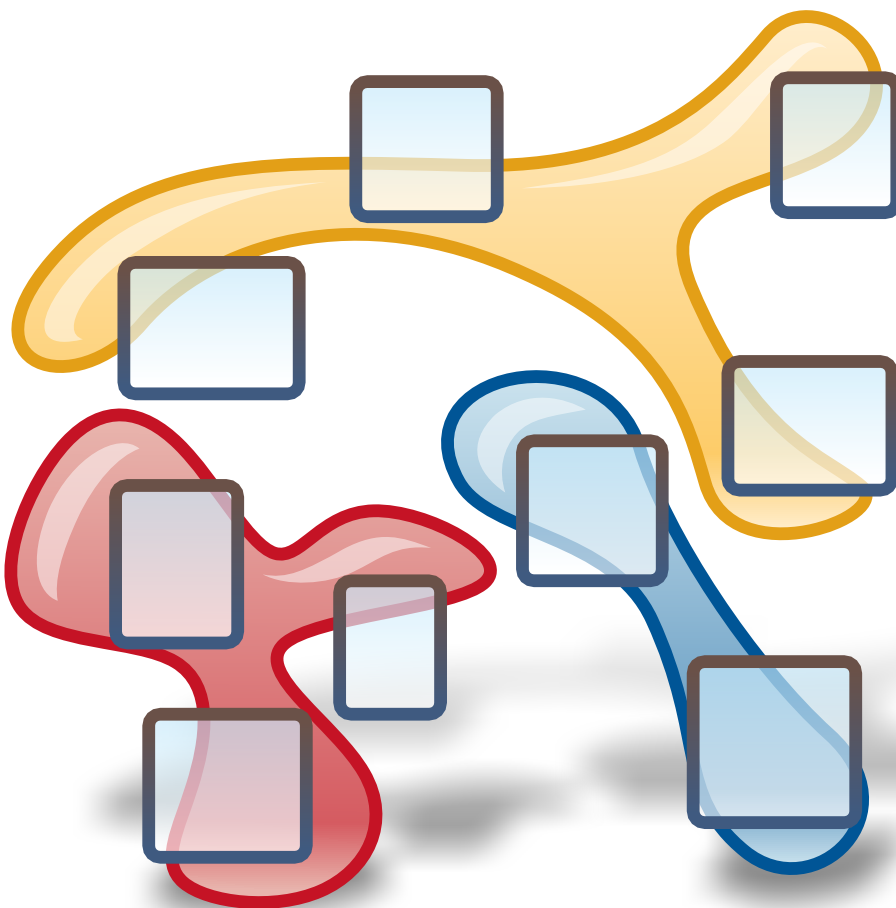




# Code Gestalt

## From UML Class Diagrams to Software Landscapes

**Christopher Kurtz**



April 2011

Thesis advisor:  
Prof. Dr. Jan Borchers

Second examiner:  
Prof. Dr. Armin B. Cremers



I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

*Aachen, April 2011*  
*Christopher Kurtz*





# Contents

<b>Abstract</b>	<b>xxiii</b>
<b>Überblick</b>	<b>xxv</b>
<b>Acknowledgements</b>	<b>xxvii</b>
<b>Conventions</b>	<b>xxix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Goals and requirements . . . . .	4
1.2 Contribution . . . . .	5
<b>2 Related work</b>	<b>9</b>
2.1 Top-level visualizations . . . . .	10
2.1.1 Seesoft . . . . .	10
2.1.2 SHriMP . . . . .	11
Creole . . . . .	13
2.2 Special purpose tool sets . . . . .	13
2.2.1 CodeCrawler . . . . .	14

---

	X-Ray . . . . .	15
2.2.2	Cultivate . . . . .	15
2.3	Program execution . . . . .	16
2.3.1	CallStax . . . . .	17
2.3.2	Extravis . . . . .	18
2.3.3	TraceCrawler . . . . .	19
2.4	SVs based on class diagrams . . . . .	20
2.4.1	Relo . . . . .	20
2.4.2	VisMOOS . . . . .	22
2.4.3	MetricView . . . . .	23
2.5	Map and city metaphors . . . . .	24
2.5.1	UML-city . . . . .	24
2.5.2	CodeCity . . . . .	25
2.5.3	CodeMap . . . . .	26
2.6	Taxonomies and evaluation . . . . .	27
2.6.1	Taxonomy by Price . . . . .	28
2.6.2	Survey by Bassil and Keller . . . . .	30
2.6.3	Evaluation and the need for integration	32
2.6.4	The effect of layout on comprehension	34
2.6.5	Dynamic visualization tools . . . . .	35
2.7	Summary . . . . .	35

---

<b>3</b>	<b>Initial user survey</b>	<b>39</b>
3.1	Design . . . . .	40
3.2	Results . . . . .	41
3.2.1	Background . . . . .	41
3.2.2	Usefulness of software visualizations	41
	Common visualizations . . . . .	42
	Visualizations from research . . . . .	44
	Comments . . . . .	46
3.2.3	Visualization software . . . . .	46
	SV tool users . . . . .	47
	Non-users . . . . .	48
	Comments . . . . .	48
3.2.4	Manual visualization . . . . .	49
	Sketching participants . . . . .	49
	Non-sketching participants . . . . .	50
	Comments . . . . .	50
3.2.5	Software visualization in documenta- tion . . . . .	51
	Comments . . . . .	51
3.3	Summary . . . . .	51
<b>4</b>	<b>Paper prototype</b>	<b>55</b>
4.1	Design . . . . .	55

---

4.1.1	Early concepts . . . . .	56
	Thematic heat map . . . . .	56
	Data trace . . . . .	57
4.1.2	Designs in light of the online survey .	58
	Framework flow . . . . .	59
	Structured context diagram . . . . .	60
	Local context view . . . . .	62
	Diagram widgets . . . . .	64
4.1.3	Designing the paper prototype . . . . .	66
4.2	Implementation . . . . .	67
4.2.1	Project integration . . . . .	69
4.2.2	Creation of a diagram . . . . .	70
4.2.3	Expanding a diagram . . . . .	70
4.2.4	Search a diagram . . . . .	72
4.2.5	Grouping and tagging . . . . .	73
4.3	Evaluation . . . . .	74
4.3.1	Test results . . . . .	75
	Project integration . . . . .	75
	Creating a diagram . . . . .	75
	Editing a diagram . . . . .	76
4.3.2	Further feedback and observations . .	78
4.3.3	Impact on next prototype . . . . .	78

---

<b>5</b>	<b>Silverlight prototype</b>	<b>79</b>
5.1	Design . . . . .	80
5.1.1	Tag overlay . . . . .	81
	Visualization . . . . .	82
	Interaction . . . . .	83
5.1.2	Thematic relation . . . . .	84
	Visualization . . . . .	85
	Interaction . . . . .	86
5.1.3	Class diagram . . . . .	86
5.2	Implementation . . . . .	87
5.2.1	Creating a new diagram . . . . .	87
5.2.2	Expanding an existing diagram . . . . .	88
5.2.3	Tag overlay . . . . .	90
5.2.4	Thematic relations . . . . .	90
5.3	Evaluation . . . . .	93
<b>6</b>	<b>Eclipse implementation</b>	<b>95</b>
6.1	Design . . . . .	95
6.2	Implementation . . . . .	96
6.2.1	Framework . . . . .	96
6.2.2	Other resources . . . . .	97
6.2.3	Eclipse integration . . . . .	98
	Diagram creation . . . . .	98

---

	Editor . . . . .	101
	Complementary Eclipse views . . . . .	102
	Drag-and-drop . . . . .	102
	Code editors . . . . .	103
6.2.4	Class diagram editor . . . . .	103
	Type boxes . . . . .	103
	Inheritance and call relations . . . . .	104
6.2.5	Tag overlay . . . . .	107
	Sweepline algorithm . . . . .	108
	Highlighting . . . . .	109
6.2.6	Thematic relations . . . . .	109
6.2.7	Notes . . . . .	112
6.3	Evaluation . . . . .	112
6.3.1	Goals . . . . .	113
6.3.2	Test design . . . . .	114
6.3.3	Results . . . . .	115
	Population . . . . .	115
	Completion rates and completion time	115
	Errors . . . . .	116
	Qualitative feedback . . . . .	117
	Questionnaire . . . . .	119
	Additional usability questions . . . . .	120

---

Individual features . . . . .	120
6.3.4 Second online survey . . . . .	121
Comparison of sketches with Code Gestalt diagrams . . . . .	123
Qualitative feedback . . . . .	123
Requested features . . . . .	124
6.3.5 Summary . . . . .	126
<b>7 Summary and future work</b>	<b>129</b>
7.1 Summary and contributions . . . . .	129
7.2 Future work . . . . .	131
7.2.1 Implementation . . . . .	131
7.2.2 Diagram customization . . . . .	132
7.2.3 Scalability . . . . .	132
7.2.4 Further evaluation . . . . .	133
7.2.5 Multiple selection in tag overlay . . . . .	133
7.2.6 Additional metrics . . . . .	134
<b>A Additional online survey materials</b>	<b>135</b>
A.1 Survey questions . . . . .	135
A.1.1 Background . . . . .	136
A.1.2 Visualization software . . . . .	139
SV users . . . . .	140
SV non-users . . . . .	142

---

All participants . . . . .	142
A.1.3 Manual visualization . . . . .	143
Sketching participants . . . . .	143
Non-sketching participants . . . . .	144
All participants . . . . .	145
A.1.4 Documentation . . . . .	145
A.2 Results and analysis . . . . .	146
A.2.1 Background . . . . .	146
A.2.2 Visualizations . . . . .	148
A.2.3 Visualization software . . . . .	153
SV tool users . . . . .	154
Non-users . . . . .	161
A.2.4 Manual visualization . . . . .	162
Sketching participants . . . . .	162
Non-sketching participants . . . . .	166
A.2.5 Software visualization in documenta- tion . . . . .	166
<b>B Paper prototype user test</b>	<b>169</b>
<b>C Additional user study materials</b>	<b>173</b>
C.1 User study forms . . . . .	173
C.1.1 Consent form . . . . .	174
C.1.2 Test tasks . . . . .	176



---

C.1.3	User study questionnaire . . . . .	181
C.1.4	Error and clutter evaluation scheme . . . . .	185
Task #1	. . . . .	185
Task #2	. . . . .	186
Task #3	. . . . .	188
Task #4	. . . . .	189
C.2	Results and analysis . . . . .	190
C.2.1	Population . . . . .	190
C.2.2	Completion rates and times . . . . .	191
C.2.3	Errors . . . . .	192
<b>D</b>	<b>Second online survey materials</b>	<b>197</b>
D.1	Survey questions . . . . .	197
D.1.1	Diagram/sketch comparison . . . . .	198
D.1.2	Additional features . . . . .	198
D.2	Results and analysis . . . . .	200
D.2.1	Comparison of sketches with Code Gestalt diagrams . . . . .	200
	<b>Bibliography</b>	<b>203</b>
	<b>Index</b>	<b>209</b>



# List of Figures

1.1	Example of UML class diagram . . . . .	3
1.2	Example of pen and paper sketch . . . . .	4
1.3	Example of the tag overlay . . . . .	6
1.4	Example of thematic relations . . . . .	7
2.1	Seesoft visualization . . . . .	11
2.2	Simple Hierarchical Multi-Perspective view .	12
2.3	SHriMP views in Eclipse . . . . .	13
2.4	System Complexity View in CodeCrawler . .	14
2.5	Cloud View of Cultivate . . . . .	16
2.6	CallStax in web browser . . . . .	17
2.7	Extravis . . . . .	18
2.8	TraceCrawler detail . . . . .	20
2.9	Relo diagram . . . . .	21
2.10	3DRD and VisMOOS view . . . . .	22
2.11	MetricView . . . . .	23

---

2.12	UML-City-View . . . . .	24
2.13	CodeCity . . . . .	25
2.14	Thematic software map . . . . .	27
3.1	Perceived usefulness of common visualizations	43
3.2	Perceived usefulness of research visualizations	45
4.1	Thematic Heatmap Concept . . . . .	57
4.2	Data Trace Concept . . . . .	58
4.3	Framework Flow Concept . . . . .	59
4.4	Structured Context Diagram concept . . . . .	61
4.5	Local Context View concept . . . . .	62
4.6	Tabbed diagram type widgets . . . . .	64
4.7	Type widgets . . . . .	65
4.8	Integration of diagrams with a project . . . . .	69
4.9	Creation of a new diagram . . . . .	71
4.10	Expanding a diagram . . . . .	72
4.11	Filtering a diagram . . . . .	73
4.12	Creating groups . . . . .	74
5.1	Tag overlay concept sketch . . . . .	81
5.2	Concept drawing of thematic relations . . . . .	85
5.3	Adding methods to a type widget . . . . .	88
5.4	Creating a new diagram by drag-and-drop . . . . .	89

---

5.5	Expanding an existing diagram . . . . .	89
5.6	Prototype of the tag overlay . . . . .	91
5.7	Thematic relations in Silverlight prototype . . . . .	92
5.8	Customizing thematic relations . . . . .	92
6.1	Code Gestalt plug.in in Eclipse . . . . .	99
6.2	'New Code Gestalt Diagram' wizard . . . . .	100
6.3	'Create New Diagram From Selection' options . . . . .	101
6.4	Properties view for a selected type . . . . .	102
6.5	The representation of a type . . . . .	104
6.6	Expand options for types . . . . .	105
6.7	Collapse options for types . . . . .	105
6.8	Inheritance relation preview . . . . .	106
6.9	Call relations . . . . .	107
6.10	Tag Overlay . . . . .	108
6.11	Tag overlay sweepline algorithm . . . . .	109
6.12	Highlighting types . . . . .	110
6.13	Highlighting tags . . . . .	110
6.14	Thematic relation contextual controls . . . . .	111
6.15	Code Gestalt note . . . . .	112
A.1	Age and programming experience . . . . .	147



# List of Tables

2.1	SV tool comparison . . . . .	38
3.1	Perceived usefulness of common SVs . . . . .	42
3.2	Perceived usefulness of experimental SVs . . . . .	44
6.1	Usability questions . . . . .	120
6.2	Usefulness of individual features . . . . .	122
6.3	Requested features. . . . .	125
A.1	Source code reading frequency . . . . .	147
A.2	Percentage of unknown code . . . . .	148
A.3	Comparison of common SVs by profession . . . . .	149
A.4	Comparison of common SVs by programming experience . . . . .	150
A.5	Comparison of common SVs by code reading frequency . . . . .	150
A.6	Trends of perceived usefulness over source code reading frequency for common SVs . . . . .	151
A.7	Comparison of common SVs by percentage of unknown code read . . . . .	151

A.8	Trends of perceived usefulness over percentage of unknown code read for common SVs . . . . .	152
A.9	Comparison of SVs from research by profession . . . . .	152
A.10	Correlation between programming experience and perceived usefulness of SVs from research . . . . .	153
A.11	Comparison of SVs from research by code reading regularity . . . . .	153
A.12	Comparison of SVs from research by percentage of unknown source code read . . . . .	154
A.13	SV tools usage frequency . . . . .	154
A.14	Most frequently created SVs . . . . .	155
A.15	Reasons for creating SVs . . . . .	156
A.16	Primarily used SV visualization tools . . . . .	157
A.17	SV tool compatibility with code base . . . . .	157
A.18	SV tool results satisfaction . . . . .	158
A.19	SV tool automation . . . . .	159
A.20	Time requirement per SV. . . . .	160
A.21	Reasons for not creating SVs . . . . .	162
A.22	Frequency of sketching . . . . .	163
A.23	Most sketched software aspects . . . . .	163
A.25	Sketching tools and materials . . . . .	165
A.26	Time needed to create sketches . . . . .	166
A.27	Reasons for not creating sketches . . . . .	167



---

C.1	Completion rates for test tasks . . . . .	191
C.2	Comparison of completion rates . . . . .	191
C.3	Completion times for test tasks . . . . .	192
C.4	Comparison of completion times . . . . .	192
C.5	Error count for test tasks . . . . .	193
C.6	Comparison of error count for test tasks . . .	194
D.1	Comparison of sketches and Code Gestalt diagrams . . . . .	201



# Abstract

*Code Gestalt* is a software visualization tool designed to aid source code exploration and communication between developers. The system implements a novel approach to combine structural code analysis with tag cloud synthesis.

We introduce the *tag overlay* and *thematic relations* to augment partial *class diagrams*. We visualize the vocabulary of a code base in the tag overlay to help programmers find important themes and concepts easily. From the tag overlay the user may create thematic relations. These are hyperedges that are visualized as fans, emanating from a term connecting all types that use that term. In *Code Gestalt* the user may create thematic relations with a diagram alongside inheritance and call relations to connect types. The editor allows the user to seamlessly integrate all kinds of relations in a diagram and freely pick which relations to include. Using affordances and constraints, the system helps the user to find relations and prevents the creation of incorrect diagrams.

The concept is based on an extensive online survey, individual interviews, and repeated user tests with paper and software prototypes. The final software was implemented as plug-in for Eclipse. In a user study we compared the system with pen and paper (the most commonly used tools for software visualization and visual communication).

The study demonstrates that *Code Gestalt* is a software visualization tool with good usability and that our work is a meaningful augmentation for class diagrams. We show that users of *Code Gestalt* make significantly fewer errors than users of pen and paper depending on the task at hand. Only in one case pen and paper users produced significantly fewer errors than *Code Gestalt* users.



# Überblick

Mit *Code Gestalt* stellen wir ein Softwarevisualisierungswerkzeug vor, das Entwicklern beim Verständnis von Quellcode hilft und die Kommunikation darüber mit Kollegen erleichtert. Unser System integriert Ansätze aus der statischen Strukturanalyse von Quellcode mit der Synthese von Schlagwortwolken (Tag Clouds).

In dieser Arbeit werden das *Tag Overlay* und *thematische Relationen* eingeführt, die eine Ergänzung zu partiellen Klassendiagrammen sind. Im Tag Overlay stellen wir das Vokabular des Quellcodes räumlich dar und erlauben dem Benutzer aus den gezeigten Schlagworten thematische Relationen zu erstellen. Thematische Relationen sind Hyperkanten, die als Fächer visualisiert werden und von einem Begriff ausgehend alle Typen verbinden, die diesen Begriff verwenden. In Code Gestalt kann der Benutzer thematische Relationen parallel zu Aufruf- und Vererbungsrelationen zur Verknüpfung von Typen verwenden. Der Editor erlaubt dem Benutzer, sich aller drei Relationsarten zu bedienen und zu bestimmen, welche Relationen so wichtig für eine Visualisierung sind, dass sie in das Diagramm aufgenommen werden. Durch den Angebotscharakter unserer Anwendungsoberfläche hilft das System dem Nutzer, relevante Relationen zu finden, und verhindert durch die Einhaltung von Randbedingungen die Erstellung fehlerhafter Diagramme.

Unser Konzept basiert auf der Auswertung einer Online-Umfrage, Einzelinterviews und wiederholten Nutzertests mit Papier- und Softwareprototypen. Das Endprodukt ist ein Plug-In für Elipse, das wir in einer Nutzerstudie mit Stift und Papier (den gebräuchlichsten Werkzeugen zur Softwarevisualisierung und visuellen Kommunikation) verglichen haben.

Diese Studie zeigt die hohe Benutzerfreundlichkeit von Code Gestalt und die sinnvolle Ergänzung von Klassendiagrammen durch thematische Relationen. Unsere Ergebnisse weisen darauf hin, dass Benutzer von Code Gestalt bei einigen Aufgabenstellungen signifikant weniger Fehler machen als mit Stift und Papier. Nur in einem der untersuchten Fälle begingen Nutzer mit Stift und Papier signifikant weniger Fehler als Code Gestalt-Nutzer.



# Acknowledgements

The author wants to thank the following people for their continuing support for Code Gestalt:

Leonhard Lichtschlag, who witnessed all highs and lows of this thesis with calm.

Prof. Dr. Jan Borchers for helping me find a catchy subtitle and providing steady feedback and encouragement.

Prof. Dr. Armin B. Cremers, Daniel Speicher, and Dr. Günter Kniesel from the *Institut für Informatik III* of the *University of Bonn* for taking interest in the thesis and allowing me to harness the power of *JTransformer* and *Cultivate*.

Jan Nonnen for helping out with anything concerning *SWI-Prolog* and providing me with early builds of *Cultivate*.

Jonathan Diehl, Jan-Peter Krämer, and Dennis Lewandowski for technical assistance.

Sarah Mennicken, Chat Wacharamanotham, and Daniel Spelmezan for helping me with various statistics-related problems.

The *Media Computing Group* at the *RWTH Aachen University* and the *Institut für Informatik III* of the *University of Bonn* for providing support and locations to perform user tests.

The numerous participants of my surveys and user studies for taking their time.

Johanna Nellen for invaluable help with my least favorite document markup language and typesetting system of them all and proofreading the final version of the thesis.

And, most importantly, my parents for having my back.

**Thank you!**





# Conventions

Throughout this thesis we use the following conventions.

**Text conventions** Definitions of technical terms or short excursus are set off in colored boxes.

**EXCURSUS:**

Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:  
*Excursus*

Source code and implementation symbols are written in typewriter-style text.

`myClass`

**Language** The whole thesis is written in US English.

Unidentified third persons are described in male and female form throughout the thesis. It is not the author's claim or intent to use both genders with equal frequency to satisfy political correctness.

