environment, no technological approach is described, e.g., no HMDs or gesture recognition sensor.

13.2.4 Enterprise Architecture and Models

In this section, we describe related work in the domain of EA and EM.

Surveys on EA tools [Matthes et al. 2008; Roth et al. 2014] are provided by the research community around Matthes. To gather information for an EM, they propose – similar to use – to extract data from existing information sources within an enterprise [Hauder et al. 2012]. To analyze the resulting EMs, they provide ad-hoc analyses means [Roth and Matthes 2014]. In contrast to their approach, we advocate that also details about the visualized applications matter, for example, in a performance analysis.

The research community around Leymann also investigates the gathering and visualization of infrastructure information [Binz et al. 2013]. For this purpose, they use Enterprise Topology Graphs [Binz et al. 2012]. In contrast to our approach, they do not provide details in the event of a potential bottleneck in the software landscape.

Many other researchers investigate EMs. For example, Frank [2011] presents a multi-perspective view on EMs. Fischer et al. [2007] gather information from different stakeholders for their EM. However, they do not provide details about how to efficiently gather the information.

Breu et al. [2011] present “living models” aiming to provide a coherent management, design, and operations of IT. The created artifacts are viewed

separately. In contrast, our approach enables to bring design and operation together by, for instance, architecture conformance checking.

13.3 Evaluation

In this section, we discuss related work on experiments with software visualization tools in a program comprehension context.

13.3.1 Experiments Comparing to an IDE

In this subsection, we list related experiments that compare visualizations to an IDE. In general, contrary to the experiments, we compare two visualization techniques without the use of an IDE to investigate the efficiency and effectiveness of the visualization techniques in comparison. Further differentiations are presented in each experiment discussion.

Marcus et al. [2005] present a controlled experiment comparing Visual Studio .NET and *sv3D* for program comprehension. The usage of sv3D led to a significant increase in the time spent. In contrast to our experiments, they compared a visualization basing upon static analysis.

Quante [2008] assessed whether additionally available *Dynamic Object Process Graphs* provide benefits to program comprehension when using Eclipse. The experiment involved two object systems and for the second system the results were not significant. In contrast to our experiments, the author investigated only the additional availability of a visualization.

Wettel et al. [2011] conducted a controlled experiment to compare the usage of Eclipse and Excel to their tool *CodeCity* with professionals. They found a significant increase in correctness and decrease of time spent using CodeCity. They compared a single software visualization basing upon static analysis to Eclipse and Excel, contrary to our comparisons of different software visualizations.

Sharif et al. [2013] present a controlled experiment comparing Eclipse and the additional availability of the *SeeIT 3D* Eclipse plug-in using eye-tracking. The experimental group with access to SeeIT 3D performed significantly better in overview tasks but required more time in bug fixing

tasks. In contrast to our experiments, they investigated only the additional availability of SeeIT 3D basing on static analysis.

Cornelissen et al. [2009]; Cornelissen et al. [2011] performed a controlled experiment for the evaluation of Extravis to investigate whether the availability of Extravis in addition to Eclipse provides benefits. The availability of Extravis led to advantages in time spent and correctness. In contrast, we compare different software visualization techniques in our experiments.

13.3.2 Experiments Comparing Software Visualizations

Storey et al. [1997] compared three static analysis tools in an experiment. The authors performed a detailed discussion of the tools' usage but provided no quantitative results.

Lange and Chaudron [2007] investigated the benefits of their enriched UML views by comparing them with traditional UML diagrams. In contrast, we compared trace visualization techniques.

Stevanetic et al. [2015] also investigate the effect of hierarchies on program comprehension by using three different documentations with UML diagrams. The first group received a low-level documentation, the second group used a high-level documentation, and the third group utilized a hierarchical documentation. They also conclude that hierarchies increase the quality of the solutions.

Part IV

Conclusions and Future Work

14 Conclusions

In this thesis, we presented our live trace visualization approach, named ExplorViz, for large software landscape. In addition, we showed how to tackle the challenge of monitoring and processing the typical huge amount of conducted method calls in such large landscapes. Furthermore, we developed alternative display and interaction concepts for the software city metaphor. In this chapter, we summarize the thesis according to the monitoring and trace processing, and the visualization. Afterwards, the results of our evaluations are summed up.

14.1 Monitoring and Trace Processing

A low overhead and fast trace processing for the monitoring and analysis approach are prerequisites for our live trace visualization. In this thesis, we presented how we monitor the method calls using minimal invasive aspect-oriented concepts as provided by, for example, AspectJ. Furthermore, we described the general approach how we monitor the communication between applications, i.e., RPCs, using a backpacking strategy for correlating the trace identifiers.

For providing a scalable, elastic analysis of the monitored data and to enable a live processing of the traces, we defined an approach similar to the MapReduce pattern where we use worker nodes as preprocessors for one master node. In contrast to the MapReduce pattern, our approach dynamically inserts and removes preprocessing worker levels during runtime of the analysis. Therefore, elasticity is provided and costs for the analysis of the monitored data are reduced, when only a small amount of method calls occur in the monitored software landscape.

14.2 Visualization

In this thesis, we presented our live trace visualization for large software landscapes which enables the visualization of a software landscape in combination with the details of each application. Therefore, we feature two perspectives, namely the landscape-level and the application-level perspective.

The landscape-level perspective uses 2D entities similar to UML deployment diagrams to visualize the communication, nodes, and applications contained in a software landscape. In addition, we use hierarchies to provide visual scalability. Nodes running the same application configuration are grouped into node groups. Furthermore, nodes and node groups are visualized in their virtual organization unit namely systems. Systems and node groups can be interactively opened and closed such that only details of interest for the current analysis are visualized.

To provide visual scalability on the application level, we feature the software city metaphor on the application-level perspective to visualize one application and its runtime information. Components, e.g., packages in the context of Java, provide the hierarchical districts of our city metaphor. They can either be opened – showing their internal details – or they can be closed and thus hiding their internal details. Again, this interactivity provides visual scalability. In our city metaphor, classes are represented by buildings and the active instance count maps to their height.

Since we provide a live trace visualization, we also feature a time shift mode where the user can jump to previous landscape visualizations. Then, she is able to analyze a specific interval and traces of the software landscape. For instance, she can look at several applications to understand the situation why a performance anomaly occurred.

To enable other researchers to build up on the gathered monitoring data, we described our landscape meta-model. The generated software landscape model could be used, for instance, to adapt the current instances in the software landscape or to generate reports of the landscape health.

Furthermore, we presented new concepts to display and interact with models following the software city metaphor. We showed a VR approach utilizing an Oculus Rift DK1 display and a Microsoft Kinect v2 to enable gesture-based interaction with the city model. In addition, we described our approach to create physical 3D-printed models following the software city metaphor. Four potential usage scenarios were shown for these physical models, inter alia, a team-based program comprehension scenario.

14.3 Evaluation

We implemented the aforementioned approaches and evaluated each for several aspects. Every component is available as open source software on Github.[1]

To evaluate our monitoring approach, we used the monitoring benchmark MooBench to compare our monitoring component to the monitoring framework Kieker. Eichelberger and Schmid [2014] already showed that

[1] https://github.com/ExplorViz

Kieker imposes a low overhead. Therefore, if our approach imposes a lower overhead, our monitoring component also provides a low overhead. The comparison showed that our monitoring approach achieves a speedup of about factor nine and decreases the overhead by about 89 %. Therefore, our monitoring imposes a low overhead to keep the impact on the production systems as low as possible.

A second lab experiment evaluated whether our analysis approach – including the steps trace reconstruction and trace reduction – is capable of processing the monitored data in a live fashion, i.e., not or only minimally decreasing the throughput, if the analysis is added after the monitoring component. Therefore, we extended the monitoring benchmark MooBench by adding both analysis steps as extra phases. Again, we compared the results to the benchmark results of Kieker to provide a quantitative relationship of our enhancements. The evaluation showed that adding our analysis after the monitoring only negligibly impacts the processed throughput. Therefore, our analysis approach is capable of live processing the gathered monitoring data which is an important prerequisite for a live trace visualization. Furthermore, our analysis approach provides a speedup of about factor 250 in comparison to the analysis of Kieker.

The third lab experiment concerning the monitoring and trace processing approach had the goal of showing the scalability and elasticity of our analysis approach. Therefore, we utilized our private cloud to elastically scale and monitor up to 160 JPetStore instances. During the 24 hours experiment, two worker levels were inserted and removed dynamically without pausing the actual analysis. In the peak, the analysis processed 20 million monitored method calls per second. Since the analysis elastically scaled in accordance to the JPetStore instances, the evaluation showed that our analysis approach provides a scalable, elastic, and live processing of the monitored data.

Since we showed that our developed monitoring and analysis approach is capable of elastically processing the gathered data, our goal G2 of the thesis (see Chapter 4) is fulfilled.

Concerning the visualization, we conducted four controlled experiments to show that our visualization approach enhances the current state-of-the-art. The first experiment compared our application-level perspective to the

trace visualization tool ExtraVis in typical program comprehension tasks analyzing the architecture of PMD. The experiment resulted in a significant decrease of 28 % of time spent and an increase in correctness by 61 %, using our application-level perspective.

We replicated the design of our first controlled experiment in a second controlled experiment. In contrast to the first controlled experiment, we changed the object system to the smaller-sized Android application Babsi. The time spent for solving the program comprehension tasks were similar in both groups. However, the use of ExplorViz significantly increased the correctness by 39 %. Therefore, the application-level perspective of ExplorViz showed more effective in both experiments and more efficient in the PMD experiment.

In the third controlled experiment, we evaluate whether physical 3D-printed models provide benefits in a team-based program comprehension scenario. For this evaluation, one group used a physical model of PMD and the other group used the virtual model at the computer screen to solve program comprehension tasks. Since the effects of each task compensated each other, there was no overall effect of using physical models. However, two tasks were positively influenced and one task was negatively influenced. Thus, physical models have beneficial effects on some program comprehension tasks which should further be investigated in future work.

Since we developed and evaluated different display and interaction concepts for the software city metaphor, our goal G3 of the thesis is fulfilled.

A fourth controlled experiment investigated whether the hierarchical landscape-level perspective provides benefits compared to current flat landscape visualization for solving system comprehension tasks. Since most landscape visualizations are part of commercial APM tools, we surveyed them and implemented a mix of the best concepts for comparing the resulting flat visualization to our hierarchical visualization. The time spent was similar between both groups. However, the correctness of the solutions increased by 14 % showing that our hierarchical additions provide benefits.

The controlled experiments revealed that our live trace visualization supports in the program and system comprehension of large software landscapes. Therefore, our main goal G1 of the thesis is fulfilled.

To evaluate the internal quality of our implementation, we let external developers extend our implementation by a prescribed control center concept. They succeeded in this task although they had only little knowledge about the technologies which should be used in the control center. Therefore, we conclude that our ExplorViz implementation is extensible by external developers which is important when other researches intend to change or experiment with our implementation.

15

Future Work

In this chapter, we describe possible future work. At first, work regarding the monitoring and trace processing is detailed. Afterwards, possible future work for the visualization is listed. Finally, further work regarding the evaluations is described.

Previous Publications

Parts of this chapter are already published in the following works:

1. [Fittkau et al. 2015j] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. Technical report 1507. Department of Computer Science, Kiel University, Germany, June 2015
2. [Fittkau et al. 2015f] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015
3. [Fittkau et al. 2015i] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015

15.1 Monitoring and Trace Processing

Future work lies in supporting the monitoring of more programming languages since our monitoring adapter adds an overhead to the actual monitoring. To impose a monitoring overhead as low as possible, a direct implementation for monitoring more programming languages, e.g., C, C++, C#, or Perl, should be provided. Regarding the RPC monitoring, more technologies should be supported to cover more technologies present in current software landscapes and thus to increase the applicability. Mining log files or even adapting monitoring data in a universal fashion is also future work. With this adaption, existing monitoring methods can be reused and thus already approved solutions can provide a higher user acceptance.

Furthermore, our described analysis approach should be enhanced with caching techniques and tested with hardware load balancers. Koppenhagen [2013] evaluated different cloud scaling strategies. Those strategies should also be implemented and tested in combination with our scalable and elastic trace processing approach.

15.2 Visualization

We structure the future work for the visualization according to the general approach, the landscape-level perspective, and the application-level perspective.

15.2.1 General Approach

A future research direction for our visualization is enabling collaborative working. Therefore, views should be savable and sharable with other users. Furthermore, users should be able to comment on entities and situations, and also be able to share such comments. One prerequisite is the already implemented user-based authorization and thus only the visual features need to be integrated and evaluated.

This user-based authorization should be further enhanced and developed to a real user management. This includes permission concepts for

who can access which entities. A possible extension could be a temporary role for external maintenance access. For example, if an application behaves abnormally or an exception is thrown, an external software engineer could access the visualization to investigate the circumstances under which the abnormal behavior occurred.

Further configuration options of our visualization should also be implemented. For example, a configuration option for the displayed interval and how long the history over the past landscapes should be persisted. A further option could be the configuration of the used colors. Then, users with a red-green-blindness would be able to configure different colors if they find the current color schema hard to perceive.

In general, our approach could be integrated in a continuous integration environment such as Jenkins.[1] With this integration, the visualization could always display the current state of the development. Furthermore, the architecture conformance checking could pass the differences between the actual and the conceptual architecture to Jenkins after each software build.

15.2.2 Landscape-Level Perspective

Our visualization can further be enhanced by featuring animations when new entities are added. In addition, an animation should be used when, for example, node groups get opened or closed. The animations will probably support in the comprehension process by providing context between the two states. Investigating this circumstance remains as future work.

According to our VR approach, the display should be extended from the application-level perspective to the landscape-level perspective. A possible implementation could visualize the landscape-level perspective by a curved cinema screen. Navigation could be achieved using the head rotation.

In respect to the control center concept, a what-if analysis could be integrated. For instance, the visualization could show the effected services when switching to an instance with a lower hardware specification. Furthermore, the what-if analysis could reveal critical infrastructure similar to RanCorr [Marwede et al. 2009] and perhaps even show countermeasures to circumvent the failure of the applications.

[1] https://jenkins-ci.org

15.2.3 Application-Level Perspective

Further future work lies in the layout for the application-level perspective. As already stated in Chapter 8, we did not find a satisfying layout solution which is compact, stable, and hierarchical. A first idea is adapting the approach of Wang and Miyamoto [1996] to achieve the hierarchical aspect.

The architecture conformance checking currently only operates at the landscape level. It should also be implemented for the application level to check the architecture conformance of individual applications.

Our clustering approach names synthetic components with "cluster" and a following increasing number. A naming according to the applied clustering criteria could further enhance the comprehension process of the synthetic components.

The performance analysis tool can also be enhanced with more configurable metrics, e.g., using the median instead of the mean for the response times. Furthermore, the components and classes could be colored according to their response times.

Further future work lies in finding approaches to overcome current limitations of our physical models. For instance, the naive mapping of communication lines to the physical models creates a set of confusing, overlapping lines – since missing the possibility to interactively hide communication. Creating physical models of other 3D software visualization metaphors, e.g., botanical trees [Kleiberg et al. 2001], spheres [Balzer et al. 2004], or solar systems [Graham et al. 2004], might also provide benefits.

Our VR approach should further be tested with other HMDs and with other display resolutions which should enable a higher immersion. Furthermore, the implementation of other input methods, e.g., brain interfaces, could achieve an even more immersive user experience.

15.3 Evaluation

Regarding the monitoring and trace processing, other application and hardware configurations should be tested. Especially, whether our elastic and scalable trace processing approach also functions with three and more dy-

namically inserted worker levels. Furthermore, our monitoring component should be directly compared to other monitoring tools with MooBench.

We listed four potential usage scenarios for our physical models of the application-level perspective. Since we only investigated one of them, the other three usage scenarios should also be evaluated. Furthermore, physical models could be compared to our VR approach to investigate which technique better supports the program comprehension process.

For each controlled experiment, future work should replicate the experiment with professional software engineers. Furthermore, we only investigated first-time usage in the experiments. Therefore, longer studies and experiment should investigate the long-term usage when a user already knows the features of the visualizations.

In respect to the application-level perspective evaluation, several other trace visualization techniques can be compared to it, for instance, the technique used by Trümper et al. [2010]. The physical model evaluation should be replicated with a larger team size where gesticulation and communication can have a higher impact.