**Mathematical symbols** This thesis adapts the statistics notation and methodology from Field [2009]. The following symbols are used to denote common statistics.

- Basic symbols:
  - $df$ : Degrees of freedom
  - $p$ : Probability value
  - $z$ :  $z$ -score
- Descriptive statistics:
  - $M$ : Mean
  - $Mdn$ : Median
  - $N$ : Sample size
  - $n, n_i$ : Sample size (usually a sub set from a larger sample)
  - $q_1$ : First/lower quartile
  - $q_2$ : Second quartile ( $q_2 = Mdn$ )
  - $q_3$ : Third/upper quartile
  - $s$ : Standard deviation
- Test statistics:
  - $H(df)$ : Kruskal-Wallis test statistic
  - $r$ : Effect size (Pearson correlation coefficient)
  - $r_s$ : Spearman's rank correlation coefficient
  - $\tau$ : Kendall's *tau* non-parametric correlation coefficient
  - $U$ : Mann-Whitney test statistic

**Significance and effect size** We consider results with  $p < .05$  to be *significant*, results with  $p < .01$  *highly significant*, and results with  $p \geq .05$  non-significant (*ns*). Effect sizes are categorized as follows:

- *Small effect*:  $.1 \leq |r| < .3$
- *Medium effect*:  $.3 \leq |r| < .5$
- *Large effect*:  $.5 \leq |r|$



# Chapter 1

## Introduction

*“A good sketch is better than a long speech.”*

—*Napoleon Bonaparte*

Maintaining source code is as important and time consuming (if not even more so) than writing it in the first place. Modern programming paradigms aim at creating reusable source code and we are pretty much depending on utilizing existing code for almost any programming task imaginable.

Source code is reused a lot.

Entering the maintenance or development of any existing project means an enormous effort has to be made by the developer in order to get a grasp of the source code at hand. Actually, this has been identified as the most time consuming aspect of program maintenance (Singer et al. [1997]). Hence, there is a strong need for assistance in understanding legacy and third-party code bases.

Understanding code bases consumes a lot of development time.

### **UNKNOWN SOURCE CODE:**

In the following, we will refer to any code bases that are not familiar to the programmer as *unknown*, without regard to where the code actually came from. So, e.g., this shall also include source code a programmer once knew well, or even wrote himself, but forgot about eventually.

Definition:  
*Unknown Source Code*

The larger the code base the less reading code will help a developer to understand it.

Understanding unknown source code is no trivial task. Object oriented design, event driven architectures, and loose coupling scatter code around. For an outsider there is usually no natural order in which to read source code. The larger the code base the less likely it is that source code reading alone will be sufficient to develop an understanding for the inner workings of a program (Ko et al. [2006]).

Visualizations help to manage complexity.

The need to tackle this complexity is a typical use case for software visualization tools. Usually, they produce much more compact views of a given project in comparison with the raw text sources. For this purpose many visualizations have been proposed in the past (see chapter 2), but the predominant formal and automatic way of visualization is the *class diagram* (refer to section 3.2).

Definition:  
*Class Diagram*

**CLASS DIAGRAM:**

*Class diagrams* are a formal depiction of the static structure of a software system. Types are rendered as boxes, containing lists of attributes and methods. Several relations between types are visualized as lines and arrows (see figure 1.1). Since class diagrams are part of the *Unified Modeling Language*, there exists a formal specification for class diagrams [Object Management Group, 2010].

In this thesis however, we will not limit ourselves to the strict formal specification by the *Object Management Group* (the organization maintaining the UML standard), but regard any diagram based on the idea of representing types as boxes and displaying relations as arrows as class diagrams. Most diagrams are not complete, but only represent an aspect of a software system. When we want to emphasize this incomplete nature we will use the term *partial class diagram*.

Many developers prefer pen and paper over software visualization tools.

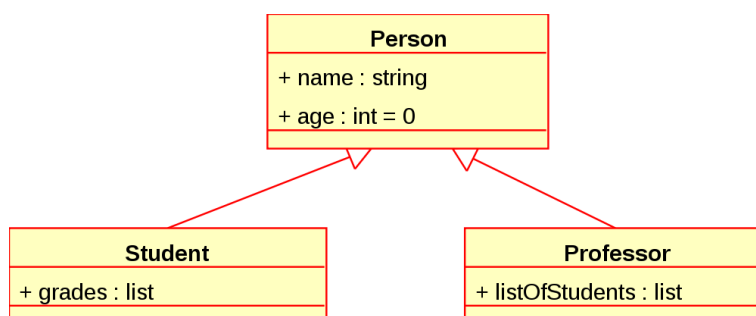
Although there are a number of systems that allow the creation of UML class diagrams, our own survey shows that the predominant method of dealing with unknown code in a visual manner is the creation of informal manual sketches (see chapter 3). Often a developer will track those parts of the program already read through. Another common scenario is that of a programmer explaining an aspect of the code base to a less experienced colleague and creating a

sketch as visual reminder and common point of reference (see section 3.2).

#### SOFTWARE VISUALIZATION (SV):

*Software visualization (SV)*, is a graphical representation of source code. It usually hides implementation details and may include information derived from outside sources such as the history of the code from a version control system. Many, but not all SVs are presented two-dimensional. In the following, SV always carries the meaning that a specialized software program is employed to create the visualization, opposed to drawings created by hand or other means.

Definition:  
*Software  
 Visualization (SV)*

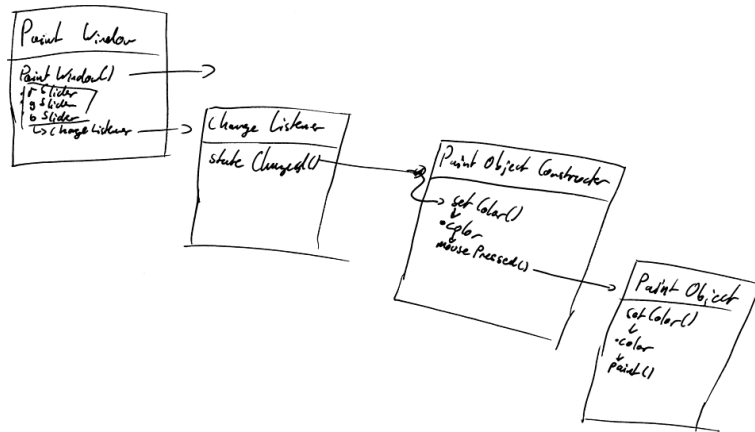


**Figure 1.1:** Example of a UML class diagram created with *Umbrello UML Modeller*. Created by Karl Pietrzak and released under public domain.

#### SKETCH:

In this thesis a *sketch* (see figure 1.2) is always meant to be some form of visualization of a piece of software created with means not explicitly designed to do so. Pen and paper serve as clear example in this regard. But also a graphics tablet with an accompanying general purpose graphics program will create sketches, even though a computer program is involved. The point is that these tools are not explicitly meant for creating visualizations of software, and the user has the freedom and burden to choose how to visualize the code she wants to illustrate.

Definition:  
*Sketch*



**Figure 1.2:** Example of a sketch that combines elements from UML class diagrams and *call graphs*. Created by a tester from our study (see section 6.3).

## 1.1 Goals and requirements

Code Gestalt is supposed to be an alternative for sketching.

Code Gestalt is meant to close the gap between sketches and SVs. I.e., we want to provide a simple, well integrated and easy to use tool that allows the user to quickly create SVs that have practical advantages over sketches. Also, we go beyond common SVs by adding relations that explore thematic dimensions of source code.

We set out to accomplish the following:

1. Use semi-automation features (live previews, drag-and-drop diagram creation, etc.) to support users in creating diagrams and prevent the creation of invalid diagrams that do not match a given code base, but let users stay in control of the diagram scope.
2. Employ a new visualization metaphor that allows the user to harness the human intelligence present in the source code (through code vocabulary analysis) and to better organize the diagram.
3. Achieve good usability and fast diagram creation to compete with pen and paper techniques.



4. The created SVs should be meaningful for any programmer, not only Code Gestalt users, so they can be used for inter-developer communication and project documentation.

## 1.2 Contribution

Related work (see chapter 2) and the results of a user survey (see chapter 3) suggest that, so far, SV tools had several shortcomings and therefore lacked broad user acceptance. Presented with pen and paper, programmers often sketch variations on UML diagrams to illustrate a piece of software.

*Code Gestalt*, the SV tool presented in this thesis, is designed to better accommodate these users. Our approach is two-fold: Speed up diagram creation by system-aided, but user-controlled selective diagram creation, and augment class diagrams with a new visualization that helps a developer to understand a code base.

We want to achieve faster creation and better usability by using interaction techniques like drag-and-drop, live previews, and context sensitive controls. E.g., the user does not draw a type box using a rectangle tool and typing in member names, but simply drags a file from the project browser to the canvas, where a graphical representation of the contained types are automatically created.

Our second focus, a novel SV to aid code understanding, is the main contribution of this thesis. We explore, how a successful visualization, namely class diagrams, can be augmented with unstructured textual information (in the form of tags). Thus these diagrams are expanded to visualize not only structural but also thematic dimensions of a given software (see chapters 4 and 5). We propose two intertwined visualizations: The *tag overlay* and *thematic relations*. We discuss these concepts in detail in chapters 5 and 6.

SV tools are not yet widely accepted in the programming community.

Code Gestalt speeds up diagram creation and provides new tools to support code understanding.

Drag-and-drop, live previews, and context sensitive controls speed up diagram creation.

This thesis introduces the *tag overlay* and *thematic relations*.

Definition:  
*Tag Overlay*

#### TAG OVERLAY:

The tag overlay is an (optional) layer on top of a class diagram, displaying a tag cloud with the terms used in the identifiers of the visualized types as in figure 1.3. The tags are located at a ‘center of gravity’, between the types that use the corresponding term. The force with which each type attracts a tag is determined by the respective term frequency.



**Figure 1.3:** Example of the tag overlay. ‘thickness’ is the most frequent term, followed by ‘color’. The types on the right are drawing tools, the types on the left are UI controllers.

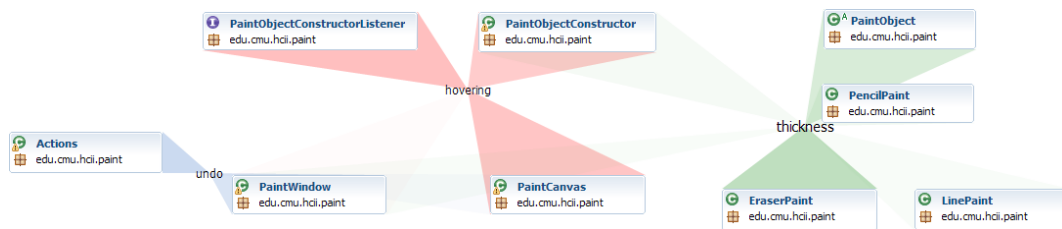
Definition:  
*Thematic Relation*

#### THEMATIC RELATION:

A thematic relation is a relation between all types of a diagram that use a given term in the identifiers of the source code. Code Gestalt visualizes thematic relations as fans, emanating from a tag in the center and connecting all type boxes in the diagram that use that term. An example is shown in figure 1.4. The opacity of a fan element illustrates the term frequency in each connected type.

We implemented  
Code Gestalt as  
plug-in for Eclipse.

Code Gestalt was implemented as plug-in for the open source IDE Eclipse after the concept had matured in two previous prototypes (chapters 4 and 5). We present the implementation and evaluation of the system in chapter 6. Finally, chapter 7 gives a short outlook to the possibilities for



**Figure 1.4:** Example of thematic relations. The types connected by a fan use the center term in at least one of their identifiers.

adopting lessons learned from Code Gestalt for other applications and possible improvements of the Code Gestalt system in the future.



## Chapter 2

# Related work

*“Legacy code often differs from its suggested alternative by actually working and scaling.”*

—Bjarne Stroustrup

In the last two decades, numerous software visualizations have been proposed by the research community. Some of these are special purpose visualizations, while other visualizations try to capture a complete code base with a single picture. Not all of them have been evaluated with respect to usability and user productivity. We take a look at some of those visualizations that share commonalities with the premise of Code Gestalt and influenced its development.

To categorize the tools that create these visualizations a number of taxonomies have been introduced and exploratory studies have been performed. We discuss some of these, as they guided several design decisions for Code Gestalt.

The grouping of tools in this chapter follows no established taxonomy. Instead, we organize the tools in five categories that match design considerations of Code Gestalt. These are top level visualizations, special purpose tool sets, program execution, visualizations based on class diagrams, and map/city metaphors. We do not discuss visualizations for algorithms, since we are only interested in systems that

There exist many SVs, but not all have been properly evaluated.

We discuss some evaluation studies and taxonomies for SV tools.

For easier comparison we group related SVs by development influences for Code Gestalt.

visualize how code bases are structured. We end the chapter discussing some evaluations and taxonomies that have been proposed for SV tools.

## 2.1 Top-level visualizations

Top-level visualizations try to tell everything about a code base in one picture.

Some SVs attempt to capture a code base from a top-level perspective, visualizing as much as possible simultaneously. An off-spring of this type of SVs are the map- and city-based SVs discussed in section 2.5. In the following SVs we will encounter one metric rather frequently, hence we introduce it up front: *lines of Code*.

Definition:  
*Lines of Code (LoC)*

### **LINES OF CODE (LOC):**

A metric, which is used in many SVs, is the number of text lines in the source code that make up a code artifact (e.g., a class). We call this metric *lines of code* or short *LoC*.

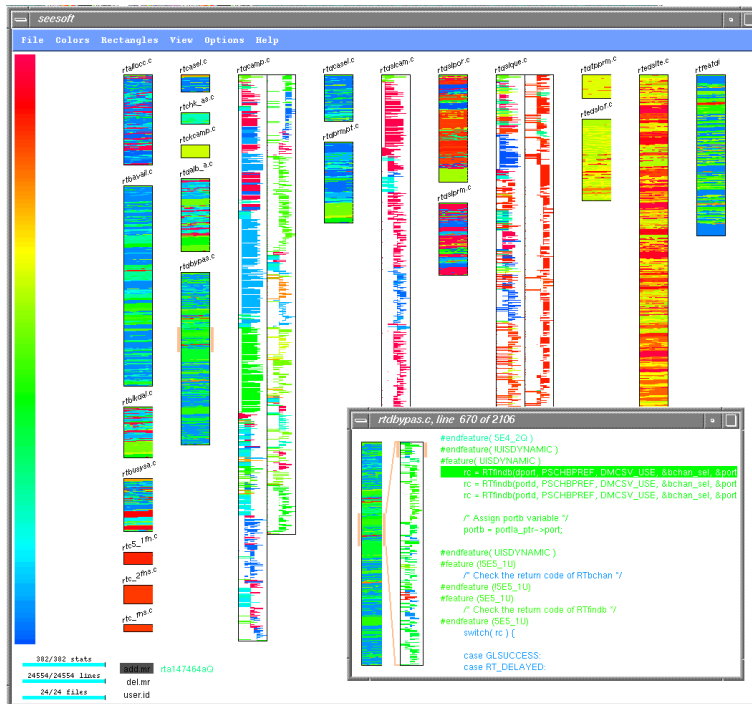
### 2.1.1 Seesoft

*Seesoft* displays type metrics as colored bars.

Eick et al. [1992] presented *Seesoft*, a tool to visualize statistical data for a lot of code at once. Each file of source code is represented by a vertical bar, where each line of pixels represents a line of code. These rows are colored depending on the statistical source chosen by the user. Figure 2.1 shows an example of how *Seesoft* visualizes the age of source code.

Colors indicate the age of code from blue (old) to red (new).

The coloring in the *Seesoft* implementation was originally based on the so-called *MR number*, i.e., the modification request number to a version control system. The user can hover over a color in the scale on the left to look up the actual statistic value represented by that color. Also, when hovering over the diagram, the line of code under the cursor is displayed at the bottom of the view, allowing the user to read the actual code.



**Figure 2.1:** Seesoft visualizes the age of a code base: red lines are newest, blue lines are oldest. Taken from Ball and Eick [1996].

Seesoft implements some direct interaction techniques. When the user hovers over a line of code, only those lines of code with the same statistic values are shown. Similarly, if the cursor hovers over a file name, all lines of code except those with the same values as those in the selected file are shown. Using mouse clicks, these selections can be made permanent and combined. We implemented a similar highlighting feature for the tag overlay of Code Gestalt (see section 6.2.5).

Interaction is limited to browsing and filtering.

### 2.1.2 SHriMP

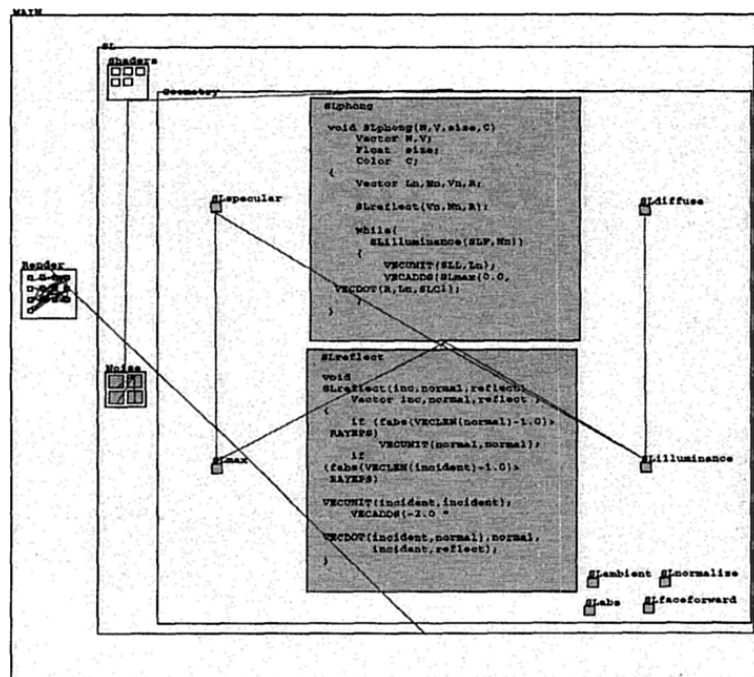
*Simple Hierarchical Multi-Perspective views* (or *SHriMP* views) were introduced by Storey and Müller [1995] and use *fish-eye views* [Sarkar and Brown, 1992] to deal with the complexity of large code bases. SHriMP was originally im-

SHriMP employs fish-eye views for code browsing.

plemented in *Tcl/Tk* as visualization for the reverse engineering and documenting tool *Rigi* [Müller et al., 1993].

The SV uses nested graphs with the source code at the highest zoom level.

The analyzed software is represented as hierarchy of nested graphs with the source code at the most detailed zoom level, through which the user may browse (see figure 2.2). The region of current interest is emphasized by the fish-eye views, while the context of the overall software system is still maintained. The fish-eye view algorithm allows the user to use several focal points at once.



**Figure 2.2:** A SHriMP view, taken from Storey and Müller [1995].

Using SHriMP improves task correctness.

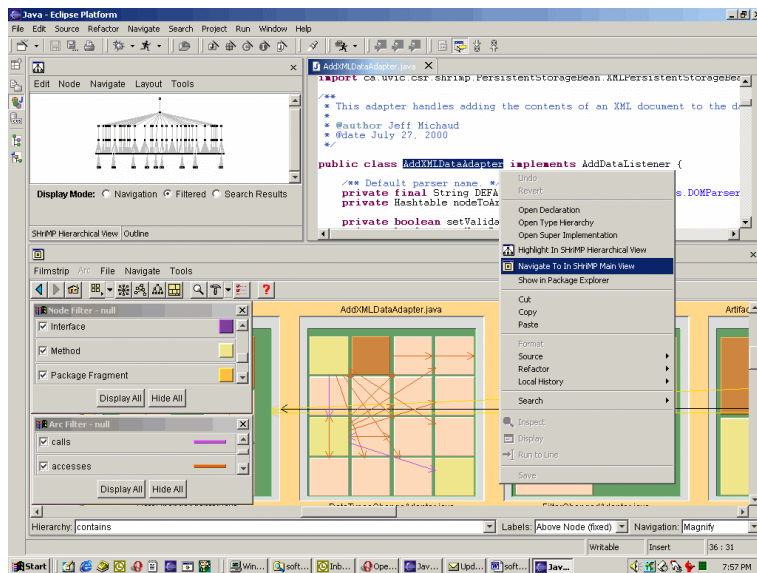
SHriMP has been evaluated against *Rigi* and a command-line interface by Storey [1998], indicating that the use of SHriMP had a positive influence on task correctness results. However, the experiment was biased by learning effects, as the study was not counterbalanced.



## Creole

The SHriMP views were integrated with the IDE *Eclipse* as a plug-in called *Creole* by Lintern et al. [2003]. This implementation was augmented by integrating SHriMP with Eclipse's *CVS* plug-in. This integration called *Xia* allows the system to visualize information based on development history.

The plug-in *Creole* provides SHriMP views for Eclipse.



**Figure 2.3:** Sample configuration of Creole, taken from Lintern et al. [2003]

Creole is one of the tools evaluated in the studies performed by Sensalire and Ogao [2007] and Park and Jensen [2009] discussed in section 2.6.

Creole was used and evaluated in several studies.

## 2.2 Special purpose tool sets

In this section we will discuss visualization tools that allow for more user customization, i.e. specifying how the visualization should be build and what scope it should have. Many of these tools are actually modular tool sets, containing different views for different tasks.

Some of the related work provides suites of special purpose tools.

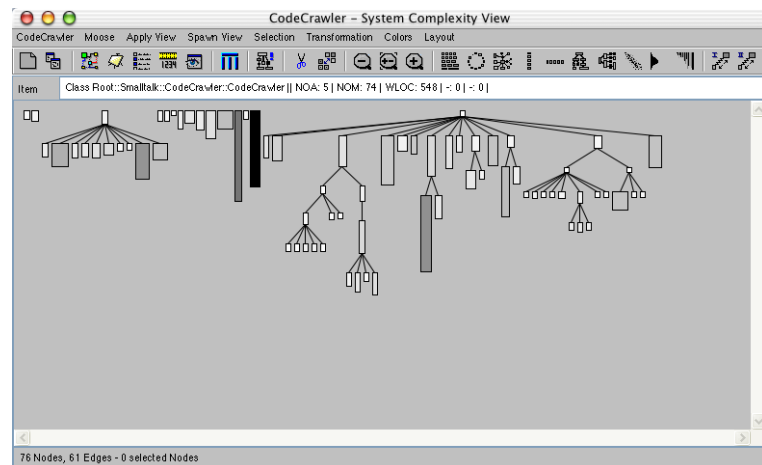
### 2.2.1 CodeCrawler

*CodeCrawler* introduces *polymetric views*.

The system allows the user to specify custom views.

The *polymetric view* was introduced by Lanza [1999] and implemented in *CodeCrawler*. It allows the visualization of up to five metrics per diagram node: position (x- and y-coordinates), size (width and height), and color. Additionally, relations between nodes can be displayed. E.g., inheritance relations can be shown for class nodes.

In *CodeCrawler* the user has access to several layouts and metrics. Some are geared toward visualizing a lot of code at once, others work better for a limited user selection. Figure 2.4 shows an example of the *system complexity view*. This SV is basically a type hierarchy, where width and height of a type are functions of LoC and *NoM*.



**Figure 2.4:** Example of system complexity view in *CodeCrawler* from the *CodeCrawler* homepage.

Definition:  
*Number of Methods (NoM)* and *Number of Attributes (NoA)*

#### NUMBER OF METHODS (NOM) AND NUMBER OF ATTRIBUTES (NOA):

The method count of types is a metric commonly used in many SVs and code analysis tools. This metric is often called *number of methods* or abbreviated with *MoN*. Likewise, the attribute count of a type is called *number of attributes* or *NoA*.

Although many combinations of metrics, layouts and selections can be chosen to create layouts, not all of them may be equally useful. CodeCrawler offers predefined views for software reverse engineering tasks. Most users of CodeCrawler limit themselves to these default views and rarely create new ones [Lanza and Ducasse, 2003].

Users tend to limit themselves to default views instead of creating custom ones.

### X-Ray

CodeCrawler is written in Smalltalk and as such not of much use for practical applications. An Eclipse plug-in that recreates three of the default CodeCrawler views is *X-Ray*, written by Malnati [2007].

X-Ray makes default CodeCrawler views available in Eclipse.

### 2.2.2 Cultivate

*Cultivate*<sup>1</sup> is an Eclipse plug-in that allows the user to visualize a wide array of metrics to analyze a code base. Several views are available to help the programmer keep track of the properties of her source code. Cultivate builds upon *JTransformer*<sup>2</sup>, “a query and transformation engine” for code bases written in Java.

Cultivate provides a variety of code analysis tools.

Figure 2.5 shows the *cloud view* from the Cultivate tool set, a tool that displays a tag cloud for a selected package or Java file. Code Gestalt builds upon the Cultivate API to generate its tag clouds and the tag overlay (see section 6.2). Other tools that are available within Cultivate include:

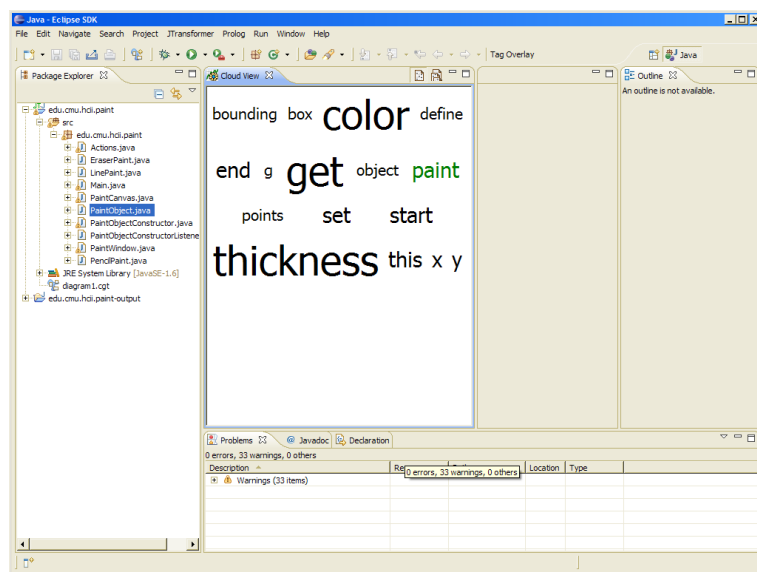
The cloud view displays a tag cloud of a given type or package.

- *Term dependency view* [Speicher and Nonnen, 2010]
- *Class dependency view*: Displays dependencies between types in a graph.
- *Package dependencies view*: Displays dependencies between packages in a graph.
- *Overview pyramid view* [Lanza and Marinescu, 2006]

<sup>1</sup><http://sewiki.iai.uni-bonn.de/research/cultivate/start>

<sup>2</sup><http://sewiki.iai.uni-bonn.de/research/jtransformer/start>

- *Software architecture editor*: For a detailed description refer to the documentation<sup>3</sup>.
- *Metric and smell view*: Searches for code smells, i.e., indicators for problems and the need of refactoring.
- *Smell context view* [Speicher and Jancke, 2010]



**Figure 2.5:** Cultivate’s cloud view shows the terms in the identifiers of the selected type, file, or package.

Some views are customizable and some are context sensitive.

The user is given a lot of control over which metrics to visualize and what scope of source code to analyze. Some views follow the user’s selection in the package explorer of Eclipse and show information regarding the currently selected item.

## 2.3 Program execution

The SVs in this section deal with call graphs and program execution traces.

In this section we will take a look at tools that were designed to give the user a visualization about what parts of an analyzed program are executed in what order. This includes simple call graph visualizations obtained from static

<sup>3</sup><http://sewiki.iai.uni-bonn.de/research/cultivate/tutorial>

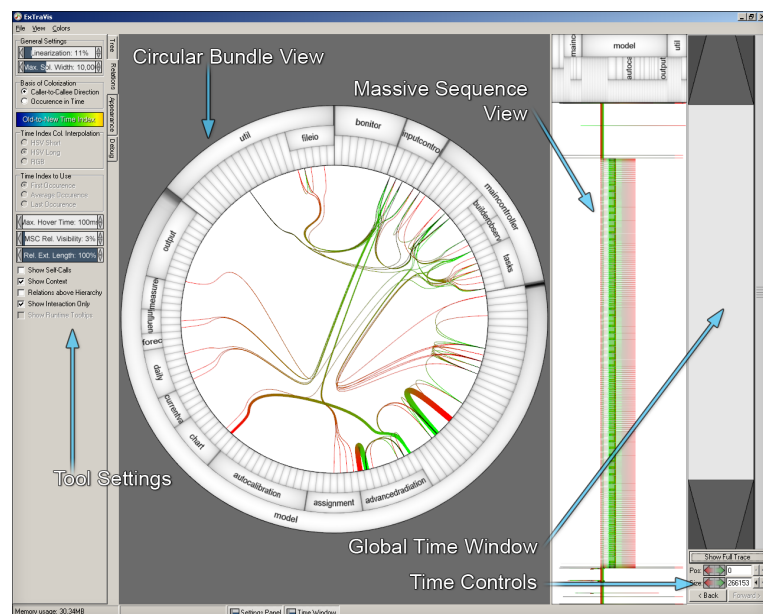


that function. When the user inspects a stack more closely (i.e. moves in closer in the VR environment), further details about the selected function is revealed. The information displayed may come from different sources, such as metrics like the *McCabe complexity* [McCabe, 1976] or profiling information.

### 2.3.2 Extravis

Extravis provides two SVs for execution traces.

This visualization is based on program execution traces. Cornelissen et al. [2007] presented the tool *Extravis* to create so called *massive sequence* and *circular bundle views*. An annotated screen shot is shown in figure 2.7.



**Figure 2.7:** Extravis with both massive sequence and circular bundle views. Taken from Cornelissen et al. [2007].

Circular bundle views display the flow of messages between parts of the source code in a given time interval.

The circular bundle view displays the call-relations between the structural entities of the program on the ring by connecting them with bundled splines. Elements on the ring are organized according to their structural relation (e.g. package inclusion) and can be collapsed and expanded. Relations of collapsed children are bundled and associated with the visible father, thus allowing for abstract

tion. The color of the splines either indicate, when in the execution trace the represented call occurs, or how the call relation is orientated (the latter mode is selected in figure 2.7). The user may choose to limit the amount of visualized information in the view, by restricting it to a time segment of the execution trace.

The massive sequence view is designed to give an overview of the selected part of the full execution trace. Structural information about the program is given at the top of the view, followed by a full record of calls between these program parts ordered along a vertical time-axis.

Massive sequence uses the y-axis for execution time.

The authors present three use cases that demonstrate the application of Extravis for exploration, feature location, and feature comprehension using three Java code bases. Based on repeating patterns in the massive sequence view, the user can identify the occurrences of similar events. The circular bundle view on the other hand is suited to identify which parts of the program communicate with each other at any given time interval. No actual user study was performed.

Extravis is evaluated by use-case discussions only, not actual users.

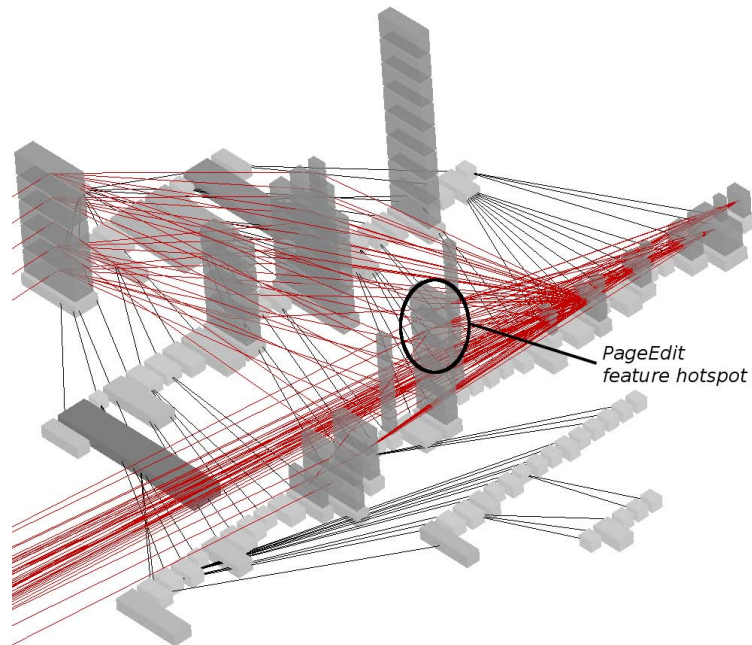
### 2.3.3 TraceCrawler

*TraceCrawler* is an extension of *CodeCrawler* by Greevy et al. [2006] and is based on three-dimensional polymetric views. *TraceCrawler* visualizes not only the static structure of a software system, but also its run-time behavior. *TraceCrawler* is a visualization build on top of *TraceScraper*, a feature analysis tool, and *Moose*, a language-independent reverse engineering platform.

Three-dimensional polymetric views are used by *TraceCrawler*.

The ground plane of the visualization (called *dynamic feature interaction view*) is basically the system complexity view introduced by *CodeCrawler* (see section 2.2.1 and also available via *X-Ray* (compare section 2.2.1). The run-time instances of any type create a new node that is stacked on top of the type node. Currently active nodes are highlighted in green. When a call between two objects occurs, a line is drawn between these objects. An example of this view is shown in figure 2.8.

The third dimension is used to stack the instances of a class. Red lines indicate method calls.



**Figure 2.8:** Detail view of the 'Edit Page' feature from *Small-Wiki*. Taken from Greevy et al. [2006].

The user controls navigation and inspector views.

The interface of TraceCrawler allows the user to navigate forward and backward in an execution trace and thus inspect the creation and destruction of objects as well as the exchange of messages.

## 2.4 SVs based on class diagrams

Several SVs expand class diagrams.

This section discusses approaches that take some aspects of class diagrams and use them as a basis for their own visualizations.

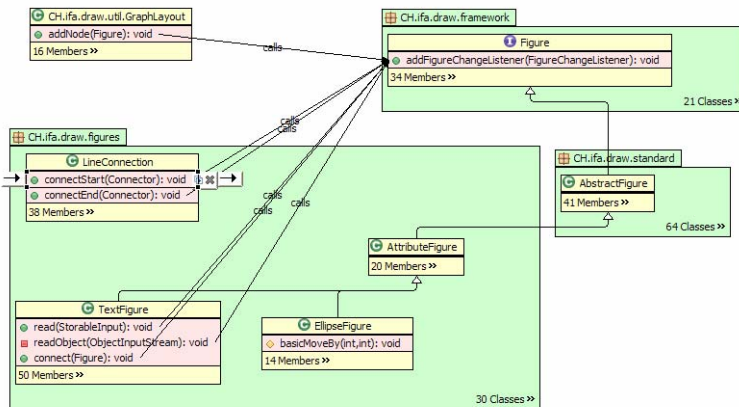
### 2.4.1 Relo

In Relo users build class diagrams iteratively.

*Relo* is an interactive diagram editor for Eclipse presented by Sinha et al. [2006]. Diagrams created with Relo are partial class diagrams. Relo allows the user to incrementally



built a diagram by expanding nodes along inheritance and call relations. A sample for a Relo diagram is shown in figure 2.9.



**Figure 2.9:** A diagram created with Relo. Taken from Sinha et al. [2006].

Types are represented as boxes that contain a (partial) list of members. When the user selects a type or member, he is offered a number of buttons that allow to expand the diagram along relations such as inheritance and method calls. These relations are shown alongside containers that visualize the package structure of the analyzed Java software. The user can view types and methods at different levels of detail, down to a view where the source code of individual methods is directly editable in the diagram editor.

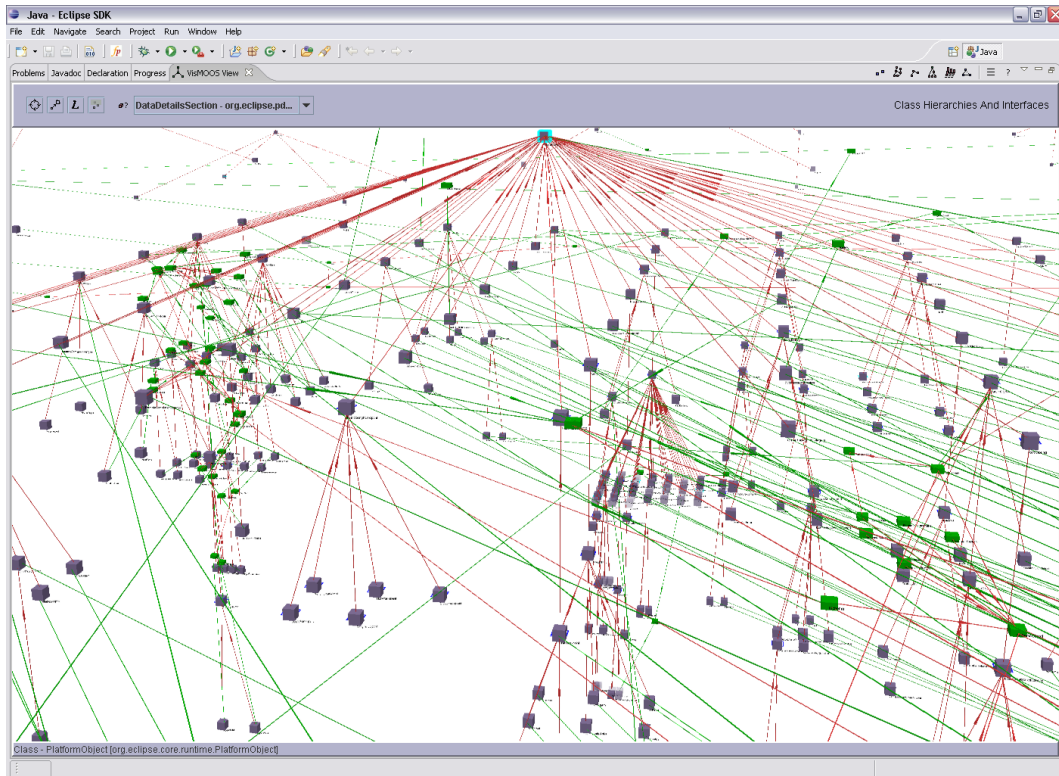
The strength of Relo lies not so much in its visualization than in its interaction techniques. By giving the user control over the visualization, Relo is a tool that promotes code exploration without overwhelming the user with uninteresting and unnecessary details.

Code Gestalt is inspired by Relo's design for the implementation of a basic class diagram editor. We add new features like live previews and the tag overlay on top of that (for details see section 6.1). The design of both the paper (see chapter 4) and Silverlight prototypes (see chapter 5) were inspired by the direct manipulation and expansion metaphor of Relo. For the mock-up of a tag overlay we actually used

The editor offers contextual expansion options depending on the current selection.

The interaction encourages exploration and lets the user control the scope.

Code Gestalt uses and expands many ideas presented by Relo for the class diagram editing functionality.



**Figure 2.10:** A 3d relation diagram in the VisMOOS view. Taken from Fronk et al. [2006].

a screen shot of a Relo diagram as starting point (see section 5.2).

### 2.4.2 VisMOOS

3DRDs use three dimensions to avoid line crossings.

*3d relation diagrams (3DRDs)* are proposed by Alfert and Fronk [2000] to offer more information than two-dimensional class hierarchies. The third dimension is used for displaying additional relations and obtain more layout flexibility to avoid line crossings. The visualization is created by *VisMOOS*, a three-dimensional view plug-in for Eclipse. In the following, we will discuss a refined visualization, as presented in Fronk et al. [2006]. An example diagram is shown in figure 2.10.

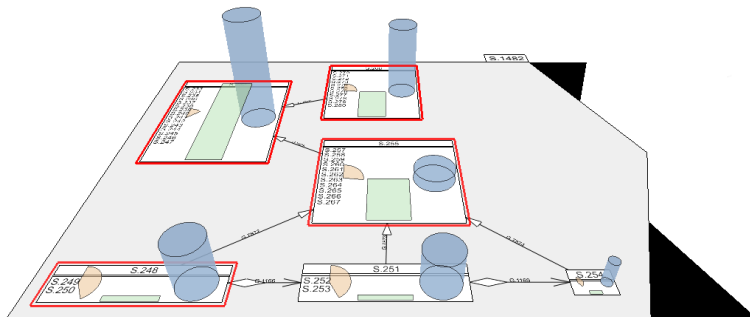
Cylinders represent packages and can therefore be nested. To see inside a package, the user has to double-click the package in the 3d view. Inside, types (represented as boxes) are organized in cone trees, thus building a type hierarchy. Packages and types are represented in different sizes proportional to their LoC. Package and type geometry is connected by dependency relations, using the additional degree of freedom in 3d-space.

Code entities are represented as nested volumes.

### 2.4.3 MetricView

Lange and Chaudron [2007] proposes the *MetricView* as one of four SVs to help a user to better understand the model of a software. In this view, three metrics are displayed inside the type boxes of a class diagram to give a programmer additional information on the visualized classes (see figure 2.11).

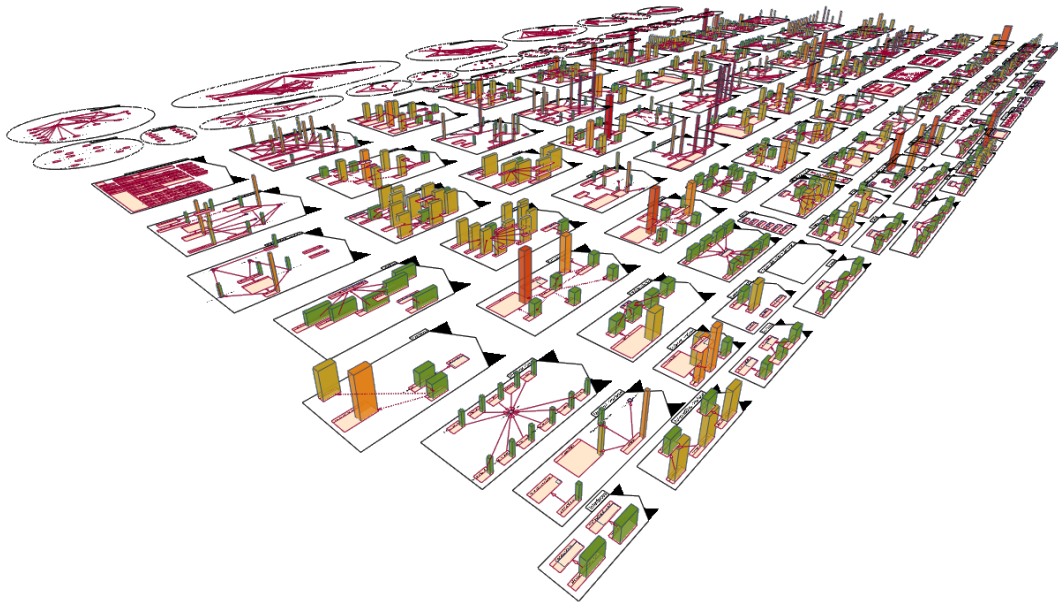
MetricView augments class diagrams with metrics visualizations.