Part V

Appendix

Appendix A

Experimental Data for the Extensibility Evaluation

A.1 Questionnaire

Florian Fittkau
Questionnaire: Evaluation of the Passed Master's Project

1 Personal Information

1.1 Target degree:

☐ Bachelor ☐ Master

1.2 Current semester ("Fachsemester"):

☐

1.3 Please rate your experience level **before** the master's project took place:

	None	Beginner	Intermediate	Advanced	Expert
Java Programming	☐	☐	☐	☐	☐
Software Architecture	☐	☐	☐	☐	☐
ExplorViz	☐	☐	☐	☐	☐

2 Questions

2.1 How much did you have to change the interfaces/classes provided by ExplorViz?

none ☐ ☐ ☐ ☐ ☐ everything

2.2 How difficult was it to **modify** the interfaces/classes provided by ExplorViz?
Note: only answer if applicable

very easy ☐ ☐ ☐ ☐ ☐ very difficult

2.3 How difficult was it to **integrate** the required features to ExplorViz?
Note: the integration and not the feature

very easy ☐ ☐ ☐ ☐ ☐ very difficult

2.4 Do you have any comments/suggestions concerning the programming within ExplorViz?

2.5 How is your overall impression of the master's project?

very good ☐ ☐ ☐ ☐ ☐ very bad

2.6 Do you have any comments/suggestions on the master's project?

List of Figures

2.1 Example execution trace and generated monitoring records . 34

3.1 Visualization pipeline based on [Card et al. 1999] 40
3.2 Software World (taken from [Anslow et al. 2006]) 42
3.3 3D City (taken from [Panas et al. 2003]) 43
3.4 ArgoUML visualized in CodeCity with annotated building archetypes (taken from [Wettel 2010]) 44
3.5 Vizz3D visualizing a C++ program (taken from [Panas et al. 2007]) 46
3.6 Night view of EvoSpaces (taken from [Dugerdil and Alam 2008]) 47
3.7 An execution trace of the web application JPetStore visualized in DyVis (taken from [Wulf 2010]) 48
3.8 Visualization of the Java Development Kit 6 using the Evo-Streets layout (taken from [Steinbrückner and Lewerentz 2010]) 49
3.9 Software map for Blender (taken from [Bohnet and Döllner 2011]) 50
3.10 Visualization of an execution of jEdit using 3D-HEB (taken from [Caserta et al. 2011]) 51
3.11 Deadlock of the dining philosophers problem visualized by SynchroVis (taken from [Waller et al. 2013]) 52
3.12 JUnit framework in SykscrapAR (taken from [Souza et al. 2012]) 53
3.13 Apache Commons Math in Manhattan (taken from [Lanza et al. 2013]) 54
3.14 Semantics of CodeMetropolis (taken from [Balogh and Beszédes 2013]) 55
3.15 SArF map of Weka 3.0 (taken from [Kobayashi et al. 2013]) . . 56
3.16 Execution view of Jinsight (taken from [De Pauw et al. 2001]) 58
3.17 JIVE (taken from [Reiss and Tarvo 2012]) 59

3.18 JOVE (taken from [Reiss and Tarvo 2012]) 60
3.19 A recorded execution trace of PMD visualized in Extravis . 61
3.20 Chromium visualized in ViewFusion (taken from [Trümper et al. 2012]) 63
3.21 Call Graph Analyzer (taken from [Bohnet and Döllner 2006]) 65
3.22 Schematic view of the visualization used in TraceCrawler (taken from [Greevy et al. 2006]) 66
3.23 Instance Collaboration View of the login feature in SmallWiki visualized in TraceCrawler (taken from [Greevy et al. 2006]) . 67
3.24 Web Services Navigator: trace selection window, Transaction Flows view, and Service Topology view (taken from [De Pauw et al. 2006]) 68
3.25 Streamsight (taken from [De Pauw et al. 2008]) 70
3.26 Root cause rating view of RanCorr (taken from [Marwede et al. 2009]) 71
3.27 VisuSniff (taken from [Oechsle et al. 2002]) 72

5.1 Overview of the fundamental activities in our ExplorViz approach 90

6.1 Filters involved in our application-level monitoring 97
6.2 Filters in our monitoring data adapter approach 98
6.3 Different RPC monitoring concepts (taken from [Matthiessen 2014]) 100
6.4 Basic idea of dynamic worker levels 103
6.5 Our scalable trace processing architecture 104
6.6 Filters in the analysis component (worker and master) 105
6.7 Initial state before scaling 106
6.8 Activities forming the upscaling process 108

7.1 Landscape meta-model 113

8.1 Time series of the overall activity in the landscape 121
8.2 One step of our interactive tutorial 122
8.3 Landscape-level perspective modeling the Kiel Data Management Infrastructure for ocean science 124

8.4 Event viewer for the software landscape 125
8.5 Conceptual architecture of the example software landscape in the modelling perspective (taken from [Simolka 2015]) . . . 126
8.6 Landscape-level perspective showing the differences in the conceptual architecture and actual architecture of the software landscape (taken from [Simolka 2015]) 127
8.7 Application-level perspective of a mockup of Neo4j 129
8.8 Visualizing the in- and outgoing communication in the application-level perspective 131
8.9 Overview of the application-level layout 132
8.10 Alternative layout using quadtrees and free space is cut off (taken from [Barbie 2014]) 133
8.11 Clustered *controller* component from Kieker 134
8.12 Dialog showing database queries conducted by JPetStore . . . 136
8.13 Replaying a monitored trace originating from Kieker 137
8.14 Analyzing the performance in Kieker 137
8.15 Dialog showing the code structure and *Version.java* of Neo4j . 138
8.16 Setup for our VR approach (doing a translation gesture) . . . 139
8.17 View on the city model of PMD through the Oculus Rift DK1 140
8.18 Gesture concepts for interacting with the city model (taken from [Krause 2015]) 142
8.19 Prusa i3 used for printing the physical models 147
8.20 Physical 3D-printed model of PMD 148
8.21 Two jigsaw pieces of our PMD model 149
8.22 One unpainted jigsaw piece of our physical PMD model . . . 150
8.23 Virtual model of ExplorViz in ExplorViz 153
8.24 Physical 3D-printed model of ExplorViz 154

9.1 Overall architecture of our ExplorViz implementation 161
9.2 Overview of the monitoring component 162
9.3 Overview of the analysis component 163
9.4 Overview of the repository component 165
9.5 Overview of the components involved in the visualization . . 167

10.1 Overview of the MooBench results in response time 173

10.2 Comparison of the resulting response times 177
10.3 Employed workload curve for the evaluation of the sclability and elasticity of our monitoring approach 180
10.4 JPetStore instance count and average CPU utilization of Master node 182
10.5 Analysis nodes and number of instances in the analysis level 184
10.6 Average monitored and analyzed method calls per second . . 185

11.1 The recorded execution trace of PMD for the first controlled experiment visualized in Extravis 193
11.2 The recorded execution trace of PMD for the first controlled experiment represented in ExplorViz v0.5 194
11.3 Overall time spent and correctness for both experiments . . . 206
11.4 Average time spent and correctness per task for PMD experiment 208
11.5 On-screenapplication-level perspective of ExplorViz v0.6 visualizing PMD 217
11.6 Physical 3D-printed and manually painted city metaphor model of PMD (334mm wide and 354mm deep) 218
11.7 Physical model of a mockup of Neo4J 223
11.8 Overall time spent and correctness for our physical model experiment 227
11.9 Average time spent and average correctness per task 229
11.10 An excerpt (29 of 140 applications) of the model of the technical IT infrastructure of the Kiel University in the flat landscape visualization 236
11.11 Excerpt of the model of the technical IT infrastructure of the Kiel University in our hierarchical landscape visualization . . 238
11.12 Overall time spent and correctness for our landscape experiment 247
11.13 Average time spent and average correctness per task 249

12.1 Cycle of the four activities in our control center 262
12.2 Mockup of the symptoms perspective visualizing PubFlow . 263

12.3 Mockup of application-level perspective of Neo4J in the diagnosis perspective . . . 265
12.4 Mockup of average response times (bars) and corresponding anomaly scores (upper part) for the `impl` component . . . 266
12.5 Mockup of planning perspective . . . 267
12.6 Mockup of reconfiguration options in planning perspective . . . 267
12.7 Time series showing a detected anomaly . . . 269
12.8 Diagnosis perspective on the application level . . . 270
12.9 Automatic capacity adjustment suggestion dialog . . . 271