**Figure 2.11:** Example of the MetricView displaying three metrics per type. Taken from Lange and Chaudron [2007].

This idea is similar to the augmentation of type boxes in Code Gestalt, where we enhance them with Eclipse markers and a tag cloud.

Code Gestalt augments class diagrams, too.



**Figure 2.12:** In this UML-City several diagrams are shown next to each other. The 3d-heightbars visualize some metric of a class. Taken from Lange and Chaudron [2007].

## 2.5 Map and city metaphors

Map and city SVs want to provide better orientation in code bases.

A multitude of tools have been developed to visualize code bases as landscapes or cities. The overall idea behind these proposals is that a user is accustomed to using maps as a help of orientation in complex domains (e.g., a public transportation network). Thus it is plausible that the presentation of a code base as map or city can help a programmer in building a mental model of the visualized software.

### 2.5.1 UML-city

UML-cities combine the MetricViews of multiple UML diagrams.

The *UML-city-view* is one of four views proposed by Lange and Chaudron [2007] to improve the user's understanding of a code base. It combines two other views, namely the *MetaView* and the *MetricView* (see section 2.4.3) as shown in figure 2.12.

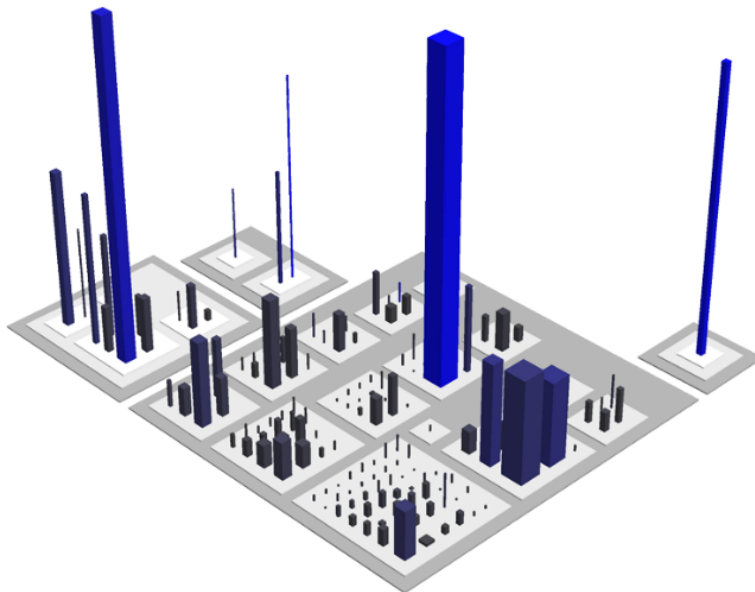
In this view, several diagrams (class, sequence, state, use case) from different stages of the software design process are set in context to each other to allow users to identify objects in different abstraction levels. Each class box is augmented by a three-dimensional height bar, depicting one or two metrics (via the dimensions height and color). This way, a 'city' is visualized.

Class boxes are augmented with three-dimensional metrics visualizations.

### 2.5.2 CodeCity

The *CodeCity*<sup>4</sup> visualization is presented in detail by Wettel [2010]. Software is visualized as a three-dimensional city organized in districts and buildings. Buildings represent classes, while districts represent packages. The properties of a class, such as NoM, NoA, and LoC, are mapped to the dimensions and color of the respective buildings. Figure 2.13 shows a visualization of the CodeCity code base.

In a CodeCity buildings are classes and districts are packages.



**Figure 2.13:** A CodeCity created from the source code of the visualization tool itself. Taken from the CodeCity homepage.

<sup>4</sup><http://www.inf.usi.ch/phd/wettel/codacity.html>

Buildings are shaped using code metrics.

In this visualization, one can detect building archetypes, such as ‘skyscrapers’ (many methods and few attributes) and ‘parking lots’ (few methods and many attributes). It can be augmented by displaying call-relations as bundled edges (refer to section 2.3.2) or color coding other metrics, such as the age of the code.

The user can visualize dependencies and metrics.

The user can interact with the visualization in CodeCity by selecting buildings, marking them by choosing colors or transparency, perform queries, visualize dependencies or code smells (the SVs of the latter are called *disharmony maps*).

### 2.5.3 CodeMap

The layout of thematic software maps is consistent over time.

*Thematic software maps* were introduced by Kuhn et al. [2008] to provide a software visualization layout that would be consistent over time. The visualization is created by the Eclipse plug-in *CodeMap* [Erni, 2010].

Classes are arranged based on their vocabulary and shown as hills on a map.

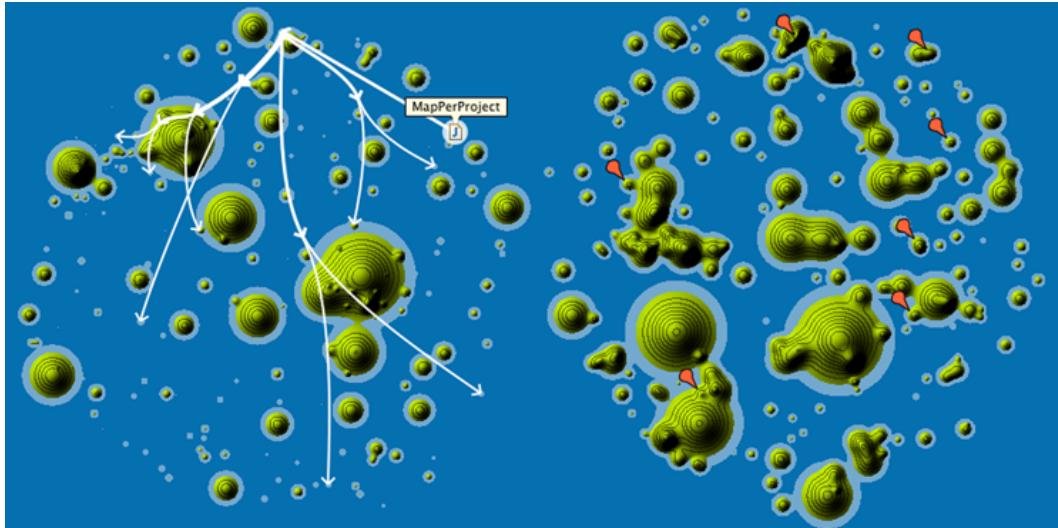
To create the map, the source code of a software project is parsed for all terms that are used in it. We denote the number of terms with  $n$ . Afterwards, all classes are located in an  $n$ -dimensional space, where each dimension is a term, and the position of a class in this dimension is determined by a term weight. This space is flattened to a two-dimensional plane, where each class is visualized as a shaded hill with a height corresponding to its LoC.

Additional information can be displayed on the map.

This software map can be used for a multitude of visualizations, such as call relations and search results as shown in figure 2.14. The user may display several metrics on this map, such as a heat map of editing or the code coverage by test cases.

Code Gestalt’s tag overlay is an inverse of the thematic software map.

Code Gestalt shares many similarities with thematic software maps, but reverses the creation process to allow for more flexibility and user control. While CodeMap creates a map purely based on term analysis and allows the user to display structure information and metrics afterwards, we start with an editable diagram based on the static structure



**Figure 2.14:** A thematic software map created with CodeMap displaying a call graph (left) and search results (right). Taken from Erni [2010].

of the code base and offer the user to augment that diagram with thematic relations in a second step.

## 2.6 Taxonomies and evaluation

Up to the early 21<sup>st</sup> century, many visualizations were only proposed and implemented, but not properly evaluated. It became apparent however that not all visualizations are equally useful. Also, the questions of what the design space for SVs looks like and where existing tools would fit in this space needed to be answered.

Taxonomies give a design space for SVs, while evaluation tells us, which designs are useful.

### 2.6.1 Taxonomy by Price

The taxonomy categorizes SVs in six major sections: Scope, content, form, method, interaction, and effectiveness.

Price et al. [1993] suggested a taxonomy for SVs based on a categorization of software visualization in the following major sections:

- *Scope*: The general characteristics of the SV
- *Content*: The contents of the SV diagram
- *Form*: The elements used in the SV
- *Method*: The specification of the SV
- *Interaction*: The interaction and control techniques to work with the SV
- *Effectiveness*: The quality of the SV

Each section contains several sub-sections.

Each of the major sections is divided into several sub-sections. E.g. *Form* is divided into *Medium*, *Graphical Elements*, *Colour*<sup>5</sup>, *Animation*, *Multiple Views*, and *Other Modalities*.

We demonstrate the taxonomy using the Eclipse implementation of Code Gestalt.

The taxonomy is demonstrated by categorizing seven SV tools (none of which is discussed in this chapter, as they are mostly limited to older or non-procedural programming languages such as Pascal, Prolog and Logo). To illustrate the taxonomy, we categorize Code Gestalt according to this taxonomy:

#### 1. Scope:

- (a) *System/Example*: System
- (b) *Program Class*: Java
- (c) *Scalability*: Screen space and visibility limits
- (d) *Multiple Programs*: Yes
- (e) *Concurrency*: Limited
- (f) *Benign/Disruptive*: Benign

---

<sup>5</sup>[sic], the taxonomy was presented in a paper written in Canadian English



## 2. Content:

- (a) *Program/Algorithm Visualization*: Program
- (b) *Code Visualization*: Through IDE editor
- (c) *Data Visualization*: No
- (d) *Compile/Run-Time*: Compile
- (e) *Fidelity and Completeness*: Partial

## 3. Form:

- (a) *Medium*: Workstation
- (b) *Graphical Elements*: 2d graphic primitives and text
- (c) *Colour*: Yes
- (d) *Animation*: No
- (e) *Multiple Views*: Yes
- (f) *Other Modalities*: None

## 4. Method:

- (a) *Specification Style*: Graphical editor
- (b) *Batch/Live*: Live
- (c) *Fixed/Customizable*: Customizable
- (d) *Code Familiarity*: Some
- (e) *Invasive*: No
- (f) *Customization Language*: Interactive
- (g) *Same Language*: N/A

## 5. Interaction:

- (a) *Navigation*: Yes
- (b) *Elision*: Yes
- (c) *Temporal Control Mapping*: N/A

## 6. Effectiveness:

- (a) *Appropriateness and Clarity*: subjective
- (b) *Experimental Evaluation*: Yes
- (c) *Production Use*: No

The dimensions of the taxonomy have not been examined in detail.

This taxonomy is used as a point of reference for several other works. E.g., the questionnaire in Bassil and Keller [2001] is roughly based on the categories of this taxonomy. However, to our knowledge a detailed evaluation of its dimensions has not yet been conducted.

Compared to other works the Price taxonomy is a good compromise between completeness and level of detail.

The benefits of the Price taxonomy against other suggestions (Myers [1990], Stasko and Patterson [1992]) is its simplicity and relative completeness. One of its drawbacks is that it does not clearly draw up the actual design space of SVs, as many sections and sub-sections are wide open to subjective interpretation (*Effectiveness*) or require their own design space to determine their limits (*Graphical Elements*). A slightly more fine grained classification of effectiveness and visual aspects can be found in Sensalire et al. [2008], while Stasko and Patterson [1992] provide more detail with regard to animation and algorithm visualization.

### 2.6.2 Survey by Bassil and Keller

One of the largest exploratory surveys of SVs was conducted in 2001.

Bassil and Keller [2001] conducted a user survey among 107 participants, asking them about the SV tools they currently employ (Part I). A second questionnaire (Part II) was filled by 41 “expert users” from the first survey.

107 users were questioned about their expectations of SV tools.

**Part I:** This part consists of 21 questions organized in six groups. Among background questions this part asked the participants about functional expectations, practical expectations, evaluation of a concrete tool, and how to improve that tool.

The survey asked about functional...

In summary, many functional aspects were deemed important for SV tools. E.g. navigation, search, the use of colors, hierarchical representations, and source code visualization in textual form. The functional aspects of SV tools that were not perceived as useful by the users were animation effects, 3d representations, and the ability to record the steps of an arbitrary navigation.

Among the practical aspects of SV tools that participants found useful were content and quality of the documentation, tool reliability, ease of using the tool, quality of the user interface, and ease of visualization of large-scale software.

... and practical aspects of SV tools.

These expectations were confronted with an evaluation of an “SV tool at hand”. In summary, these tools held up to some expectations and failed others. Ease of use, usefulness of the generated results and the confidence in the generated results were all perceived positively with more than 50% approval. Gaps between expectations and the reality of tools were identified at the quality of the user interface and the ease of use (72% of the users rated this aspect ‘very important’, yet only 63% indicated ‘high’ approval for that aspect for the “tool at hand”).

Finally, users reported on their experience with SV tools.

Insufficient documentation and code complexity reduced the effectiveness of an SV tool the most. The survey also identified the gains through SV as perceived by the survey participants. 15 tools offered time savings, six better code comprehension, and four tools improved productivity and complexity management each.

SV tools offered time saving advantages, but suffered from insufficient documentation.

**Part II:** This part was designed to gain some insight into what current SV tools provide in terms of code analysis support. The results yield that the majority of tools allow the visualization of call graphs (75%), the visualization inheritance graphs (71%), and the separation of several levels of detail in separate windows (63%). Aspects that were not well represented were function cloning detection (25%) and metrics calculation (21%).

Expert users classified SV tools.

The participants were also asked what they would like to see in future SV tools. The answers were strongly influenced by the tools they already used. Among the most wanted features were call graphs with recursion detection, subsystem architecture graphs with relation to other visualizations, inheritance graphs with multiple inheritance detection and direct access to source code from call graph nodes.

SV tools shape the users’ expectations of desirable features.

We designed Code Gestalt to close identified gaps and avoid usability pitfalls.

The results from this survey heavily influenced our approach for Code Gestalt. The focus of our prototypes was specifically geared towards high usability and a high quality user interface. Also, the findings on low animation and 3d acceptance were instrumental in pursuing a clean two-dimensional approach and pushing back the use of animations. In response to the findings of Part II we do not allow for code artifacts to appear as multiple diagram widgets, thus realizing automatic recursion detection in the call graphs. Another consequence of the survey we realized in Code Gestalt is that all diagram elements that relate to code artifacts can be double-clicked in order to directly jump to the corresponding source code.

### 2.6.3 Evaluation and the need for integration

Many SVs are lacking IDE integration.

Charters et al. [2003] observed that many of the SV tools are in fact stand-alone programs, which are cumbersome to use on an everyday basis. The authors criticize the limitation of the science community to improving several aspects of SVs, without much regard for actual usability through IDE integration.

Users desire better integration.

Sensalire and Ogao [2007] asked five expert programmers to compare three representative SV tools (*Code Crawler* (see section 2.2.1), *Creole* (see section 2.1.2), and *Source Navigator*<sup>6</sup>), to derive a list of features, desired by expert users, but are not provided by current SV tools. The features in high demand were:

- IDE integration
- speed of graph generation and overall responsiveness
- ability to work with arbitrary code bases without reformatting or conversion
- search functionality for SVs
- editable graphs
- simplicity and clarity

---

<sup>6</sup><http://sourcnav.sourceforge.net/>

One year later, Sensalire et al. [2008] suggested a new taxonomy for SV tools based on the following categories:

- *Effectiveness*: Does the tool help the user in solving the problem it was designed to address?
- *Tasks supported*: limited to code smell detection, code refactoring, trace analysis, and debugging support
- *Availability*: license and platform
- *Techniques used*: visualization and interaction techniques

15 tools were evaluated by 16 professional developers. Each developer was assigned one to three tools for evaluation based on his or her background. The results indicate that the usability of an SV tool heavily depends on an appropriate set of available interactions, where too many options are as bad as too little. A tight integration of SV presentation and analysis (queries and searches) is essential in this regard. It is also important that users can tune the visualization in terms of color, layout, and annotations. In general, the use of colors, multiple views, user interaction, and navigation were well implemented in the analyzed tools. Users were much more willing to use SV tools that were properly integrated with their accustomed IDE. However, more than 50% of the users questioned the need for visualization tools that went beyond of what Eclipse had to offer with its built-in views. Moreover, users tend to prefer static two-dimensional SVs over animated or three-dimensional visualizations, although the data on this is not completely conclusive, as not many of the analyzed tools employed these techniques.

Park and Jensen [2009] investigated, if SVs help newcomers to open source projects to get acquainted with the new code base. The authors conducted a study with three groups: Group A, the control group, was only allowed to use the resources offered by sourceforge<sup>7</sup> and the built-in tools of Eclipse; group B was additionally given the Eclipse plug-ins *Version Tree*<sup>8</sup> and *Creole* (see section 2.1.2). Group C had

Sensalire et al. [2008] propose a simpler taxonomy.

16 professional users evaluated existing tools. More than half of them did not see any advantages of employing SVs over built-in Eclipse views.

The right SV tools can aid newcomers to an open source project. Plug-ins did not perform better than stand-alone tools.

<sup>7</sup><http://sourceforge.net/>

<sup>8</sup><http://versiontree.sourceforge.net/>

access to the stand-alone tools *Augur* [de Souza et al., 2005] and *SA4*<sup>9</sup> which roughly offered the same feature set as the plug-ins used by group B. Park and Jensen asked participants questions about the project history and organization (“Who contributed the most in this package?”) and some counting tasks about the code base (“How many classes are in this package?”). They obtained mixed results for the visualization tools. Source code reading was sometimes more effective and efficient than looking at SVs. There were indications that the use of the right SV tool for the right task lead to slower performance and higher correctness. However, the usefulness of SV tools was highly dependent on the nature of the given task.

#### 2.6.4 The effect of layout on comprehension

Arranging classes in clusters depending on the number of relations between them is more useful than enforcing strict orthogonal layouts.

Sharif and Maletic [2009] performed an experiment to determine what effect layout has on the comprehension of UML class diagrams. The study compared an *orthogonal layout* (90 degree bends, minimized line crossing) with a *multi-cluster layout* (grouping of classes based on number and importance of connections and association between them). The participants’ accuracy, confidence, and time to complete 22 tasks was measured. The results favor multi-clustering layout. The layout impacts accuracy (medium effect) in general, with the exception of refactoring and addition tasks (small effect). Speed in general is influenced (low to medium effects, depending on the software framework), especially for bug fixing, but not for refactoring tasks. Also, the multi-cluster layout positively influences confidence and comprehension preference scores. However, the layout does not impact the design experience or aesthetic preference scores.

<sup>9</sup><http://www.alphaworks.ibm.com/tech/sa4j>

### 2.6.5 Dynamic visualization tools

An evaluation of dynamic visualization tools is given by Pacione et al. [2003]. Five tools from the field of research are tested against general software comprehension and reverse engineering tasks. The findings of the evaluation reveal that the tested tools are more successful in dealing with the later than the former. On average, the tools were only able to answer roughly a third of the questions about the code base (*HotDraw*<sup>10</sup>). The authors conclude with the observation that the dynamic tools examined are not able to answer large-scale questions, e.g., about the high-level architecture of the software or present design patterns. Small-scale questions about data flow and object interaction however could be answered more satisfactory and more completely by some tools.

The investigated tools were not able to help with answering questions about a code base on a large-scale.

## 2.7 Summary

Existing SVs have influenced our approach to Code Gestalt. The five rudimentary concepts that we looked at for inspiration were the following:

Related work influenced many aspects of Code Gestalt's design.

- *Top-level visualizations*: They try to capture a large part or even the whole code base in a single picture.
- *Special purpose tool sets*: These collections of different smaller tools that may work in concert or on their own to deal with specific problems.
- *Program execution*: These SVs try to illustrate the behavior of a program at run-time and visualize calls between functions or methods.
- *SVs based on class diagrams*: Some SVs take advantage of the user's familiarity with class diagrams and expand it.
- *Map and city metaphors*: A certain kind of mostly top-level SVs that try to help a user orient in a code base by rendering it as a city or map.

<sup>10</sup><http://st-www.cs.illinois.edu/users/brant/HotDraw/HotDraw.html>

We decided to use an editable class diagram as baseline of our SV.

Code Gestalt is influenced by these approaches to varying degrees. Although a starting point for our development, we moved away from the ‘one image tells it all’ visualizations when designing Code Gestalt, and considered how elements from city and map SVs could serve our design goals. Together with class diagrams as a baseline the map metaphor is certainly the most relevant related SV. While class diagrams allow the user to arrange his diagram more freely, city and map SVs usually prescribe fixed layouts. We integrate both approaches by keeping a flexible class diagram editor as our primary user interface and SV, but allow the user to display a tag overlay (a very simple thematic map) on top of it as an individual layer (see section 6.2.5). Of course, we sacrifice the consistency achieved by Kuhn et al. [2008]. Since our tool is not meant to generate diagrams automatically, we believe this is no significant drawback.

Data flow and execution traces could not be easily incorporated, as they introduced clutter.

The other three categories had less influence on the final draft of Code Gestalt, but were explored in various preliminary concepts, and some ideas carried over to the final implementation. Code Gestalt allows for rudimentary program execution and data flow analysis using call graphs, but not execution traces for reasons described in section 4.1.1. Through its tight integration with the Eclipse IDE, Code Gestalt can be used as special purpose tool, side-by-side with any number of other plug-ins, although it is designed to be a general purpose editor. Last, but not least, Code Gestalt can create top-level visualizations, if desired, but does not provide dedicated features to support the creation of very large diagrams.