List of Tables

10.1 Throughput of the ExplorViz' Monitoring Component (Traces per Second) . . . 172

10.2 Throughput for Kieker 1.8 Monitoring and Analysis (Traces per Second) . . . 176

10.3 Throughput for our ExplorViz Monitoring and Analysis Solution (Traces per Second) . . . 176

11.1 Description of the Program Comprehension Tasks for the First Application Perspective Experiment (PMD) . . . 197

11.2 Description of the program comprehension activity categories defined by and taken from [Pacione et al. 2004] . . . 198

11.3 Description of the Program Comprehension Tasks for our Application Perspective Replication (Babsi) . . . 201

11.4 Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) for PMD Experiment . . 204

11.5 Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) for Replication . . . 205

11.6 Debriefing Questionnaire Results for our PMD Experiment (1 is better – 5 is worse) . . . 214

11.7 Description of the Program Comprehension Tasks for the Physical Model Experiment . . . 221

11.8 Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) . . . 225

11.9 Debriefing Questionnaire Results for our Physical Model Experiment (1 is better – 5 is worse) . . . 233

11.10 Description of the System Comprehension Tasks for the Landscape Experiment . . . 241

11.11 Descriptive Statistics of the Results Related to Time Spent (in Minutes) and Correctness (in Points) . . . 245

11.12 Debriefing Questionnaire Results for our Landscape (1 is better – 5 is worse) . . . 254

12.1 Debriefing Questionnaire Results for our Extensibility Evaluation . . . 272

List of Listings

2.1 Manual Instrumentation Example in Java 30
2.2 Around Aspect for Instrumenting Java Source Code in AspectJ 32

Acronyms

AOP
: Aspect-Oriented Programming

APM
: Application Performance Management

CDO
: Cloud Deployment Option

DSL
: Domain-Specific Language

EA
: Enterprise Architecture

EM
: Enterprise Model

GQM
: Goal, Question, Metric

GWT
: Google Web Toolkit

HMD
: Head-Mounted Display

JDBC
: Java Database Connectivity

QoS
: Quality of Service

RPC
: Remote Procedure Call

RSF
: Rigi Standard Format

SLA
: Service Level Agreement

SQL
: Structured Query Language

VR
: Virtual Reality

WebGL
: Web Graphics Library

Bibliography

[Aher and Lambhate 2015] S. B. Aher and P. D. Lambhate. Real time monitoring of system counters using Map Reduce Framework for effective analysis. *International Journal of Current Engineering and Technology* 5.4 (2015). (Cited on page 278)

[Alam and Dugerdil 2007] S. Alam and P. Dugerdil. EvoSpaces: 3D visualization of software architecture. In: *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*. IEEE, 2007. (Cited on page 281)

[Anslow et al. 2006] C. Anslow, S. Marshall, and J. Noble. X3D-Earth in the software visualization pipeline. *X3D Earth Requirements* 6 (2006). (Cited on page 42)

[Avizienis et al. 2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004). (Cited on page 261)

[Balogh and Beszédes 2013] G. Balogh and A. Beszédes. CodeMetropolis – a Minecraft based collaboration tool for developers. In: *Proceedings of the First IEEE Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, 2013. (Cited on pages 54, 55)

[Balzer and Deussen 2004] M. Balzer and O. Deussen. Hierarchy based 3D visualization of large software structures. In: *Proceedings of the IEEE Conference on Visualization (VIS 2004)*. IEEE, 2004. (Cited on page 280)

[Balzer et al. 2004] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz. Software landscapes: visualizing the structure of large software systems. In: *Proceedings of the 6th Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym 2004)*. Eurographics Association, 2004. (Cited on page 299)

[Barbie 2014] A. Barbie. Stable 3D city layout in ExplorViz. Bachelor's thesis. Kiel University, Sept. 2014. (Cited on pages 19, 133, 134)

[Baresi and Guinea 2013] L. Baresi and S. Guinea. Event-based multi-level service monitoring. In: *Proceedings of 20th International Conference on Web Services (ICWS 2013)*. IEEE, June 2013. (Cited on page 278)

[Barham et al. 2003] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: online modelling and performance-aware systems. In: *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS 2003)*. USENIX Association, 2003. (Cited on page 277)

[Barzel 2014] M. Barzel. Evaluation von Clustering-Verfahren von Klassen für hierarchische Visualisierung in ExplorViz. (in German). Bachelor's thesis. Kiel University, Sept. 2014. (Cited on pages 20 and 135)

[Basili 2007] V. R. Basili. The role of controlled experiments in software engineering research. In: *Proceedings of the International Workshop on Empirical Software Engineering Issues: Critical Assessment and Future Directions*. Springer, 2007. (Cited on page 79)

[Basili and Weiss 1984] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). (Cited on pages 195, 219, and 239)

[Basili et al. 1999] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25.4 (1999). (Cited on page 79)

[Bennett et al. 2008] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice* 20.4 (2008). (Cited on page 57)

[Beye 2013] J. Beye. Technology evaluation for the communication between the monitoring and analysis component in Kieker. Bachelor's thesis. Kiel University, Sept. 2013. (Cited on pages 17, 18, and 163)

[Bielefeld 2012] T. C. Bielefeld. Online performance anomaly detection for large-scale software systems. Diploma thesis. Kiel University, Mar. 2012. (Cited on page 21)

[Binz et al. 2012] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm. Formalizing the cloud through enterprise topology graphs. In: *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD 2012)*. IEEE, 2012. (Cited on page 282)

[Binz et al. 2013] T. Binz, U. Breitenbucher, O. Kopp, and F. Leymann. Automated discovery and maintenance of enterprise topology graphs. In: *Proceedings of the IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA 2013)*. IEEE, Dec. 2013. (Cited on page 282)

[Bohnet and Döllner 2006] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In: *Proceedings of the International Symposium on Software Visualization (SoftVis 2006)*. ACM, 2006. (Cited on pages 64, 65, and 281)

[Bohnet and Döllner 2011] J. Bohnet and J. Döllner. Monitoring code quality and development activity by software maps. In: *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD 2011)*. ACM, 2011. (Cited on pages 49, 50)

[Brauer and Hasselbring 2013a] P. C. Brauer and W. Hasselbring. PubFlow: a scientific data publication framework for marine science. In: *Proceedings of the International Conference on Marine Data and Information Systems (IMDIS 2013)*. ENEA, Sept. 2013. (Cited on page 123)

[Brauer and Hasselbring 2013b] P. C. Brauer and W. Hasselbring. PubFlow: provenance-aware workflows for research data publication. In: *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2013)*. USENIX Association, Apr. 2013. (Cited on page 261)

[Brauer et al. 2014] P. C. Brauer, F. Fittkau, and W. Hasselbring. The aspect-oriented architecture of the CAPS framework for capturing, analyzing and archiving provenance data. In: *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW 2014)*. Springer, June 2014. (Cited on page 17)

[Breu et al. 2011] R. Breu, B. Agreiter, M. Farwick, M. Felderer, M. Hafner, and F. Innerhofer-Oberperfler. Living models – ten principles for change-driven software engineering. *International Journal of Software and Informatics* 5.1-2 (2011). (Cited on page 282)

[Briand et al. 2004] L. Briand, Y. Labiche, and J. Leduc. Towards the reverse engineering of UML sequence diagrams for distributed, multithreaded Java software. Technical report SCE-04-04. Carleton University, Sept. 2004. (Cited on pages 63, 72, and 280)

[Briand et al. 2005] L. C. Briand, Y. Labiche, and J. Leduc. Tracing distributed systems executions using AspectJ. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*. IEEE, 2005. (Cited on page 72)

[Briand et al. 2006] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transaction on Software Engineering* 32.9 (Sept. 2006). (Cited on pages 72 and 279)

[Brunst and Nagel 2003] H. Brunst and W. E. Nagel. Scalable performance analysis of parallel systems: concepts and experiences. In: *Proceedings of the 10th Conference on Parallel Computing: Software Technology, Algorithms, Architectures, and Applications (ParCo 2003)*. Elsevier, 2003. (Cited on page 278)

[Bulej 2007] L. Bulej. Connector-based performance data collection for component applications. PhD thesis. Charles University in Prague, 2007. (Cited on page 30)

[Card et al. 1999] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. Readings in information visualization: using vision to think. Morgan Kaufmann, 1999. (Cited on pages 39, 40)

[Caserta et al. 2011] P. Caserta, O. Zendra, and D. Bodenes. 3D hierarchical edge bundles to visualize relations in a software city metaphor. In: *Proceedings of the 6th IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011)*. IEEE, 2011. (Cited on pages 51 and 281)

[Cornelissen et al. 2011] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37.3 (May 2011). (Cited on page 284)

[Cornelissen et al. 2007] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In: *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007)*. IEEE, 2007. (Cited on pages 7, 60, and 280)

[Cornelissen et al. 2008] B. Cornelissen, L. Moonen, and A. Zaidman. An assessment methodology for trace reduction techniques. In: *Proceedings of the 24th International Conference on Software Maintenance (ICSM 2008)*. IEEE, 2008. (Cited on page 89)

[Cornelissen et al. 2009] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey. Trace visualization for program comprehension: a controlled experiment. In: *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC 2009)*. IEEE, May 2009. (Cited on pages 60, 191, 192, 196, 198, 211, 213, 216, 220, 239, 240, and 284)

[Cortellessa et al. 2011] V. Cortellessa, A. Di Marco, and P. Inverardi. Model-based software performance analysis. Springer, 2011. (Cited on page 266)

[Crick et al. 2014] T. Crick, B. A. Hall, and S. Ishtiaq. Can I implement your algorithm?: a model for reproducible research software. In: *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE 2014)*. arXiv, Nov. 2014. (Cited on page 191)

[De Pauw et al. 2005] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar. Web Services Navigator: visualizing the execution of web services. *IBM Systems Journal* 44.4 (2005). (Cited on page 279)

[De Pauw et al. 2001] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. In: *Proceedings of the ICSE Workshop on Software Visualization*. ACM, 2001. (Cited on pages 57, 58)

[De Pauw et al. 2002] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of Java programs. In: *Proceedings of the International Seminar on Software Visualization*. Springer, 2002. (Cited on page 57)

[De Pauw et al. 2006] W. De Pauw, S. Krasikov, and J. F. Morar. Execution patterns for visualizing web services. In: *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis 2006)*. ACM, 2006. (Cited on pages 66, 68, 69)

[De Pauw et al. 2008] W. De Pauw, H. Andrade, and L. Amini. Streamsight: a visualization tool for large-scale streaming applications. In: *Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis 2008)*. ACM, 2008. (Cited on pages 69, 70, and 279)

[Dean and Ghemawat 2010] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM* 53.1 (Jan. 2010). (Cited on pages 8, 102, and 278)

[Delimarschi et al. 2014] D. Delimarschi, G. Swartzendruber, and H. Kagdi. Enabling integrated development environments with natural user interface interactions. In: *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. ACM, 2014. (Cited on pages 140 and 282)

[Di Lucca and Di Penta 2006] G. A. Di Lucca and M. Di Penta. Experimental settings in program comprehension: challenges and open issues. In: *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*. IEEE, 2006. (Cited on pages 79, 192, 216, and 239)

[Di Penta et al. 2007] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In: *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007)*. IEEE, June 2007. (Cited on pages 79, 192, 216, and 239)

[Döhring 2012] P. Döhring. Visualisierung von Synchronisationspunkten in Kombination mit der Statik und Dynamik eines Softwaresystems. (in German). Master's thesis. Kiel University, 2012. (Cited on pages 51 and 281)

[Ducasse et al. 2005] S. Ducasse, L. Renggli, and R. Wuyts. SmallWiki: a meta-described collaborative content management system. In: *Proceedings of the 2005 International Symposium on Wikis (WikiSym 2005)*. ACM, 2005. (Cited on page 65)

[Dugerdil and Alam 2008] P. Dugerdil and S. Alam. Execution trace visualization in a 3D space. In: *Proceedings of the 5th International Conference on Information Technology: New Generations (ITNG 2008)*. Apr. 2008. (Cited on pages 3, 45, and 47)

[Dwyer et al. 2007] M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE, 2007. (Cited on page 34)

[Ehlers 2011] J. Ehlers. Self-adaptive performance monitoring for component-based software systems. PhD thesis. Kiel University, Department of Computer Science, 2011. (Cited on page 64)

[Ehlers et al. 2011] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In: *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC 2011)*. ACM, 2011. (Cited on pages 89 and 263)

[Eichelberger and Schmid 2014] H. Eichelberger and K. Schmid. Flexible resource monitoring of Java programs. *Journal of Systems and Software* 93 (2014). (Cited on pages 9, 173, 186, 277, and 291)

[Elliott et al. 2015] A. Elliott, B. Peiris, and C. Parnin. Virtual reality in software engineering: affordances, applications, and challenges. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. IEEE, May 2015. (Cited on page 140)

[Finke 2014] S. Finke. Automatische Anleitung einer Versuchsperson während eines kontrollierten Experiments in ExplorViz. (in German). Master's thesis. Kiel University, Sept. 2014. (Cited on pages 20 and 122)

[Finkel and Bentley 1974] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4.1 (1974). (Cited on pages 19 and 133)

[Fischer et al. 2007] R. Fischer, S. Aier, and R. Winter. A federated approach to enterprise architecture model maintenance. In: *Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2007)*. GI, 2007. (Cited on page 282)

[Fittkau 2013] F. Fittkau. Live trace visualization for system and program comprehension in large software landscapes. Technical report 1310. Department of Computer Science, Kiel University, Germany, Nov. 2013. (Cited on pages 2, 11, 78, and 88)

[Fittkau and Hasselbring 2015a] F. Fittkau and W. Hasselbring. Data for: Elastic application-level monitoring for large software landscapes in the cloud. DOI: 10.5281/zenodo.19296. July 2015. (Cited on page 181)

[Fittkau and Hasselbring 2015b] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*. Springer, Sept. 2015. (Cited on pages 14, 88, 96, 170, and 276)

[Fittkau et al. 2012a] F. Fittkau, S. Frey, and W. Hasselbring. CDOSim: simulating cloud deployment options for software migration support. In: *Proceedings of the 6th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2012)*. IEEE, Sept. 2012. (Cited on page 15)

[Fittkau et al. 2012b] F. Fittkau, S. Frey, and W. Hasselbring. Cloud user-centric enhancements of the simulator CloudSim to improve cloud deployment option analysis. In: *Proceedings of the 1st European Conference on Service-Oriented and Cloud Computing (ESOCC 2012)*. Springer, Sept. 2012. (Cited on page 15)

[Fittkau et al. 2013a] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Data for: Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. DOI: 10.5281/zenodo.7622. Nov. 2013. (Cited on page 174)

[Fittkau et al. 2013b] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the ExplorViz approach. In: *Proceedings of the 1st IEEE International Working*

Conference on Software Visualization (VISSOFT 2013). IEEE, Sept. 2013. (Cited on pages 2, 11, 88, 118, 119, and 276)

[Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDays 2013)*. Volume 1083. CEUR Workshop Proceedings, Nov. 2013. (Cited on pages 11, 96, 160, 170, and 276)

[Fittkau et al. 2014a] F. Fittkau, P. Stelzer, and W. Hasselbring. Live visualization of large software landscapes for ensuring architecture conformance. In: *Proceedings of the 2nd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2014)*. ACM, Aug. 2014. (Cited on pages 12, 22, 120, and 276)

[Fittkau et al. 2014b] F. Fittkau, A. van Hoorn, and W. Hasselbring. Towards a dependability control center for large software landscapes. In: *Proceedings of the 10th European Dependable Computing Conference (EDCC 2014)*. IEEE, May 2014. (Cited on pages 13, 119, 259, 260)

[Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. IEEE, May 2015. (Cited on pages 14, 38, 119, 190, and 276)

[Fittkau et al. 2015b] F. Fittkau, A. Krause, and W. Hasselbring. Data for: Exploring software cities in virtual reality. DOI: 10.5281/zenodo.23168. Sept. 2015. (Cited on page 143)

[Fittkau et al. 2015c] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Experimental data for: Comparing trace visualizations for program comprehension through controlled experiments. DOI: 10.5281/zenodo.11611. May 2015. (Cited on pages 192 and 205)

[Fittkau et al. 2015d] F. Fittkau, A. Krause, and W. Hasselbring. Experimental data for: Hierarchical software landscape visualization for system comprehension: a controlled experiment. DOI: 10.5281/zenodo.18853. June 2015. (Cited on pages 235 and 246)

[Fittkau et al. 2015e] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Experimental data for: Research perspective on supporting software engineering via physical 3D models. DOI: 10.5281/zenodo.18378. June 2015. (Cited on pages 215 and 227)

[Fittkau et al. 2015f] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015. (Cited on pages 12, 118, 276, and 296)

[Fittkau et al. 2015g] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. AIS, May 2015. (Cited on pages 12, 88, 112, 118, 119, 160, 243, and 276)