Code Gestalt puts an emphasis on interaction techniques and diagram customization.

Table 2.1 lists the visualization tools from this chapter in comparison to Code Gestalt as presented in chapter 6. Of course, the table simplifies and abstracts from many details, and is not a complete categorization of all presented tools (see section 2.6.1). The comparison shows that we put an emphasis on user interaction and diagram customization. While other tools are capable of creating larger-scale or more detailed SVs, we decided on a visualization at the class and member level based on our survey finding presented in chapter 3. Code Gestalt is limited in so far as it offers only one optional overlay (the tag overlay) based on a single metric (similarity in code vocabulary). Addi-



tional information is given by indicating compiler errors and warnings for all diagram elements, something we have not found in any other SV tool, except the built-in Eclipse views. With regard to the range of available metrics tools like CodeMap, Cultivate, and MetricView allow for more flexibility. However, we will discuss in chapter 7 how additional metrics can be integrated with Code Gestalt.

The evaluation and surveys performed by other authors gave us direction what functional and practical features were the most important when designing Code Gestalt. The goals we described in section 1.1 are partially derived from it. Some solutions to our requirements can be recognized in several SVs and tools presented in this chapter. Especially with regard to usability and speedy diagram creation, many lessons can be learned from previous work. Also, the evaluation of SVs has guided our design in finding forms of presentation which are intuitive and meaningful for users.

However, related work does not exemplify how we could build an SV that aids users in understanding source code on the basis of anything other than code syntax or structure. Instead, we are interested in a property that is rather the product of human intelligence and understanding and making it accessible to the user. Also, with the exception of Relo, we have found no examples of systems that let users stay in control of the visualization scope, while providing strong support through semi-automation. During the development of Code Gestalt we dealt with both these research questions, as presented through chapters 4–6.

The development of Code Gestalt followed a cycle of design, implementation and analysis (*DIA cycle*). The designs from section 4.1.1 were implemented as paper sketches and evaluated using discussions with co-workers and related work. Since we did not find any related work on what programmers do in lack of an SV tool that suit their needs we conducted our own investigation into this matter, which is presented in chapter 3.

Related work points out pitfalls and important features, presentation and usability questions.

Open research questions remain.

We developed Code Gestalt using the *DIA-cycle*.

Tool	Scale	Presentation	Metrics	Interaction	Edit.	Integr.
CallStax	Functions	3d primitives	McCabe complexity, LoC, code distribution	Zooming, filtering	No	No
Code Gestalt	Classes, methods, attributes	Class diagram, tag cloud	Code vocabulary, errors/warnings	Zooming, filtering, toggle overlays, exp. & collapse	Yes	Eclipse
CodeCity	Packages, classes	3d city	NoM, NoA, LoC	Zooming, filtering, toggle overlays	No	No
CodeMap	Classes	Topographic map	Code vocabulary, LoC, additional customizable	Zooming, toggle overlays	No	Eclipse
Creole	Top-level .. source code	Nested 2d graphs	None	Zooming, filtering, expand & collapse	Yes	Eclipse
Cultivate	Mixed	2d graphs, tables, text	Customizable	Dependent on the view	Mixed	Eclipse
Extravis	Top-level, runtime	2d diagrams	None	Zooming, filtering	No	No
MetricView	Classes, methods, attributes	UML diagrams, 3d primitives	Customizable	Zooming, creation of overlays	Yes	No
Relo	Packages .. source code	Class diagram	None	Zooming, expand & collapse	Yes	Eclipse
Seesoft	Top-level, files	2d primitives, heat map	Code age, file size	Filtering	No	No
TraceCrawler	Top-level, runtime	3d primitives, source code	NoM, LoC	Zooming	No	No
VisMOOS	Packages, classes	3d graph	LoC	Zooming	No	Eclipse
X-Ray	Packages, classes	2d graphs	NoM, LoC	Zooming, toggle overlays	No	Eclipse

**Table 2.1:** A comparison of several SV tools and Code Gestalt.

## Chapter 3

# Initial user survey

*“USA Today has come out with a new survey — apparently, three out of every four people make up 75% of the population.”*

*—David Letterman*

While reviewing the related work, we wanted to get a better understanding and quick overview about what worked well with available SV tools and where we could make out potential for improvement. In practice, our co-workers seemed to prefer pen and paper or marker and white board over any SV software. If that observation was confirmed by the findings of a survey, what were the factors causing this?

To gain more insight into these matters, we conducted an online survey. It ran for ten days (November 6th – November 15th, 2009) and was primarily web based, complemented with additional personal interviews with student assistants from the *Media Computing Group* at the *RWTH Aachen University*. The link to the bilingual (English and German) online questionnaire was posted on roughly a

Observation:  
Programmers rather use pen and paper than specialized SV software.

An online survey was conducted for ten days.

dozen websites<sup>1</sup>, among them the *Apple* and *Microsoft* development communities, and some university mailing lists.

### 3.1 Design

The questions of the survey are divided in five groups.

The survey consists of five parts:

1. Background
2. Usefulness of software visualizations
3. Visualization software
4. Manual visualization
5. Software visualizations in documentation

20 to 29 questions were presented depending on the participant's answers.

The survey contained a total of 31 questions. Since some questions are mutually exclusive based on answers given in previous questions, a minimum of 20 and maximum of 29 questions can show up for any individual participant. The complete survey is reproduced in appendix A.1.

The survey generated 128 full responses.

A total number of 128 full responses was recorded by the survey software *LimeSurvey 1.86*<sup>2</sup>. To our knowledge, this is the largest number of participants to be questioned for a survey on SVs. The remainder of this chapter will focus on the results of the survey and our conclusions leading to the development of the first prototype.

---

<sup>1</sup><http://www.apple.com>, <http://www.c-plusplus.de>,  
<http://www.codecall.net>, <http://www.daniweb.com>,  
<http://www.dreamingcode.net>, <http://www.free2code.net>,  
<http://www.gamedev.net>, <http://www.infostudium.de>,  
<http://msdn.microsoft.com>, <http://www.programmingforums.org>,  
<http://www.tutorial.de>

<sup>2</sup><http://www.limesurvey.org>

## 3.2 Results

This section will present the questions of the survey and the analysis of the participants' answers to them. The data was exported from LimeSurvey to *Excel 2010*<sup>3</sup> and reformatted, so it could be processed by Excel's data analysis tools and *IBM SPSS Statistics 19*<sup>4</sup>. For basic descriptive statistics, we used the built-in functionality of LimeSurvey. The interested reader can find details on the results and their statistical analysis in appendix A.2.

Details of and more data from the survey can be found in appendix A.2.

### 3.2.1 Background

In the first part of the survey we asked the participants about their background such as their current occupation, age, and programming experience. Our group was primarily male (98.4%), but diverse with regard to age, occupation, and experience. The 'median participant' is a male student of age 25, who reads source code on a daily basis (40% or less of that is unknown code) and has 10 years of programming experience. For more details see appendix A.2.1.

Our participants were mostly male, but covered a wide range of age, experience, and work practice.

### 3.2.2 Usefulness of software visualizations

In this part we investigated how useful the participants rate existing SVs. We asked them to look at both common SVs like UML diagrams and experimental SVs from research.

We presented 11 SVs to the participants

---

<sup>3</sup><http://office.microsoft.com/en-us/excel/>

<sup>4</sup><http://www.spss.com/software/statistics/>

### Common visualizations

The participants rated seven common visualizations for their usefulness.

The participants were shown a series of common visualizations:

- Class diagram
- Call graph
- Flowchart
- Data flow diagram
- Software layer diagram
- Sequence diagram
- Package diagram

All SVs were illustrated with examples.

Each visualization was illustrated by a sample diagram, shortly explained, and linked to respective articles on *Wikipedia*.<sup>5</sup> Based on this information, the participants were asked to rate the diagram as ‘useful’, ‘not useful’, or ‘don’t know/never used’<sup>6</sup>. Table 3.1 shows an overview of the responses.

Visualization	Useful	Not useful	Don’t know/never used
Class diagram	80.47%	9.38%	10.15%
Flowchart	61.72%	28.91%	9.37%
Sequence diagram	44.53%	21.88%	33.59%
Software layer diagram	43.75%	23.44%	32.81%
Call graph	38.28%	22.66%	39.06%
Data flow diagram	36.72%	24.22%	39.06%
Package diagram	24.22%	30.47%	45.31%

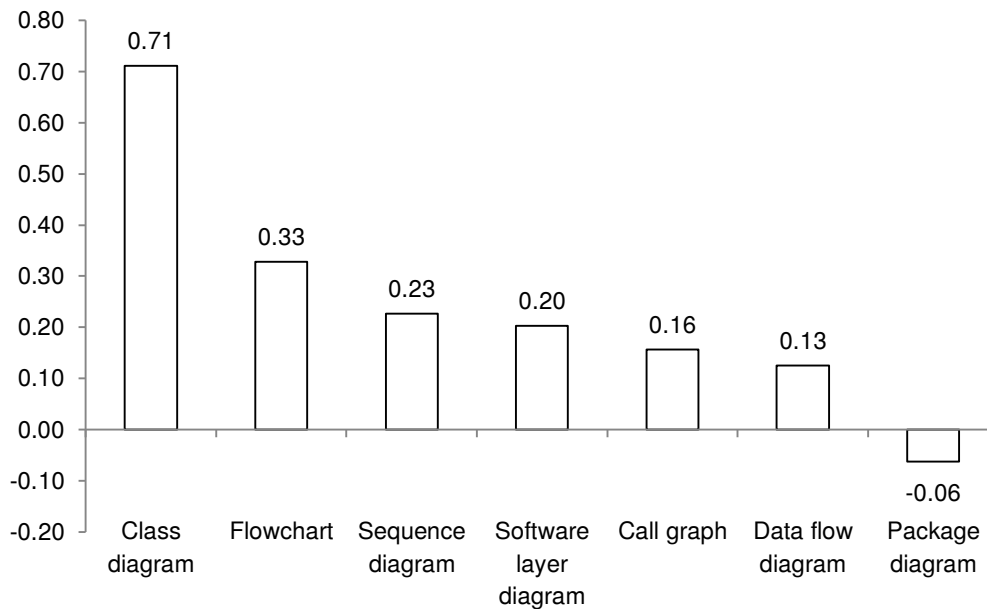
**Table 3.1:** Perceived usefulness of common visualizations ( $N = 128$ )

We computed mean usefulness scores for easier comparison.

To easily compare the overall perceived usefulness of visualizations, we represented the data in a different format. Mapping ‘useful’ to 1, ‘not useful’ to  $-1$ , ‘don’t

<sup>5</sup>[http://en.wikipedia.org/wiki/Main\\_Page](http://en.wikipedia.org/wiki/Main_Page)

<sup>6</sup>The survey form allowed for participants to give no answer. We interpret these votes as ‘don’t know/never used’.



**Figure 3.1:** Ranking of common visualizations by mean of perceived usefulness from  $-1$  ('not useful') to  $1$  ('useful').

know/never used' to  $0$ , and dividing the result by the number of participants, figure 3.1 shows how the means of common visualizations rank on a normalized scale from  $1$  to  $-1$ .

In general, class diagrams were rated most useful and package diagrams least useful. Package diagrams were the only common visualization with a negative mean. When looking at the perceived usefulness of SV by different groups, we found some interesting results. Students rate call graphs and software layer diagrams significantly lower than programmers,  $U = -1123.5, z = -2.085, p < .05, r = -.20$  and  $U = -1105.0, z = -2.201, p < .05, r = -.21$ . However, we found a significant negative correlation  $\tau = -0.16, p < .05, r_s = -.19, p < .05$  between programming experience and the perceived usefulness of class diagrams when looking at all participants. We also found a highly significant trend that the more often developers read source code, the less useful software layer diagrams become,  $J = 1990, z = -2.81$ . Similarly, we detected a significant trend for package diagrams,  $J = 1751, z = -2.004$ . Tests splitting the participants in groups according to the

There are significant differences in the perceived usefulness of several SVs between groups.

percentage of unknown source code read did not reveal any significant results or trends. Details on the statistical analysis can be found in appendix A.2.2.

### Visualizations from research

The participant were shown four experimental SVs.

We selected four visualizations from the research community and asked the participants of the study to rate their usefulness as before, based on their first impression. Again, we provided samples of each visualization, a short description of how the visualization worked, and a link to the project home page.

The chosen visualizations were:

- CodeCity (see section 2.5.2)
- Thematic software map (see section 2.5.3)
- 3d relation diagram (see section 2.4.2)
- CallStax (see section 2.3.1)

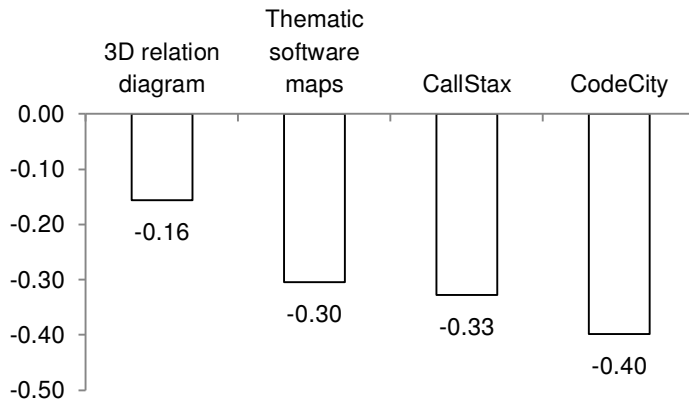
This selection is not complete.

The selection of visualizations was limited by the requirement to offer the participants a link to a project page. Also, literature research was still ongoing at the time of the survey, so not all visualizations from chapter 2 were known to us, yet. Table 3.2 gives a summary of the responses for this question and figure 3.2 an overview of the mean scores on the normalized scale.

Visualization	Useful	Not useful	Don't know/never used
3d relation diagram	27.34%	42.97%	49.69%
Thematic software map	19.53%	50.00%	30.47%
CallStax	16.41%	49.22%	34.37%
Code city	17.97%	57.81%	24.22%

**Table 3.2:** Perceived usefulness of visualizations from research





**Figure 3.2:** Ranking of research visualizations by normalized score of perceived usefulness from  $-1$  ('not useful') to  $1$  ('useful').

Overall, experimental visualizations got lower scores than the common visualizations. That is to be expected, since the meaning of the uncommon visualization might not be as clear, despite our attempt to provide a short explanation for each sample and a link to the research project home page for further information.

Common SVs were rated higher than SVs from research.

Interestingly, the 3DRD received the best score, although six participants stated (unprompted) in an open-ended question that they disliked three-dimensional representations. In our opinion, the 3DRD pursues the use of three dimensions the most rigorously among those SVs that employ three-dimensional graphics. While Code City and CallStax also render three-dimensional visualizations, they arrange elements still in one (Code City) or two (CallStax) planes. Yet, this more restrictive use of the third degree of freedom does not result in higher perceived usefulness.

We got conflicting evidence about the usefulness of a third dimension.

We did not find any significant differences in the evaluation of experimental SVs by our participants, when taking different background into consideration (occupation, programming experience, frequency of code reading, percentage of unknown code dealt with). For details refer to appendix A.2.2.

Different groups did not rate these SVs significantly differently.

### Comments

We collected qualitative feedback.

The participants were encouraged to answer an open-ended question to give comments about SVs in general. 22 users took the opportunity to tell us about their impressions.

In comments users were very vocal about three-dimensional SVs.

The most frequent comment stated that three-dimensional visualizations were less desirable than two-dimensional visualizations (six participants). Three participants felt that diagrams created by SV tools were too big to be comprehensible. Two comments mentioned that there existed good work flow visualizations that should be carried over to the software domain.

Participants questioned the usefulness of several aspects of the shown SVs.

Other comments were singular. several participants disregarded the usefulness of SVs for different reasons. Coding guidelines were deemed more important, the need for SVs did not exist, and the visualizations were too unintuitive. One commenter noted that no single diagram could capture everything. Individual comments were concerned with specific SVs, but mostly fall in the above categories. E.g., occlusion problems in CodeCity and CallStax were mentioned, the usefulness of specific metrics (LoC for entity size) questioned, and varying usefulness of common visualizations discussed.

### 3.2.3 Visualization software

SV tools are rarely used by most developers.

We asked the participants about their experience and practice working with SV tools in preparation for parallel questions regarding sketching. The participants from our survey were quite reluctant to use SV tools. More than half of them used SV tools less than once a month, almost a third of all participants never use SV tools. We set up the the questionnaire to display different questions to the 41 non-users than the 87 tool users.

### SV tool users

SV application users were asked, what SVs they usually create. Class diagrams were by far the most often used SV (81.6%), followed by flowcharts (23.8%) and sequence diagrams (20.7%). The reasons for creating SVs were mainly internal documentation (73.6%), personal use/understanding (65.5%), and public documentation (33.3%).

SV tools are primarily used to create class diagrams.

To get an impression of the SV tools used in practice, we asked users to name their most often employed SV solution. We got a wide range of answers. Frequently mentioned tools were *Visual Studio*<sup>7</sup> (11 mentions), *Doxygen*<sup>8</sup> (10), and *Eclipse*<sup>9</sup> (8). It should be noted that two participants used this opportunity, to express their (negative) feelings about the software they use:

A wide range of SV tools are used in practice.

- “bluej, leider gezwungenermaßen in der Schule”<sup>10</sup>
- “inkscape, uml sculptor (in Ermangelung besserer Software)”<sup>11</sup>

Half of the users were able to create SVs in 15 minutes or less. The users were asked, if said tool would work seamlessly and without compatibility issues regarding their code bases. Compatibility was an issue for 37% of the users. We also asked about their satisfaction with the produced SVs, which was positively answered by 79%.

The majority of tools works seamlessly with given projects and creates satisfying results.

We asked the users to categorize the tool they use most often as either automatic, semi-automatic, or manual using criteria laid out in an accompanying explanation. Participants reported 39% automatic, 25% semi-automatic, and 36% manual tools. Using this categorization we were not able to make out any significant differences in the satisfaction reported for tools of different automation-levels. However, we found significant differences for creation times

Tool automation does reduce SV creation time, but does not significantly impact result satisfaction.

<sup>7</sup><http://www.microsoft.com/visualstudio/en-us/>

<sup>8</sup><http://www.stack.nl/~dimitri/doxygen/>

<sup>9</sup><http://www.eclipse.org/>

<sup>10</sup>“compelled to use bluej at school, unfortunately”

<sup>11</sup>“inkscape, uml sculptor (for lack of better software)”

( $H(2) = 12.46, p < .05$ ) and a highly significant trend with  $J = 1687.50, z = 3.58, r = .38, p < .001$ , meaning that tools with less automation demand more of the users' time to produce diagrams (medium effect).

There are problems with generalizing the result.

However, this result must be treated carefully. StarUML ( $Mdn = '> 60 \text{ min}'$ ), a dedicated tool to design UML diagrams, performs significantly slower than the more general purpose tool Dia ( $Mdn = '5-15 \text{ min}'$ ),  $U = 4, p = .04, r = -.68$ , or sketches (see next section),  $U = 46, p < .001, r = -.30$ . We also tested the differences between the most automatic tool (Doxygen,  $Mdn = '5-30 \text{ min}'$ ) and the most manual (Dia,  $Mdn = '5-15 \text{ min}'$ ) for which we have at least five samples. Their comparison shows that we cannot transfer the general trend to any two tools from different ends of the spectrum of automation:  $U = 28.5, p = .54, ns$ . More details can be found in appendix A.2.3.

### Non-users

Time consumption is an important factor to avoid SV tools.

We asked non-users of SV tools to give us their reasons for avoiding these applications. Half of the non-users (51%) stated to have no need for SVs. Equally many found the creation process too time consuming. 39% considered the results to be unsatisfying. For more details see appendix A.2.3.

### Comments

Developers feel that automated tools cannot emphasize important aspects of a code base.

At the end of this survey part, we requested the participants to leave comments on SV tools. 27 participants provided us with feedback. Four comments were concerned with automatically generated SVs not being able to weight important aspects like a human designer would. Three developers stated that they needed SVs only as a reminder. Two participants reported, they had trouble gaining new insight from SVs. That reading source code was more efficient than using SVs was stated by two developers. Also, two users felt that SVs were not practical for large code bases. Two other participants claimed that, although very

important, run-time behavior was not represented with SV tools.

The other comments were individual remarks. Two participants reported their frustration with different SV tools, one in particular was unhappy with the slow processing speed of Doxygen. Two comments underlined the usefulness of SVs for various reasons, and one participant stated, he felt the use of pen and paper was simpler than using SV tools.

Some SV users have grievances.

### 3.2.4 Manual visualization

In this part we asked the participants about their experience with sketching. The questions in this part are similar to those in the previous one, modified to be applicable for sketching techniques. We found that more than half our participants create sketches at least once a month or more often. Only 12.5% never sketch. Again, the survey branches into one part for participants who sketch at least occasionally and those who selected 'never'.

Sketching is more commonplace than using SV tools.

#### Sketching participants

Most of the sketches contain elements from class hierarchies (66%) and class dependencies (54%). But also dataflow and function/method calls are often represented (both 41%). The primary mean to create sketches is the use of pens/pencils (99%) and paper (98%). Other tools and materials used are white boards (45%), multiple colors (36%), and erasers/sponges (32%). Most developers create sketches for personal use and understanding (85%), followed by internal documentation (44%) and project management (25%). 62.5% our participants spend 15 minutes or less on creating a sketch. For further details refer to appendix A.2.4.