[Fittkau et al. 2015h] F. Fittkau, A. Krause, and W. Hasselbring. Hierarchical software landscape visualization for system comprehension: a controlled experiment. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015. (Cited on pages 15, 118, 119, and 190)

[Fittkau et al. 2015i] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. In: *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*. IEEE, Sept. 2015. (Cited on pages 13, 118, and 296)

[Fittkau et al. 2015j] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research perspective on supporting software engineering via physical 3D models. Technical report 1507. Department of Computer Science, Kiel University, Germany, June 2015. (Cited on pages 14, 118, 190, and 296)

[Fonseca et al. 2007] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI 2007)*. USENIX Association, 2007. (Cited on page 277)

[Frank 2011] U. Frank. The MEMO Meta Modelling Language (MML) and language architecture (2nd Edition). Technical report 24. ICB-Research Report, Feb. 2011. (Cited on page 282)

[Frey et al. 2013] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE, May 2013. (Cited on pages 16, 17)

[Frey et al. 2015] S. Frey, F. Fittkau, and W. Hasselbring. Optimizing the deployment of software in the cloud. In: *Proceedings of the Conference on Software Engineering & Management 2015*. Köllen Druck+Verlag, Mar. 2015. (Cited on page 17)

[Frotscher 2013] T. Frotscher. Architecture-based multivariate anomaly detection for software systems. Master's thesis. Kiel University, Oct. 2013. (Cited on page 269)

[Gill 2015] J. Gill. Integration von Kapazitätsmanagement in ein Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015. (Cited on pages 20 and 270)

[Goldin-Meadow 2005] S. Goldin-Meadow. Hearing gesture: how our hands help us think. Harvard University Press, 2005. (Cited on pages 145 and 215)

[Goltz et al. 2015] U. Goltz, R. H. Reussner, M. Goedicke, W. Hasselbring, L. Märtin, and B. Vogel-Heuser. Design for future: managed software evolution. *Computer Science - R&D* 30.3-4 (2015). (Cited on page 3)

[Graham et al. 2004] H. Graham, H. Y. Yang, and R. Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. In: *Proceedings of the Australasian Symposium on Information Visualisation (APVIS 2004)*. Australian Computer Society, 2004. (Cited on page 299)

[Greevy et al. 2006] O. Greevy, M. Lanza, and C. Wysseier. Visualizing live software systems in 3D. In: *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis 2006)*. ACM, 2006. (Cited on pages 3, 46, 64, 66, 67, and 281)

[Hamou-Lhadj 2007] A. Hamou-Lhadj. Effective exploration and visualization of large execution traces. In: *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*. IEEE, 2007. (Cited on page 3)

[Hamou-Lhadj and Lethbridge 2004] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In: *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2004)*. IBM Press, 2004. (Cited on page 57)

[Hauder et al. 2012] M. Hauder, F. Matthes, and S. Roth. Challenges for automated enterprise architecture documentation. In: *Proceedings of the Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation (TEAR 2012)*. Springer, 2012. (Cited on page 282)

[Herndon et al. 1994] K. P. Herndon, A. van Dam, and M. Gleicher. The challenges of 3D interaction: A CHI '94 Workshop. *SIGCHI Bull.* 26.4 (Oct. 1994). (Cited on page 139)

[Hilbrich and Muller-Pfefferkorn 2012] M. Hilbrich and R. Muller-Pfefferkorn. Identifying limits of scalability in distributed, heterogeneous, layer based monitoring concepts like SLAte. *Computer Science* 13.3 (2012). (Cited on page 278)

[Holten 2006] D. Holten. Hierarchical edge bundles: visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006). (Cited on page 51)

[ISO/IEC/IEEE 24765 2010] ISO/IEC/IEEE 24765. Systems and software engineering – Vocabulary. 2010. (Cited on page 30)

[Jähde 2015] D. Jähde. Performance Analyse Tool in ExplorViz. Seminar Software Performance Engineering Winter Term 14/15, (in German). 2015. (Cited on page 138)

[Jain et al. 1991] R. Jain, D. Menasce, L. W. Dowdy, V. A. Almeida, C. U. Smith, and L. G. Williams. The Art of Computer Systems Performance Analysis: Techniques. John Wiley & Sons, 1991. (Cited on page 33)

[Jedlitschka and Pfahl 2005] A. Jedlitschka and D. Pfahl. Reporting guidelines for controlled experiments in software engineering. In: *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2005)*. IEEE, 2005. (Cited on pages 79, 192, 216, and 239)

[Juristo and Moreno 2010] N. Juristo and A. M. Moreno. Basics of software engineering experimentation. Springer, 2010. (Cited on pages 79, 211, 232, and 253)

[Kahn 2006] K. Kahn. Time travelling animated program executions. In: *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis 2006)*. ACM, 2006. (Cited on page 135)

[Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Springer, 1997. (Cited on page 31)

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*. Springer, 2001. (Cited on page 31)

[Kitchenham et al. 2002] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28.8 (2002). (Cited on pages 79, 192, 216, and 239)

[Kleiberg et al. 2001] E. Kleiberg, H. Van De Wetering, and J. J. Van Wijk. Botanical visualization of huge hierarchies. In: *Proceedings of the IEEE Symposium on Information Visualization (IV 2001)*. IEEE, 2001. (Cited on page 299)

[Knight and Munro 1999] C. Knight and M. Munro. Comprehension with[in] virtual environment visualisations. In: *Proceedings of the Seventh International Workshop on Program Comprehension (IWPC 1999)*. IEEE, 1999. (Cited on pages 9, 41, 42)

[Knight and Munro 2000] C. Knight and M. Munro. Virtual but visible software. In: *Proceedings of the IEEE International Conference on Information Visualization (IV 2000)*. IEEE, 2000. (Cited on pages 41 and 128)

[Knoche et al. 2012] H. Knoche, A. van Hoorn, W. Goerigk, and W. Hasselbring. Automated source-level instrumentation for dynamic dependency analysis of COBOL systems. In: *Proceedings of the 14th Workshop on Software-Reengineering (WSR 2012)*. GI, May 2012. (Cited on page 98)

[Kobayashi et al. 2012] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo. Feature-gathering dependency-based software clustering using dedication and modularity. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2012)*. IEEE, 2012. (Cited on page 56)

[Kobayashi et al. 2013] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, and A. Matsuo. SArF map: visualizing software architecture from feature and layer viewpoints. In: *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC 2013)*. IEEE, 2013. (Cited on page 56)

[Koppenhagen 2013] E. Koppenhagen. Evaluation von Elastizitätsstrategien in der Cloud im Hinblick auf optimale Ressourcennutzung. (in German). Bachelor's thesis. Kiel University, Sept. 2013. (Cited on pages 18 and 297)

[Kosche 2013] M. Kosche. Tracking user actions for the web-based front end of ExplorViz. Bachelor's thesis. Kiel University, Sept. 2013. (Cited on page 18)

[Koschke 2003] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice* 15.2 (2003). (Cited on page 139)

[Koschke and Quante 2005] R. Koschke and J. Quante. On dynamic feature location. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*. ACM, 2005. (Cited on page 3)

[Krause 2015] A. Krause. Erkundung von Softwarestädten mithilfe der virtuellen Realität. (in German, in progress). Bachelor's thesis. Kiel University, Sept. 2015. (Cited on pages 21, 22, 84, 140–143, and 257)

[Kruskal and Landwehr 1983] J. B. Kruskal and J. M. Landwehr. Icicle plots: better displays for hierarchical clustering. *The American Statistician* 37.2 (1983). (Cited on page 62)

[Lange and Chaudron 2007] C. Lange and M. R. V. Chaudron. Interactive views to improve the comprehension of UML models – an experimental validation. In: *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007)*. IEEE, June 2007. (Cited on page 284)

[Lanza 2003] M. Lanza. CodeCrawler – lessons learned in building a software visualization tool. In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*. IEEE, Mar. 2003. (Cited on page 64)

[Lanza et al. 2013] M. Lanza, M. D'Ambros, A. Bacchelli, L. Hattori, and F. Rigotti. Manhattan: supporting real-time visual team activity awareness. In: *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013)*. IEEE, 2013. (Cited on pages 53, 54)

[Levene 1960] H. Levene. Robust tests for equality of variances. *Contributions to probability and statistics: Essays in honor of Harold Hotelling* 2 (1960), pages 278–292. (Cited on pages 205, 227, and 246)