Sketches visualize more than regular class diagrams and are used for personal understanding.

### Non-sketching participants

Many non-sketchers see no use in visualizations.

We asked the 16 participants, who indicated to never sketch, to give us their reasons for staying away from pen and paper and other tools. 56% reported to have no need for visualizing code. Again, time consumption was also an important aspect (44%), as was the difficulty of creating sketches (31%). More details can be found in appendix A.2.4.

### Comments

Sketches are often used for planning and building mental models.

18 of 128 participants gave feedback in the comments section of this survey part. Two of them entered only placeholders. Four users found sketching useful rather in the planning stages of a software project or code change, but not for existing code bases. Two use sketches for building mental models or discussing source code with colleagues each. Our participants liked the fast creation of sketches (two mentions), their easily disposable nature (two mentions), and their availability in the physical world (one mention). The latter however was also regarded as an disadvantage by one user, since physical documents require additional effort for digitalization, when distributed electronically.

Individual comments were often concerned with individual working conditions.

We also got a number of individual comments. One participant integrated UI graphics with his sketches. Another user noted he visualized algorithmic and functional behavior with sketches. One developer reported that sketching was redundant for him, since work processes required him to create SVs anyway. One participant claimed that pure imagination of code was faster than using sketching. Sketches were not clean enough for public documentation in the opinion of one participant. One user created sketches only for short-term overviews.

### 3.2.5 Software visualization in documentation

We wanted to briefly touch upon the topic of source code documentation, as we intended for Code Gestalt to be usable as such. 59% of the questioned developers actively seek out visualizations in order to understand unknown code. When asked, if they would consult SVs in documentation regularly when they were more familiar with a project, only 29% affirmed that.

SVs in documentation are more sought after in the initial phase of investigating source code.

#### Comments

We got 10 relevant responses in the comments section for this part of the survey. These comments stated that SVs were a good start for discovering unknown code (two participants), provided a mental map (two participants), and were more expressive than code (one participant). Other comments suggested that tutorials in general are more important, that the quality of the SV was heavily dependent on the quality of the visualized code, that the process of reading code and documentations are intertwined, that a mental model does not need to have a visual counterpart, that listings of code artifacts might be equally appropriate, and that usually neither documentation nor SVs exist (one mention each).

SVs are considered a good starting point for exploring unknown code.

## 3.3 Summary

Our findings regarding SV tools are mostly in line with related work (see section 2.6). We have gathered additional insight in what users actually do in order to create SVs, even if they do not find a suitable tool. There is a wide range from pen and paper sketches to general purpose painting programs and white board drawings to plain text files that fill this role.

We find our results in line with related work.

Users tend to prefer sketching over SV software. Creation speed is of the essence.

Programmers faced with the task of visualizing source code are more likely to fetch pen and paper than using a specialized SV software. More than half of our participants create a sketch at least once a month, while less than every third uses SV tools that often. Although it appears likely that a computer offers a significant speed advantage over hand drawings of class and interface boxes, users say they are faster using manual means. The time required to create SVs and sketches is an important factor for programmers to avoid visualization. We call this circumstance *time consumption problem*.

Automation can reduce effort for SV, if done right.

Automation features can help with regard to the time consumption problem. But this general result does not guarantee that any automation reduces time effort. E.g., StarUML, a dedicated tool to design UML diagrams, performs significantly slower than the general purpose diagram drawing tool Dia. Also, no significant differences in creation times between the manual tool Dia and the automatic tool Doxygen were found.

Result satisfaction does not depend on the level of automation offered by a tool.

Surprisingly, automatic tools created as satisfying results as manual tools, although one might expect that less automation and more customization would allow users to create SVs that better fit their vision. This encouraged us to pursue automation in Code Gestalt, but we remained cautious to design automation features in a way that they would be controllable and predictable for the user.

SV is suited for professional looking results.

In general, SV software creates the results envisioned by the users. The survey also suggests that one of the primary advantages of SV software over sketches are professional looking results. Also, users appreciate visualizations for illustration purposes in publications and documentation.

Sketching helps with understanding and communication. Class diagrams are a good common denominator.

Sketching on the other hand is more suitable for informal information exchange between co-workers and self-study of code. Many sketches seem to be loosely based on class diagrams, but is much easier to hide irrelevant details in sketching and to convey human insight by deliberately emphasizing certain aspects over others.



From the comment sections we gather that SVs help programmers with building a mental model and support initial understanding. According to qualitative feedback automatic SV tools lack understanding of a code base and might become less useful as a project grows due to increasing diagram size. We call this property of SVs *lack of insight*.

In the following three chapters we describe the development the SV tool Code Gestalt. We tackle lack of insight by designing visualizations and interaction techniques for users to harness some of the human intelligence present in the vocabulary of a code base. Our SV should reduce the time consumption problem, to become attractive for those users, who avoid SVs for that reason. When designing different aspects of the SV we will refer to the results from the survey and related work, to make informed decisions on what techniques to use.

It is also more flexible and captures human insight better.

Code Gestalt tackles lack of insight and the time consumption problem.



## Chapter 4

# Paper prototype

*“Ideas are elusive, slippery things. Best to keep a pad of paper and a pencil at your bedside, so you can stab them during the night before they get away.”*

—Earl Nightingale

In this chapter we discuss, how we initially conceived Code Gestalt and built a paper prototype with a fleshed out UI and interaction scheme. In section 4.1 we will take a look at the initial concepts that mostly manifested as rough drawings to explore various ideas of software visualization during the stages of literature research, the survey, and the interviews. The final paper prototype is presented in section 4.2. We performed a qualitative evaluation of the prototype, which is presented in section 4.3.

The first implementation of Code Gestalt was a paper prototype.

### 4.1 Design

The concepts in this section were conceived in the initial phase of the thesis while investigating related work. Looking at the problem of dealing with unknown code the idea of Code Gestalt emerged: creating a tool that would communicate the ‘big picture’ needed to understand a piece of software. The programmer should be able to get a rough understanding of the most important pieces of a code base.

We want to grant programmers better access to the ‘big picture’ of a software system.

Even if she did not understand the details, the visualization should be clear and meaningful as to identifying major and important software components.

Using concept sketches we explored several directions for Code Gestalt.

Around this initial idea we developed several concepts for visualizations, some of which were tested in paper prototypes prior to the creation of more sophisticated mock-ups and the final implementation (see chapters 5 and 6). In the following, we will discuss some of the very early designs for which we created concept drawing.

#### 4.1.1 Early concepts

The following section will discuss several preliminary designs, and how they evolved based on survey findings, user feedback, and pragmatic considerations.

##### Thematic heat map

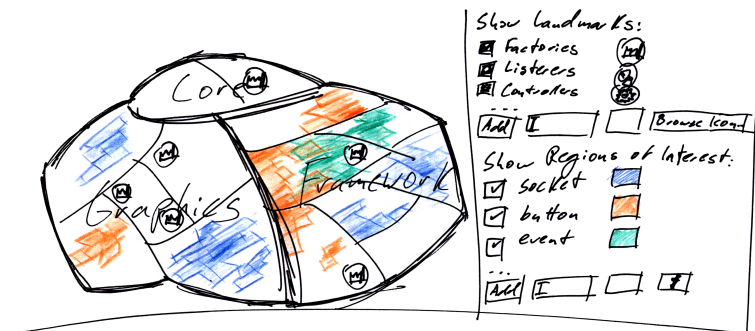
The paper of Kuhn et al. [2008] about *thematic software maps* inspired the first fleshed out concept, namely the *thematic heat map*.

The code is represented as a hierarchy of shapes.

In this concept the whole code base is represented as a hierarchical composition of nested two-dimensional shapes, representing types and namespaces/packages. So a package shape contains a sub-shape for each type or package in that package. The size of any region is determined by its LoC. A paper prototype of this visualization and its user interface is shown in figure 4.1.

The user can place landmark icons and specify terms for which to draw heat maps.

Using a filter and search interface the user is able to specify landmarks by entering a term and associating it with an icon. All types containing this term in their identifier are then identified by that icon on the map. Moreover, the user can specify regions of interest, i.e., a heat map based on a term and an assigned color. All types that use the given term in their source code are highlighted using an importance weight derived from tag synthesis.



**Figure 4.1:** One of the earliest concepts for Code Gestalt used heat maps as visualization.

**Discussion:** This early concept would not have allowed the user to actually modify the structure of the diagram. Instead, the user would have been restricted to specify one or more overlays that allowed for visual searches in the code base (very much like CodeMap, see section 2.5.3). We deemed the concept to be too limiting. Nevertheless, both the idea of highlighting regions of interest and the concept of creating landmarks for better orientation ended up as core features in the final Code Gestalt implementation.

The concept did not allow for much user editing, but core ideas made it into the final implementation.

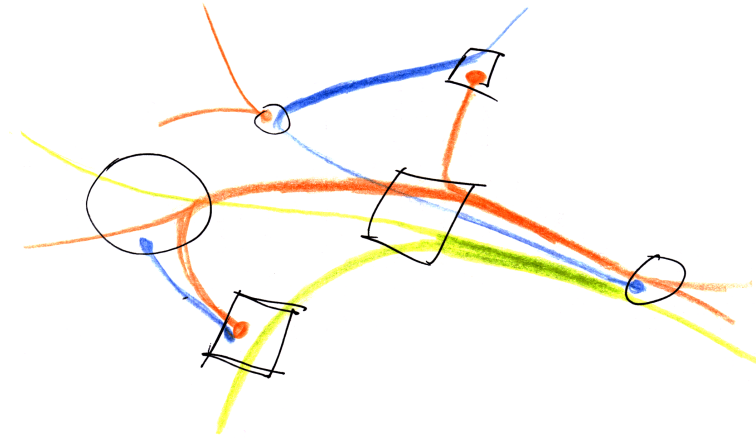
### Data trace

The concept of data flow was revisited several times during the design phases of Code Gestalt, but was ultimately dropped, as visualizing the flow of data between objects resulted in too much visual cluttering, when displaying other relations. However, our online survey suggested that data flow is an important property for users to visualize, so we will briefly discuss our concepts for this aspect of SV.

We were unsuccessful in developing clear SV for data flow.

The *data traces* in figure 4.2 were the first such concept. It traces objects over several method calls and visualizes them as colored lines between shapes that represent types. The idea is to illustrate how a data structure is passed from type to type, where it may be known under different parameter names. The thickness of the lines indicates the amount of *traffic* between types, i.e., the number of objects sent from one type to the next and their memory footprint.

Traffic between types is illustrated as lines of varying thickness.



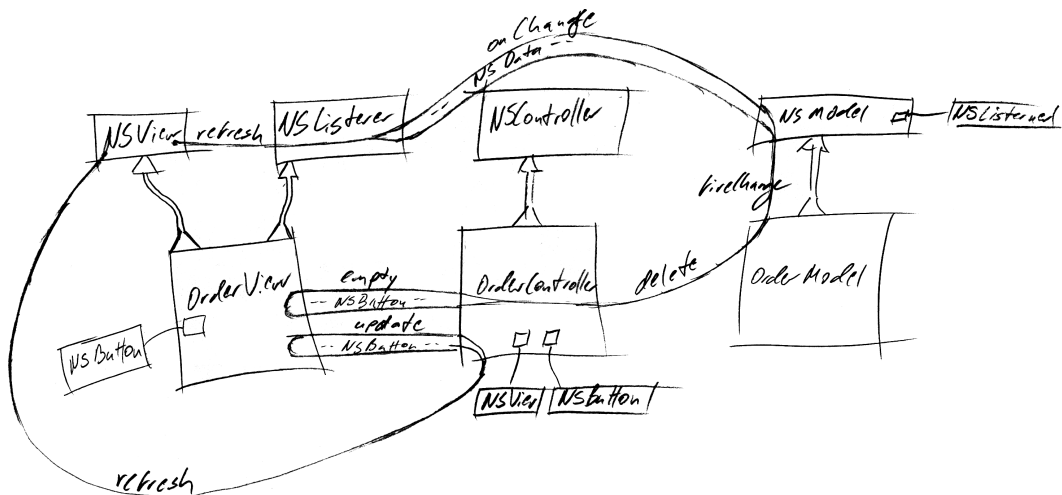
**Figure 4.2:** A dropped concept for tracing the flow of objects.

The concept is hard to implement and produces much clutter.

**Discussion:** This concept was pursued further, when the online survey revealed the importance of data flow for potential users (see sections 3.2.2 and A.2.4). However, the technical implementation of this concept would have been quite challenging. E.g., branching and merging of traces would have required some understanding of cloning and integration of objects by the SV tool. Ultimately, the concept was abandoned, since other relations were deemed more important, and the data traces introduced another level of visual clutter to the diagrams.

#### 4.1.2 Designs in light of the online survey

After reviewing the survey results we iterated on our previous designs and developed new concepts on our path toward a UI to be tested with users.



**Figure 4.3:** This concept visualizes the composition of framework objects and their communication.

### Framework flow

This concept is based on the idea of message flow in application frameworks. We derived this design from a discussion with a student after conducting an interview along the lines of our survey. The rough paper prototype in figure 4.3 shows, how messages are passed around in a fictive shopping application. Framework messages (e.g., button clicks) and data become relations that connect the elements of a static type hierarchy. Also, the composition of framework objects is shown. Basic framework components are shown as labels attached with ‘plugs’ (small squares) to the custom user classes that use them.

Framework flow visualizes event routing and composition in UI frameworks.

**Discussion:** Visualizing the composition and message routing of modern application frameworks is a logical step in terms of static hierarchy visualizations such as class diagrams and call graphs. The SV tool, however, becomes usable only for those frameworks and framework versions it has been adopted for. Like the data trace concept, this requires a deep level of understanding not only of the programming language, but the framework as well. After some consideration this concept was put off, since it was another SV of the static structure of the source code, with-

The concept only works for a very limited number of code bases and is very centered on code structure.

out much emphasis of the desired properties (see sections 2.7 and 3.3). We wanted Code Gestalt's capabilities to go beyond these structural relations and find new ways to promote program understanding on a more conceptual and semantic level.

### Structured context diagram

The structured context diagram incorporates elements from Nassi-Shneiderman diagrams.

This SV was a low-level approach designed to maintain as much information as possible from the actual source code. It is an SV based on class diagrams. We looked at the group of participants, who claimed that reading code, good naming conventions, and syntax highlighting were their most favorite ways to learn unknown code. It draws inspiration from Nassi-Shneiderman diagrams [Yoder and Schrag, 1978], as the code inside a method is structured to indicate the occurrence of alternatives and loops.

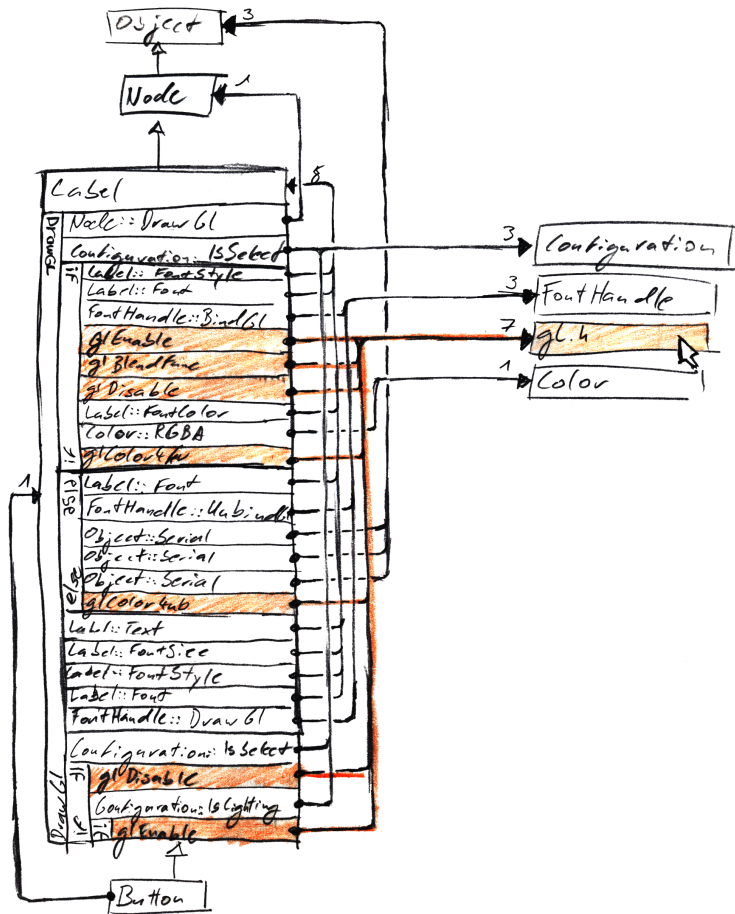
The diagram is created automatically, centered around the currently edited method.

In this concept, the diagram is created automatically in a view of the IDE, centered at the method the developer is currently working with (see figure 4.4 for a case study performed with a real code base of a 3d engine). The type hierarchy for the current class is shown vertically. The current method is shown inside the box of the current class. We structured its code by vertical bars to group local functions, alternatives and loops. Horizontally, the call hierarchy for the current method is shown. In our example, `DrawGl` is only called by `Button`, a child class of `Label`. If calls from other classes outside the class hierarchy of `Label` had been used, they were to appear in a column to the left of the type hierarchy, just like called methods are shown in a column to the right. The numbers labeling the call relations indicate the fan-in of call relations at the target. Although automatically generated, the user would be able to interact with the visualization by highlighting call and inheritance relations by hovering.

Studies with practical code bases led to much clutter.

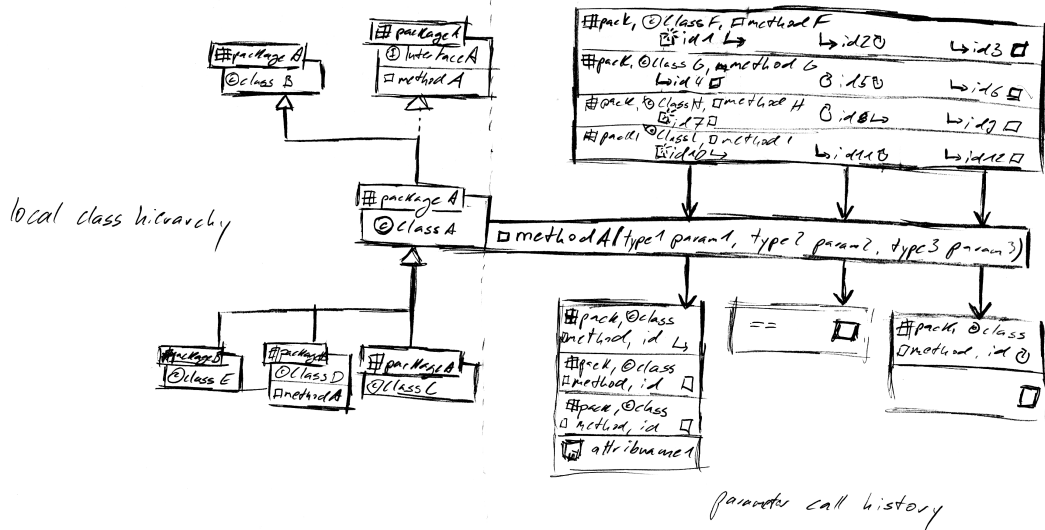
**Discussion:** Initially, the concept promised to provide the developer with an informative context of what she was currently working with, since we managed to capture the most important properties of the code in a well organized format.





**Figure 4.4:** The diagram is centered at the current method `DrawGl` from the class `Node`. Incoming calls are visualized on the left, outgoing calls on the right of the method. Vertical relations (top and bottom of the type) make up the type hierarchy. The calls to `gl.h` are highlighted. This case study of the structured context diagram convinced us to not pursue it further.

Therefore we applied the principles of this technique to the case study, shown in figure 4.4, that illustrates problems of this visualization. The call hierarchy could not be displayed without a lot of clutter. Also, the visualization did not offer much potential for user customization, abstraction, and other desired features.



**Figure 4.5:** Concept drawing for the local context view. On the left the local type hierarchy and on the right the local call hierarchy are shown. The call hierarchy tracks individual parameters across calling and uses symbols to indicate the use of them before and after the method call.

We kept some layout ideas in Code Gestalt.

However, some of the design elements like vertical layout for type and horizontal layout for call hierarchies were kept and developed further. We came back to this concept, when designing the local context view (see section 4.1.2).

### Local context view

This SV is an iteration of the structured context view.

Our last noteworthy concept before the final paper prototype was the *local context* view. This concept was an iteration of the structured context design, putting more emphasis on data flow and hiding implementation details on the source code level. The idea behind the local context view is to allow developers to track parameters across multiple method calls. Again, the view is centered on the method currently edited in the IDE. The view is divided vertically into two sections (see figure 4.5).

We show the type hierarchy on the left and a call hierarchy on the right.

On the left we have a local type hierarchy, similar to the one in the structure context diagram. On the right a special call hierarchy is displayed. At the top there is a list of all methods calling the currently edited method. For each

parameter we are given the local name of the passed variable. Two icons illustrate, how the calling method obtained the local variable and how it will treat it after the current method returns to it.

- *Carriage return symbol in front of the local variable name:* indicates that it is a parameter of the calling method (i.e. passed down from another method)
- *Carriage return symbol behind the local variable name:* indicates that the calling method will pass the same variable to another method after the current method has returned to it
- *Looped arrow:* a special case of the carriage return for recursions
- *Square:* indicates that the calling method will not pass the variable along to another method
- *Square with a star:* indicates that the variable was declared locally in the calling method

Icons give details on the role of parameters.

Similarly, a list for each parameter illustrates, how they are passed by the curring method to called methods, and what these called methods do with the parameter.

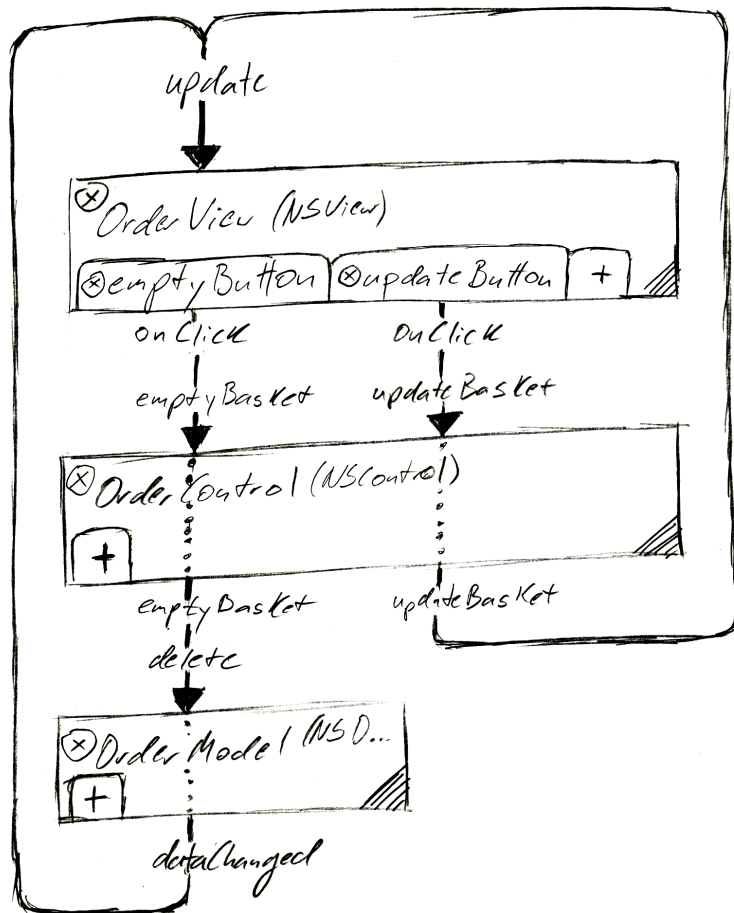
**Discussion:** This concept was another instance of an inflexible visualization, focused only on structural features of the code base. Although it allows developers to gain more information than from a regular call graph, it does not allow them to go beyond very basic syntactical information. This visualization, however, introduces the idea of using icons for conveying additional information, an idea that we revisited in the implementation of the Eclipse plugin, when we added icons to all Java entities to present more information.

The view allows for little customization. We kept the idea of augmenting the diagram with icons.

## Diagram widgets

Our diagram type boxes should provide a UI elements for interaction.

When we settled for class diagrams as baseline visualization, we needed some sort of user interface to manipulate diagram elements (or widgets). We designed some UI elements to directly work with types, methods and relations. A *tabbed browsing* interface as in figure 4.6 was based on the idea that users would work with the diagram over a longer period of time and add or remove members on the fly in analogy to the management of tabs in web browsers like *Mozilla Firefox*<sup>1</sup>. Note that the concept drawing follows the idea of framework specific relations from section 4.1.2.



**Figure 4.6:** Members are added to types as tabs.

<sup>1</sup><http://www.mozilla.com/firefox/>

Since the horizontal layout of tabs requires a lot of screen space, we redesigned the interface to add and remove members in a vertical layout, converging to a more familiar representation of types in class diagrams and Relo. Figure 4.7 shows this layout. The concept was drawn before we decided to use a vertical layout for inheritance and a horizontal layout for call relations.

Horizontal layout of members is more practical than vertical tabs.

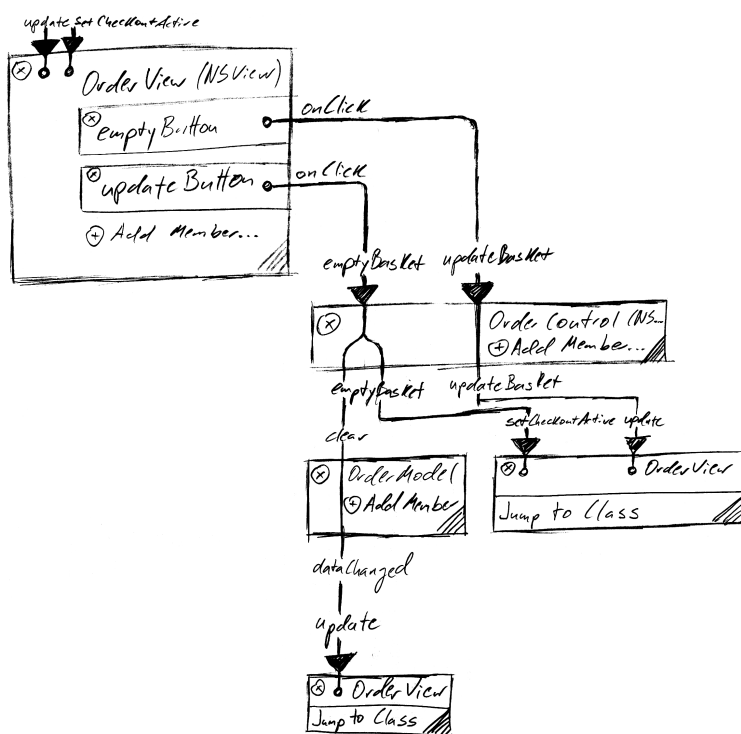


Figure 4.7: Members are added to types as list entries.

Users can directly manipulate these widgets, by using common UI elements, such as close buttons and the resize handles. Widgets can be dragged around and rearranged. Relations are added and removed by selecting elements and clicking context sensitive expansion controls (see section 4.2.3).

The widgets allow for much direct manipulation.

Most elements from this design are present in the final implementation.

**Discussion:** We kept most of the design and interaction techniques from these prototypes in the paper prototype. As they were immediately understood by all participants, the design is very close to what we later realized in Code Gestalt.

### 4.1.3 Designing the paper prototype

Survey results lead to using class diagrams as a baseline for Code Gestalt's visualization.

The paper prototype was geared toward closing existing usability gaps and was rather conservative in terms of SV. The results from the survey indicate that the visualization metaphor should revolve around concepts like classes, methods, and data flow, as well as inheritance, dependency, and call relations. We decided to use class diagrams as a baseline for the visualization, since they are built around many of the above concepts. As shown in the previous section, incorporating data flow proved to be difficult and was dropped in favor of call relations.

We enhance the UI of Relo to create a simple and fast editor for partial class diagrams.

After exploring different approaches (see section 4.1.2), we came back to an interface and diagram design that is reminiscent of Relo (see section 2.4.1). Our design is based on partial class diagrams created through interaction with a context sensitive interface. This way we intend to address the time consumption problem and promote simplicity. A user adds new elements to a class diagram by simply expanding it along relations of the elements in the diagram to new elements not yet visible. Our prototype adopts the notion of class diagrams as a familiar baseline for our visualization that is then augmented with information that goes beyond structural properties.

Remove the drawing of packages and instead allow for user grouping.

Since the user survey shows that package diagrams are least favored by users among the common visualizations, our prototype does not include the package container boxes drawn by Relo. Instead we allow users to group and color classes freely by criteria of their choice. In order to assist the user in finding diagram entities for grouping, the prototype provides visual search and filter functions..

From the related work we know that the ability to search SVs is important, but missing in many implementations. We think is an important reason for lack of insight. Relo has no built in search option and does not support the IDE search functionality. Eclipse's find-feature is tailored towards text and presents its results in a sortable list. Results from a diagram, however, are more naturally visualized spatially, since line numbers (or coordinates) in a list are not intuitively suited to quickly locate an occurrence. Therefore, we designed a new search feature from scratch, present in the prototype as sidebar. This feature allows the user to specify filters in different scopes (like method names, comments, or parameter names) and to combine several of these filters.

Search is implemented using filters, results are displayed within the diagram.

One of the few disadvantages of sketches in relation to SVs is their physical nature that makes them inconvenient for electronic communication. Scanning sketches or photographing white board drawings creates overhead that reduces or cancels the advantage in speed during creation. To explore this aspect, our prototype had one use case dedicated to the integration of persistent diagram documents in the structure of an IDE project. We envision that such a tight integration of diagrams in the project and therefore its version control system would almost eliminate any overhead in sharing and synchronizing the visualizations among co-workers.

Sketches cannot be mailed easily and create overhead. Therefore we integrate our diagrams tightly into the IDE project structure.

## 4.2 Implementation

The prototype was realized using printed screen shots of the Eclipse IDE with two empty editor windows equally sharing the majority of the screen. In these editor windows, source code and diagrams were drawn by hand to illustrate three use cases. Menus were realized using sticky-notes, changing diagrams by swapping smaller pieces depicting one editor window.

Drawings on printed screen shots and sticky-notes make up the paper prototype.

We considered a total of twelve use cases grouped in three major tasks.

We following considered the following use cases:

1. The user has to identify and open diagrams from the project view.
2. The user creates a new diagram from selected source code.
3. The user edits an open diagram.
  - (a) The user changes the diagram by
    - i. adding a parent type of a type.
    - ii. adding a calling method (and the corresponding type) of a type.
    - iii. adding a called method (and the corresponding type) of a method.
    - iv. deleting a type.
    - v. adding a child type (and overwritten methods) of a type.
  - (b) The user creates filters to hide irrelevant elements/groups.
  - (c) The user creates filters to hide irrelevant relations.
  - (d) The user edits a filter.
  - (e) The user groups and highlights elements.
  - (f) The user edits the appearance of an element/a group.

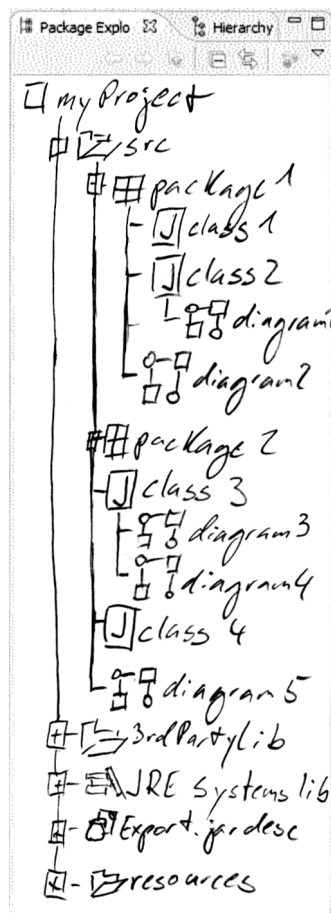
For each use case the users had to answer questions before given the actual task, to identify interactive elements and to describe their affordances. After describing how they would perform the task, the participants were asked open-ended questions on how intuitive the UI was and what problems they had with the interaction both conceptually and practically.



### 4.2.1 Project integration

Figure 4.8 shows, how we presented the integration of diagrams with the IDE project structure. A diagram can be a child of any package or type, indicating that the diagram visualizes an aspect of that package or type. This mock-up was used to test use case #1.

Diagrams integrate with the IDE project.



**Figure 4.8:** Eclipse package explorer displaying the contents of a Java project with Code Gestalt diagrams

### 4.2.2 Creation of a diagram

Diagrams are created using the default mechanisms in Eclipse.

We designed the prototype to allow for multiple ways to create a new Code Gestalt diagram from a given method. One solution is depicted in figure 4.9. All solutions were based on putting a Code Gestalt command in Eclipse's 'Show in' sub-menu, accessible from various locations of the IDE. This decision was made for consistency reasons and had to be revised, when the evaluation demonstrated a lack of visibility (see section 4.3).

### 4.2.3 Expanding a diagram

We use a consistent interaction scheme for editing interactions.

The paper prototype contained samples to test use cases 3.(a)i.–3.(a)v. All, but 3.(a)iv (delete a type), are based on the same interaction-scheme:

1. The user selects a code artifact (type or member).
2. Code Gestalt searches for related code artifacts in the source code, displaying those not already contained in the diagram as semi-transparent *live previews*.
3. When the user clicks any preview it is added to the diagram and made persistent. Previews ignored by the user disappear as the selection changes.

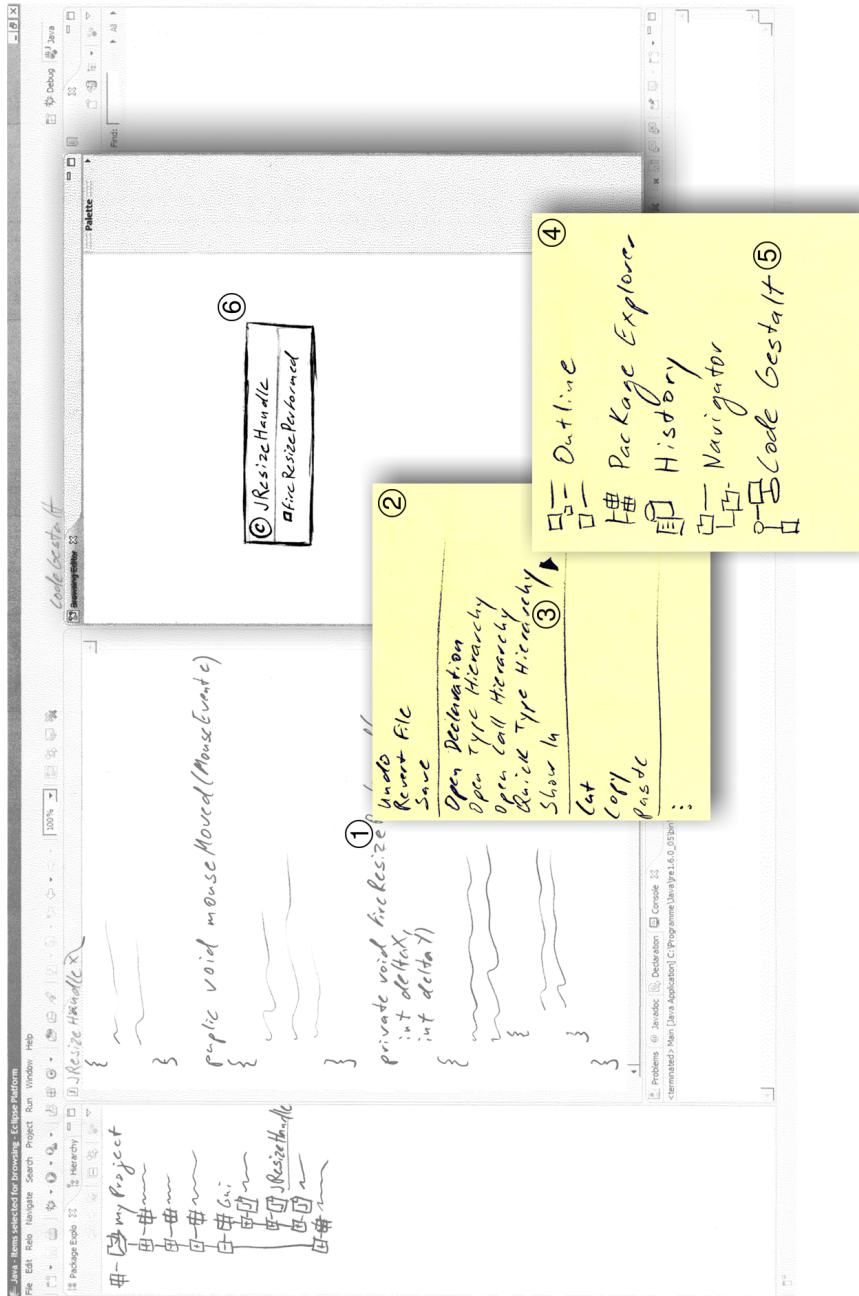
Definition:  
*Live Preview*

#### **LIVE PREVIEW:**

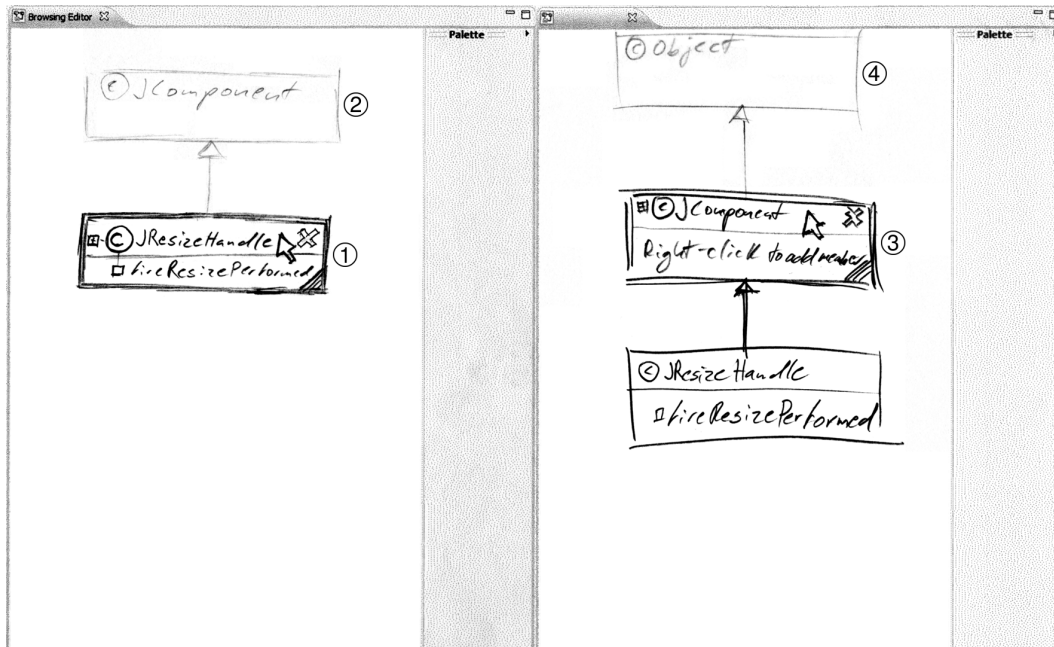
*Live previews* show a user the results of a possible interaction, before it is performed and any actual changes are applied. When the user decides to take no action, the preview is removed without effect and the edited document remains unchanged.

Relations are shown using live previews.

Related elements are identified by having either a call or inheritance relation to the current selection. The interaction for an expansion is exemplified in figure 4.10, which shows how the user may explore a type hierarchy.



**Figure 4-9:** Creation of a Code Gestalt digram from file context menu. The user selects a method (1), opens the context menu (2) and selects the 'Show in' entry. From the sub-menu (4) the user picks Code Gestalt (5) and a new Code Gestalt editor window with the selected method and type body is opened (6).



**Figure 4.10:** Expanding a Code Gestalt using context sensitive previews. The user selects a type (1) and the diagram shows semi-transparent live previews of related code artifacts, e.g. a parent type (2). By clicking the preview, it becomes persistent and added to the diagram. Selecting the new element (3) triggers Code Gestalt to display previews in the new context (4).

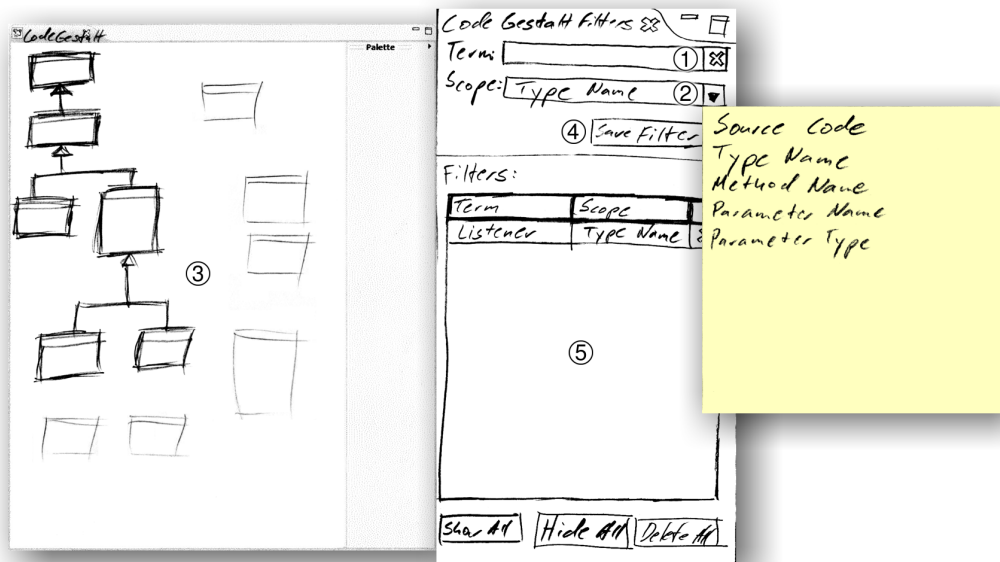
#### 4.2.4 Search a diagram

Users search the diagram visually by specifying filters.

For use cases 3.(b)–3.(d) we designed a visual filter interface. We allow the user to search for terms in the source code and define the scope of the search. E.g., if a user wants to find controllers in a code base dealing with user accounts, she could first define a filter for the search term ‘controller’ with scope ‘Type Name’ and then a second filter ‘account’ with scope ‘Source Code’.