[Lewerentz and Noack 2004] C. Lewerentz and A. Noack. "'CrocoCosmos – 3D Visualization of Large Object-Oriented Programs'". In: *Graph Drawing Software*. Springer, 2004. (Cited on pages 3 and 48)

[Likert 1932] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology* 22.140 (1932). (Cited on pages 199, 222, and 242)

[Limberger et al. 2013] D. Limberger, B. Wasty, J. Trümper, and J. Döllner. Interactive software maps for web-based source code analysis. In: *Proceedings of the 18th International Conference on 3D Web Technology (Web3D 2013)*. ACM, 2013. (Cited on page 50)

[Lindsay and Ehrenberg 1993] R. M. Lindsay and A. S. Ehrenberg. The design of replicated studies. *The American Statistician* 47.3 (1993). (Cited on page 191)

[Mahmens 2014] S. Mahmens. Architektur Rekonstrukion mit Kieker durch AOP-basierte Instrumentierung einer C++-Anwendung. (in German). Master's thesis. Kiel University, Apr. 2014. (Cited on page 98)

[Maletic et al. 2001] J. I. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object-oriented software in virtual reality. In: *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*. IEEE, 2001. (Cited on page 281)

[Malnati et al. 2008] G. Malnati, C. M. Cuva, and C. Barberis. JThreadSpy: a tool for improving the effectiveness of concurrent system teaching and learning. In: *Proceedings of the 2008 International Conference on Computer Science and Software Engineering (CSSE 2008)*. IEEE, 2008. (Cited on page 64)

[Mannstedt 2015] K. C. Mannstedt. Integration von Anomalienerkennung in einem Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015. (Cited on pages 20, 21, and 270)

[Marcus et al. 2005] A. Marcus, D. Comorski, and A. Sergeyev. Supporting the evolution of a software visualization tool through usability studies. In: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*. IEEE, May 2005. (Cited on page 283)

[Marwede et al. 2009] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*. IEEE, 2009. (Cited on pages 69, 71, 270, 279, and 298)

[Matthes et al. 2008] F. Matthes, S. Buckl, J. Leitel, and C. Schweda. Enterprise architecture management tool survey 2008. Technical report. Technische Universität München, 2008. (Cited on page 282)

[Matthiessen 2014] N. Matthiessen. Monitoring remote procedure calls – concepts and evaluation. Bachelor's thesis. Kiel University, Mar. 2014. (Cited on pages 18, 19, 99, 100)

[Meng et al. 2010] S. Meng, L. Liu, and V. Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In: *Proceedings of the 11th International Middleware Conference (Middleware 2010)*. ACM, 2010. (Cited on page 278)

[Michaelis 2015] J. Michaelis. Integration von Ursachenerkennung in ein Kontrollzentrum für Softwarelandschaften. (in German). Bachelor's thesis. Kiel University, Mar. 2015. (Cited on pages 21 and 270)

[Miller 1956] G. A. Miller. The magical number 7, plus or minus 2: some limits on our capacity for processing information. *The Psychological Review* 63 (1956). (Cited on page 6)

[Moonen 2003] L. Moonen. Exploring software systems. In: *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM 2003)*. IEEE, Sept. 2003. (Cited on page 3)

[Oechsle and Schmitt 2002] R. Oechsle and T. Schmitt. JAVAVIS: automatic program visualization with object and sequence diagrams using the java debug interface (JDI). In: *Proceedings of the International Seminar on Software Visualization*. Springer, 2002. (Cited on page 63)

[Oechsle et al. 2002] R. Oechsle, O. Gronz, and M. Schüler. VisuSniff: a tool for the visualization of network traffic. In: *Proceedings of the Second Program Visualization Workshop*. ACM, 2002. (Cited on pages 72 and 280)

[Oppezzo and Schwartz 2014] M. Oppezzo and D. L. Schwartz. Give your ideas some legs: the positive effect of walking on creative thinking. *Journal of experimental psychology. Learning, memory, and cognition* 40.1 (2014). (Cited on page 140)

[Pacione et al. 2004] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In: *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*. IEEE, Nov. 2004. (Cited on pages 196, 198, 213, 220, 222, 234, and 255)

[Panas et al. 2007] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating software architecture using a unified single-view visualization. In: *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE, 2007. (Cited on pages 45, 46)

[Panas et al. 2003] T. Panas, R. Berrigan, and J. Grundy. A 3D metaphor for software production visualization. In: *Proceedings of the 7th International Conference on Information Visualization (IV 2003)*. IEEE, 2003. (Cited on pages 3, 42, 43)

[Parsons 2007] T. Parsons. Automatic detection of performance design and deployment antipatterns in component based enterprise systems. PhD thesis. University College Dublin, 2007. (Cited on page 30)

[Pearson and Hartley 1972] E. Pearson and H. Hartley. Biometrika tables for statisticians. 2nd edition. Cambridge University Press, 1972. (Cited on pages 205, 227, and 246)

[Penny 1993] D. A. Penny. The software landscape: a visual formalism for programming-in-the-large. PhD thesis. University of Toronto, 1993. (Cited on page 3)

[Price et al. 1993] B. Price, R. Baecker, and I. Small. A principled taxonomy of sofware visualization. *Journal of Visual Languages and Computing* 4.3 (1993). (Cited on page 37)

[Quante 2008] J. Quante. Do dynamic object process graphs support program understanding? – a controlled experiment. In: *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC 2008)*. IEEE, June 2008. (Cited on page 283)

[Rajlich and Cowan 1997] V. Rajlich and G. S. Cowan. Towards standard for experiments in program comprehension. In: *Proceedings of the 5th International Workshop on Program Comprehension (IWPC 1997)*. IEEE, 1997. (Cited on pages 191, 196, 220, and 240)

[Razali and Wah 2011] N. Razali and Y. B. Wah. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. *Journal of Statistical Modeling and Analytics* 2.1 (2011), pages 21–33. (Cited on pages 205, 227, and 246)

[Reiss 2003] S. P. Reiss. JIVE: visualizing Java in action demonstration description. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. IEEE, May 2003. (Cited on page 59)

[Reiss and Renieris 2005] S. P. Reiss and M. Renieris. JOVE: Java as it happens. In: *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis 2005)*. ACM, 2005. (Cited on page 59)

[Reiss and Tarvo 2012] S. P. Reiss and A. Tarvo. What is my program doing? Program dynamics in programmer's terms. In: *Proceedings of the 5th International Conference on Runtime Verification (RV 2012)*. Springer, 2012. (Cited on pages 59, 60, and 280)

[Rohr et al. 2010] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In: *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW 2010)*. ACM, 2010. (Cited on page 179)

[Roth and Matthes 2014] S. Roth and F. Matthes. Visualizing differences of enterprise architecture models. In: *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM) at Software Engineering (SE)*. 2014. (Cited on page 282)

[Roth et al. 2014] S. Roth, M. Zec, and F. Matthes. Enterprise architecture visualization tool survey 2014. Technical report. Technische Universität München, 2014. (Cited on page 282)

[Salehie and Tahvildari 2009] M. Salehie and L. Tahvildari. Self-adaptive software: landscape and research challenges. *ACM Transactions on Autonomous Adaptive Systems* 4.2 (2009). (Cited on page 261)

[Schulze et al. 2014] C. D. Schulze, M. Spönemann, and R. von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014). (Cited on page 125)

[Sensalire et al. 2009] M. Sensalire, P. Ogao, and A Telea. Evaluation of software visualization tools: lessons learned. In: *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*. IEEE, Sept. 2009. (Cited on pages 79, 192, 216, and 239)

[Shadish et al. 2002] W. R. Shadish, T. D. Cook, and D. T. Campbell. Experimental and quasi-experimental designs for generalized causal inference. Wadsworth – Cengage Learning, 2002. (Cited on pages 211, 232, and 253)

[Shapiro and Wilk 1965] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika* 52.3/4 (1965). (Cited on pages 205, 227, and 246)

[Sharif et al. 2013] B. Sharif, G. Jetty, J. Aponte, and E. Parra. An empirical study assessing the effect of SeeIT 3D on comprehension. In: *Proceedings of the 1st IEEE Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013. (Cited on page 283)

[Shneiderman 1980] B. Shneiderman. Software psychology: human factors in computer and information systems. *Winthrop Publishers, Inc.* (1980). (Cited on page 62)

[Shneiderman 1996] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In: *Proceedings of the 1996 IEEE Symposium on Visual Languages (VL 1996)*. IEEE, 1996. (Cited on page 6)

[Shull et al. 2002] F. Shull, V. Basili, J. Carver, J. C. Maldonado, G. H. Travassos, M. Mendonca, and S. Fabbri. Replicating software engineering experiments: addressing the tacit knowledge problem. In: *Proceedings of the International Empirical Software Engineering Symposium (ESEM 2002)*. ACM, 2002. (Cited on page 80)

[Sigelman et al. 2010] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report dapper-2010-1. Google, 2010. (Cited on page 277)

[Simolka 2015] T. Simolka. Live architecture conformance checking in ExplorViz. (in German, in progress). Bachelor's thesis. Kiel University, Sept. 2015. (Cited on pages 22, 120, 126–128)

[Souza et al. 2012] R. Souza, B. Silva, T. Mendes, and M. Mendonca. SkyscrapAR: an augmented reality visualization for software evolution. In: *Proceedings of the 2nd Brazilian Workshop on Software Visualization (WBVS 2012)*. SBC, 2012. (Cited on pages 52, 53, and 282)

[Steinbrückner 2010] F. Steinbrückner. Coherent software cities. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2010)*. IEEE, Sept. 2010. (Cited on page 133)

[Steinbrückner and Lewerentz 2010] F. Steinbrückner and C. Lewerentz. Representing development history in software cities. In: *Proceedings of the 5th International Symposium on Software visualization (SoftVis 2010)*. ACM, 2010. (Cited on pages 48, 49)

[Stelzer 2014] P. Stelzer. Scalable and live trace processing in the cloud. Bachelor's thesis. Kiel University, Mar. 2014. (Cited on pages 19 and 179)

[Stevanetic et al. 2015] S. Stevanetic, M. A. Javed, and U. Zdun. The impact of hierarchies on the architecture-level software understandability – a controlled experiment. In: *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC 2015)*. IEEE, Sept. 2015. (Cited on page 284)

[Storey et al. 1997] M.-A. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In: *Proceedings of the 4h Working Conference on Reverse Engineering (WCRE 1997)*. IEEE, 1997. (Cited on page 284)

[Systä 2000] T. Systä. Static and dynamic reverse engineering techniques for Java software systems. PhD thesis. Acta Electronica Universitatis Tamperensis, Finland, 2000. (Cited on page 63)

[Teyseyre and Campo 2009] A. Teyseyre and M. Campo. An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics* 15.1 (Jan. 2009). (Cited on page 139)

[Trümper et al. 2010] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In: *Proceedings of the 5th International Symposium on Software Visualization (SoftVis 2010)*. ACM, 2010. (Cited on pages 256 and 300)

[Trümper et al. 2012] J. Trümper, A. Telea, and J. Döllner. ViewFusion: correlating structure and activity views for execution traces. In: *Proceedings of the 10th Theory and Practice of Computer Graphics Conference (TP.CG.2012)*. EG, 2012. (Cited on pages 62, 63, and 280)

[Van Hoorn et al. 2009a] A. van Hoorn, M. Rohr, I. A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In: *Proceedings of the Warm Up Workshop (WUP 2009) for ICSE 2010*. ACM, Apr. 2009. (Cited on page 278)

[Van Hoorn et al. 2009b] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: design and application of the Kieker framework. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009. (Cited on page 33)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012. (Cited on pages 9, 17, 98, 101, 163, and 277)

[Van Solingen and Berghout 1999] R. Van Solingen and E. Berghout. The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill, 1999. (Cited on page 80)

[Van Solingen et al. 2002] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (GQM) approach. *Encyclopedia of Software Engineering* (2002). (Cited on page 80)

[Vierhauser et al. 2014] M. Vierhauser, R. Rabiser, P. Grunbacher, C. Danner, S. Wallner, and H. Zeisel. A flexible framework for runtime monitoring of System-of-Systems architectures. In: *Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture (WICSA 2014)*. IEEE, Apr. 2014. (Cited on page 278)

[Vierhauser et al. 2013] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, and S. Wallner. Evolving systems of systems: industrial challenges and research perspectives. In: *Proceedings of the 1st International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013)*. ACM, 2013. (Cited on pages 3, 4)

[Voets 2008] R. Voets. JRET: a tool for the reconstruction of sequence diagrams from program executions. Master's thesis. TU Delft, Electrical Engineering, Mathematics, Computer Science, Information, and Communication Technology (ICT), Sept. 2008. (Cited on page 64)

[Waller 2014] J. Waller. Performance Benchmarking of Application Monitoring Frameworks. Kiel Computer Science Series 2014/5. Department of Computer Science, Kiel University, Dec. 2014. (Cited on pages 9, 33, 151, 173, and 186)

[Waller et al. 2013] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring. SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE, Sept. 2013. (Cited on pages 16, 51, 52, and 281)

[Waller et al. 2014a] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance 2014 (SOSP 2014)*. University of Stuttgart, Nov. 2014. (Cited on pages 13, 170–173, and 175)

[Waller et al. 2014b] J. Waller, F. Fittkau, and W. Hasselbring. Data for: Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability. DOI: 10.5281/zenodo.11428. Nov. 2014. (Cited on page 172)

[Wang and Miyamoto 1996] X. Wang and I. Miyamoto. Generating customized layouts. In: *Proceedings of the International Symposium on Graph Drawing (GD 1996)*. Springer, 1996. (Cited on page 299)

[Ware 2013] C. Ware. Information visualization – perception for design. Morgan Kaufmann, 2013. (Cited on page 6)

[Ware and Mitchell 2005] C. Ware and P. Mitchell. Reevaluating stereo and motion cues for visualizing graphs in three dimensions. In: *Proceedings of 2nd Symposium on Applied Perception in Graphics and Visualization (APGV 2005)*. ACM, 2005. (Cited on page 140)

[Ware et al. 1993] C. Ware, K. Arthur, and K. S. Booth. Fish tank virtual reality. In: *Proceedings of the INTERACT 1993 and Conference on Human Factors in Computing Systems (CHI 1993)*. ACM, 1993. (Cited on page 140)

[Wechselberg 2013] N. B. Wechselberg. Monitoring von Perl-basierten Webanwendungen mittels Kieker. (in German). Bachelor's thesis. Kiel University, Apr. 2013. (Cited on page 98)

[Weißenfels 2014] B. Weißenfels. Evaluation of trace reduction techniques for online trace visualization. Master's thesis. Kiel University, May 2014. (Cited on pages 19 and 164)

[Wettel 2010] R. Wettel. Software systems as cities. PhD thesis. University of Lugano, 2010. (Cited on pages 44, 130, and 134)

[Wettel and Lanza 2007] R. Wettel and M. Lanza. Visualizing software systems as cities. In: *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*. IEEE, June 2007. (Cited on pages 3, 43, 44, and 128)

[Wettel et al. 2011] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 2011. (Cited on pages 45, 192, 196, 216, 220, 239, 240, and 283)

[Wohlin et al. 2012] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. Experimentation in software engineering. Springer, 2012. (Cited on pages 79, 211, 232, 246, and 253)

[Wulf 2010] C. Wulf. Runtime visualization of static and dynamic architectural views of a software system to identify performance problems. Bachelor's thesis. Kiel University, 2010. (Cited on pages 46, 48, and 281)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework. In: *Symposium on Software Performance 2014: Joint Descartes/Kieker/Palladio Days (SOSP 2014)*. University of Stuttgart, Nov. 2014. (Cited on page 177)

[Wysseier 2005] C. Wysseier. Interactive 3-D visualization of feature-traces. Master's thesis. University of Bern, Switzerland, 2005. (Cited on pages 46 and 64)

[Yang et al. 2006] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. ACM, 2006. (Cited on page 34)

[Young and Munro 1998] P. Young and M. Munro. Visualising software in virtual reality. In: *Proceedings of the 6th International Workshop on Program Comprehension (IWPC 1998)*. IEEE, 1998. (Cited on page 282)

[Zirkelbach 2015] C. Zirkelbach. Performance monitoring of database operations. Master's thesis. Kiel University, July 2015. (Cited on pages 21 and 135)

[Zirkelbach et al. 2015] C. Zirkelbach, W. Hasselbring, F. Fittkau, and L. Carr. Performance analysis of legacy Perl software via batch and interactive trace visualization. Technical report 1509. Department of Computer Science, Kiel University, Germany, Aug. 2015. (Cited on page 17)