We updated the filter interface during testing.

The second iteration of this interface is shown in figure 4.11. The difference between the first and second iteration is the replacement of an eye-symbol next to each entry with a list view allowing for multi-selection.

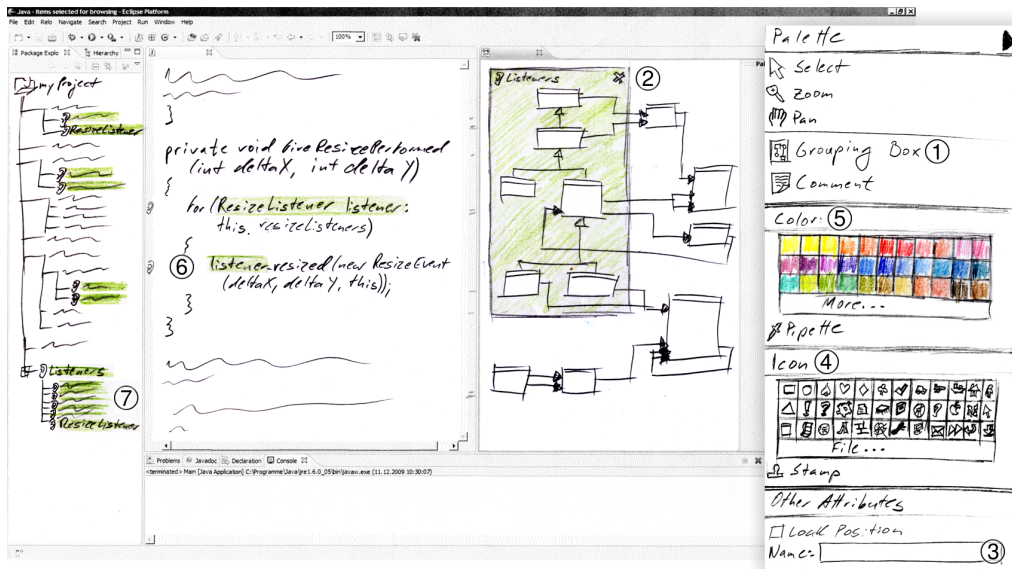


**Figure 4.11:** Searching a diagram using the filter side bar. The user enters a search term (1) and selects a search scope (2). The search results are shown in the diagram by highlighting corresponding code artifacts (3). The user may save the filter (4) and manage saved filters in a list (5).

#### 4.2.5 Grouping and tagging

We designed a tool palette for use cases 3.(e) and 3.(f). The palette contains several tools and an context sensitive properties panel that is only shown, when editing a group (this mode is shown in figure 4.12). Using the grouping tool (1), users may create groups of types by dragging a box in the diagram (2). In order to make the group a recognizable landmark, we allowed for customization using the properties panel (3–5). This customization is not only limited to the diagram editor, but causes custom color highlighting in the Java editor (6) and package explorer (7). There, we also create a virtual folder for the elements of the group.

Groups are created and edited using a palette.



**Figure 4.12:** Creating groups and landmarks. The user selects the grouping tool (1) and drags a box around the elements of the diagram, she wants to group (2). The group can be assigned a name (3), icon (4) and color (5). The icon and color will be used to highlight group elements in the code editor (6) and project browser (7), where a virtual folder for the group members is created.

### 4.3 Evaluation

A small user study to check for design flaws and gather user input.

Five student assistants not involved with the project (all male) were interviewed. They were shown the screen shots depicting the different stages of the IDE and asked to perform the twelve tasks described in the use cases from section 4.2. The investigator changed the interface based on the actions the testers choose (*Wizard of Oz* technique [Kelley, 1983]). You can find the instructor guide in appendix B. All participants were encouraged to provide qualitative feedback on the prototype.

### 4.3.1 Test results

#### Project integration

For the first use case, users had to identify and assign diagrams in the project. The diagrams were integrated into the hierarchical project structure and could be mounted into each package or type. The intention was that root-level diagrams illustrated general concepts, while diagrams at the leave-level only showed type-specific relations. Although all participants had no problems with identifying the diagrams in the project explorer, two had another understanding of what the scope of the diagrams at different levels in the project hierarchy meant. In the open-ended questions the idea was proposed to collect all diagrams in one virtual project folder and to make the individual diagrams accessible by each class via the classes context menu, if that diagram included the respective class. However, most users preferred to freely place diagrams in the project tree.

The hierarchy level of a diagram raised different expectations of its scope.

#### Creating a diagram

The second use case gave an interesting insight into the affordances of IDEs in general (or at least those similar to Eclipse). Although the task description made no suggestion as how to create a new diagram (a total of five different ways were prepared in the prototype), four of the five users chose to use *drag-and-drop* from a piece of source code or an item from the project explorer to the Code Gestalt editor. Only one user started the diagram using a context-menu after wanting to know the keyboard shortcut, which was not implemented in the prototype. Another user suggested a button in the IDE toolbar to start a new diagram from selected code, which was also not present in the paper prototype.

Starting a diagram was predominantly accomplished using a *drag-and-drop* metaphor.

### Editing a diagram

The input (and live-preview) focus should follow each expansion, thus making in-depth-exploration easier than breadth-first-expansion.

Live previews were intuitive for users. An auto-scroll feature was suggested to follow current focus.

Filtering with several scopes for searching were intuitive, but multi-selection lists and layer-metaphor for filter maintenance did cause confusion.

The third use case is divided into several smaller tasks. Only use case 3.(a) had a detailed full interface mock-up for 3.(a)i. The other sub-use cases were demonstrated using a partial view showing only changes to the diagram widgets, ignoring the rest of the IDE UI. The interaction was well conceived by all but one participant, who expected different results based on his conceptual problem with the static nature of the SVs. The participants expressed a preference for an interface allowing for one-click selection in all cases and therefore making exploration (depth-first navigation) more fluid, but expansion of the diagram after a breadth-first scheme a bit more click-intensive (since the focus creating the extension preview would change with each expansion step and thus changing the selection of available extensions).

The live preview was so effective as to allowing two of the five participants to identify a minor inconsistency in one of the paper prototype screens. The way another screen was accidentally drawn, triggered one participant to wish for an auto-scroll-feature panning the editor canvas, so that all newly available expansion options were visible when changing the focus and not cut-off by the canvas border.

For use cases 3.(b), (c) and (d) the one-page screen shot of the paper prototype was supplemented with a larger hand drawn sidebar depicting the filter UI. All users were immediately able to filter a given diagram using different scopes and combining two filters to refine their search results. Disabling and enabling existing filters from a list however presented a challenge to the testers. Originally, each filter had an eye symbol assigned to its row. This 'Photoshop layer'-metaphor did not work well, since disabling the filter (closing the eye) would display more information in the diagram, not less as suggested by the metaphor. Using a multi-selection list without additional UI elements, where selected elements indicated active filters, was also a problem due to the low affordance of the list for multi-selection and poor visibility. These difficulties were one reason for us to design a much simpler filtering interface and metaphor for the next prototypes (see section 5.1).



The filtering of the diagram was followed by the use case to group the so found elements, thus making the result persistent and highlighting it permanently in the diagram. Presented with a hand drawn palette of tools available in the editor, three of the participants started looking for a tool allowing them to group the filter results with a single click. The grouping tool from the palette had such an affordance, but we intended it to work differently. After explaining to the testers that the tool was meant to draw a grouping box manually in the diagram to allow for groups other than those generated from filtering, all participants created the group without problems.

After filtering, users want a one-click solution for creating a group from the results.

To highlight and annotate the newly created group the users were asked to assign it a name, color, and icon. All users were able to perform this task using the palette tools. Some users requested to change the order of the tools in the palette to better represent the importance of each property (in descending order: name, color, and icon).

Changing color and other attributes worked well.

After these tasks the testers were presented with a rendering of how the other parts of the IDE changed in response to the grouping and tagging. The identifiers of group members were highlighted in the color of that group in the code editor and assigned the corresponding icon on the left hand side of the editor window. In the project explorer group members were also highlighted using the chosen color and their generic icon swapped with the user selected icon. Additionally, another virtual folder with the group name appeared in the project, containing links to all group members. Two participants voted for keeping the syntax highlighting optional and three participants wanted the icons in the project explorer removed from the files in the package browser. We were also made aware that the icons on the left side of the editor window might overlap with the Eclipse icons indicating errors and warnings.

Users wanted to keep syntax and file highlighting using the group's color and icon optional.

### 4.3.2 Further feedback and observations

A fundamental problem with the SVs conception occurred with one participant. Two participants were enthusiastic about prototype realization.

User suggestions made us rethink filtering and grouping.

Two students expressed an explicit interest in having a tool like the one portrayed in the paper prototype for their favorite IDE. One of the participants had conceptual problems with the limitations of the SVs to the static analysis of the program structure. Therefore, this participant had problems understanding the tasks given by the use cases during the test. We chose not to change the conceptual basis of the prototype on his feedback, since the user survey clearly demonstrated an overwhelming majority of users prefer class diagrams over any other visualization.

The testers suggested a number of interesting ideas to change and augment Code Gestalt. Most of them were concerned with the filtering and grouping mechanism:

- Provide debugging features by displaying/manipulating breakpoints directly within the diagram and visualize the run of a program in the diagram.
- Save the filters created from searches with the project file for future reference by the programmer.
- Allow for optional display of parameter types along call edges
- Provide an interface to collapse and expand groups.
- Allow for auto-grouping by defining a filter once and let the system automatically add new classes matching the specified criteria to the corresponding group afterwards.

### 4.3.3 Impact on next prototype

We refined the prototype by simplifying the filter interaction.

We reacted to the feedback and the results from the user tests by simplifying the filtering and grouping interactions. We introduce *thematic relations* and the *tag overlay*. These concepts are presented in the following chapter and were implemented in a Silverlight prototype and the final Eclipse plug-in.

## Chapter 5

# Silverlight prototype

*“We don’t trust it  
until we can see it and feel it.”*

—Win Ng

In light of the paper prototype evaluation presented in the last chapter, we decided to redesign the visualization and interaction concerning filters and groups. To this end we introduce the concept of *thematic relations* and the *tag overlay*, which are fundamental contributions of Code Gestalt. We evaluated these ideas using a low-fidelity online software prototype<sup>1</sup> built with the application framework *Silverlight*<sup>2</sup>.

We redesigned the filtering and grouping interaction.

We introduce the design rationale behind the tag overlay and thematic overlay in section 5.1. The implementation of a proof of concept prototype with the *SketchFlow*<sup>3</sup> prototyping framework in Silverlight is discussed in section 5.2. We present an evaluation of this prototype in section 5.3.

We built and evaluated a software prototype.

---

<sup>1</sup><http://www.startrek-journey.de/webcontent/prototype/index.html>

<sup>2</sup><http://silverlight.net/>

<sup>3</sup>[http://www.microsoft.com/expression/products/Sketchflow\\_Overview.aspx](http://www.microsoft.com/expression/products/Sketchflow_Overview.aspx)

## 5.1 Design

Four reasons led to the redesign of filtering and grouping.

Introducing filters and thus augmenting the visualization of a class diagram with information derived from the code vocabulary was a first step to harness the human intelligence present in the naming conventions of source code. There were however four reasons to redesign the filtering and grouping features of the paper prototype:

- User confusion about the filter interface
- The dichotomy of filters and groups
- Users requested to save filters and to automatically add new code entities to existing filters
- Filtering solely relies on the user to understand the code base and does not help in code exploration

We introduce the tag overlay and thematic relations.

We attempt to solve these issues by introducing two intertwined visualizations, namely the *thematic relation* and the *tag overlay*. Using tags to filter the diagram and to group types, unifies the models of filters and groups, thus allowing types to automatically update their group membership based on their source code without surprising the user. Finally, we eliminate the *vocabulary problem*, by making the search space explicit and allow for exploration of the vocabulary of the source code.

Definition:  
*Vocabulary Problem*

### **VOCABULARY PROBLEM:**

The *vocabulary problem* states that users, designers, and systems often use different words to identify a thing [Furnas et al., 1987]. We encounter this problem, when searching source code for code entities. E.g., the generic data structure known as `HashMap` in *Java* is called `Dictionary` in the *.NET* framework and `NSMutableDictionary` in *Cocoa*. Similarly, programmers might call a concrete observer from the *observer pattern* a number of different things (and name the class accordingly): listener, observer, controller, server, receiver, etc.

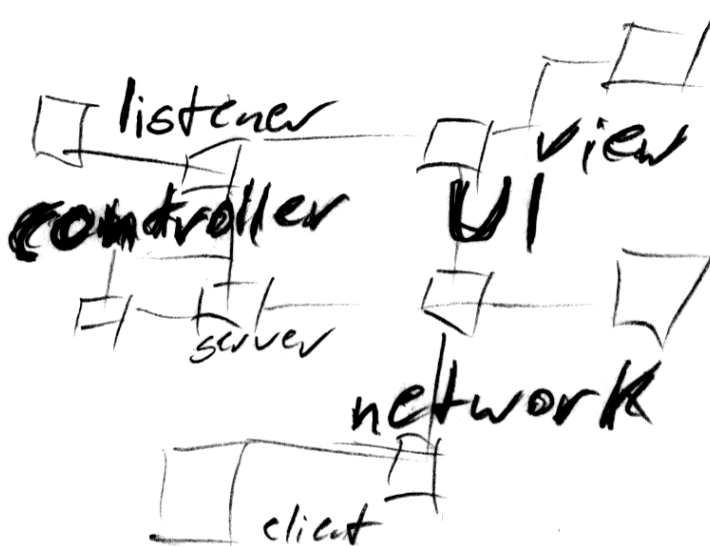
Where visualizations like the thematic software map prescribe a rigid layout of the diagram based on vocabulary metrics and allow for the display of structural visualizations afterwards, we want to start with an editable class diagram and display the vocabulary information as augmentation. While we lose some expressiveness and precision with regard to the metric, we gain a lot of flexibility and customizability for the created diagrams.

In Code Gestalt metrics augment the SV, but do not prescribe the layout.

### 5.1.1 Tag overlay

The tag overlay is an additional layer on top of an existing class diagram that displays a tag cloud. The sketch in figure 5.1 shows, how we originally envisioned it. Each tag has three visible attributes: a term (single word), a weight (expressed in font size), and a location.

The tag overlay is a tag cloud for class diagram.



**Figure 5.1:** Our initial concept for the tag overlay. The layer in front of the class diagram displays a tag cloud to indicate where in the diagram certain themes and concepts are implemented.

Definition:  
*Tags and Terms*

#### TAGS AND TERMS:

We differentiate between a *tag* and its *term*. While a term is a simple lower case string (ideally, a meaningful word, acronym, or symbol), the tag is a tuple of a term and a list of classes that use that term in their source code. The latter information is required to calculate the weight and location of the tag. We visualize the weight (based on a simple frequency analysis) of tags using different font-sizes.

Terms are extracted from identifiers and positioned near those types using them.

The terms from the tag overlay are extracted from the identifiers used in the source code of all types present in the diagram. To determine weight and position we calculate tag clouds for each individual type. Each tag is assigned the mean weight of its term in each class that uses it. Similarly, the position is the center of gravity between these classes, where each class 'pulls' at the tag with a strength proportional to the individual weight in the tag cloud of that class.

#### Visualization

The position of tags conveys where in the diagram types implement themes and concepts.

We visually display the vocabulary of the source code as overlay to the existing diagram. Since the position of each tag is based on the frequency of its term for the visualized classes, the position of each tag has an intuitive meaning. Ideally, terms with similar semantics are located close to each other and close to types that deal with these concepts. This way, the user can get a quick overview, what regions of the diagram cover which functionality.

Tags might be positioned away from related types.

However, there is no guarantee that classes with similar vocabulary are located close to each other, since the underlying class diagram is organized along structural relations (calls, inheritance). One might suspect that classes using the same term end up at different ends of the diagram and thus most tags would be positioned close to the center of the diagram, nowhere near the classes that use their terms. Kuhn et al. [2008] however found indications that classes with similar vocabulary have indeed structural connections, if the developer uses reasonable naming conventions for identifiers (a prerequisite for Code Gestalt). So we

are confident that in most cases tags are positioned close to their referencing types, since the types are likely part of a cluster.

Our proof of concept (in section 5.3) strengthens the intuition that problematic situations like the one described above are rare. We reserved the possibility to use clustering techniques to ‘split’ tags if necessary.

Another anticipated problem is the overlap of tags with similar meaning and use. In the Silverlight prototype we had no trouble dealing with it, since we created the tag overlay manually and could resolve overlaps easily. Still, we would have to deal with this problem in an actual application. Our solution is a sweep line algorithm that is presented in section 6.2.5.

Our initial visualization used two different colors for tags, based on the distinction of *introduced* terms (orange) and terms reused from other sources (blue). Introduced terms are those that can not be found in the source code the current type is structurally dependent on (e.g. term introduced by the framework or a parent type). This distinction was adopted from Cultivate’s term dependency and term cloud views (see section 2.2.2).

Tags were rarely positioned inappropriately in the prototype.

We resolved overlaps between tags manually.

Orange marks introduced terms, blue reused terms.

## Interaction

The tag overlay is a semi-static visualization, meaning its layout cannot be edited directly, e.g., by dragging and dropping tags. This would compromise the usefulness of tag positions as metric for code vocabulary similarity. Instead, the user may rearrange the underlying class diagram, thus shifting the center of gravity for related tags and hence indirectly rearrange them.

The position of tags is computed from the position of types.

The primary interaction technique is a highlighting function, similar to the filters in the paper prototype. The user may click any tag in order to display a highlight of classes that use its term in their source code. The more intense the highlight, the more frequent the term is in the source code of the class. With this heat map the user may quickly

The user can quickly create heat maps from terms...

identify the classes that implement key features for a given concept (compare the thematic heat map concept in section 4.1.1). E.g., clicking a tag with the term 'undo' will probably highlight those classes that allow an application to revert user actions.

... and types to find diagram areas of interest.

Likewise, the user may select classes to highlight tags. In this case, the intensity of the highlight represents the term frequency according to the 'local' tag cloud of the selected class. We expect this feature to help a user identify relations to other parts of the diagram, not made explicit by structural relations in the source code of the project (e.g., relations that exist in the framework and are not in the scope of the diagram).

The tag overlay is a highly transient interactive medium.

However, the position of a highlighted tag does not convey much information about a relation between the selected class and another. Also, the highlight is highly transient, since it depends on the current selection and we only allow one highlight to be shown at any given time. In addition, the optional overlay will eventually be toggled off, so that the user may continue working with the class diagram without the clutter induced by the tag cloud. These are some reasons that led to our second augmentation of class diagrams: the *thematic relation*.

### 5.1.2 Thematic relation

Thematic relations extend class diagrams to indicate that types use a common term.

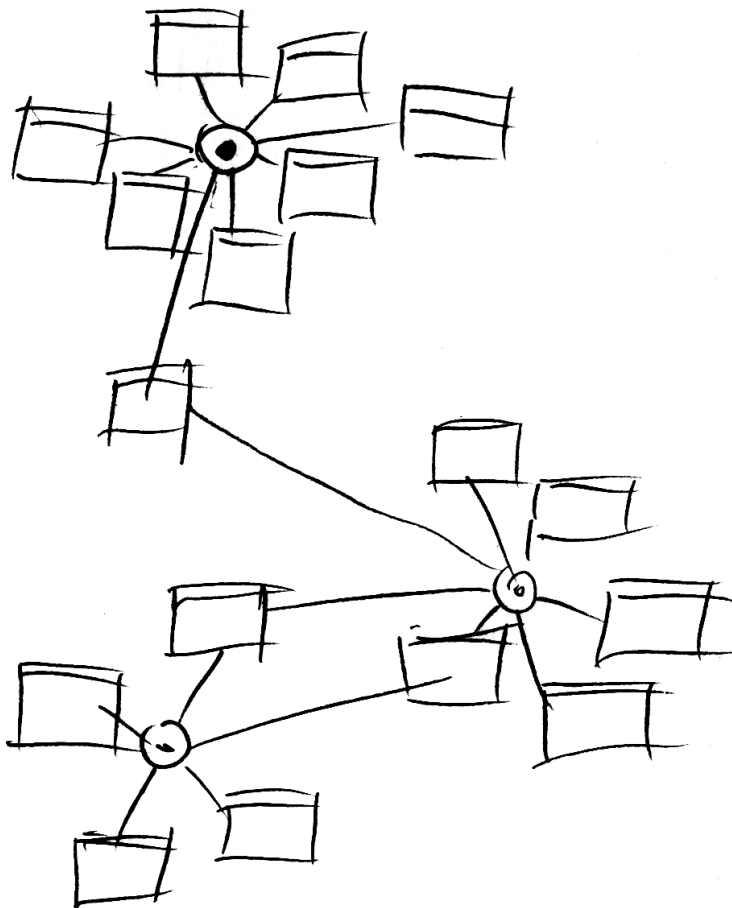
The thematic relation is a new relation for class diagrams and a supplement to the tag overlay as it is presented in the previous section. Thematic relations replace the grouping feature of the paper prototype (see section 4.2.5). Thematic relations are undirected hyperedges between classes using the same term in the identifiers of their source code. They are visualized as geometric shape, connecting all relevant classes with the relevant tag.



## Visualization

Figure 5.2 shows our first sketch of thematic relations. We thought of thematic relations as landmarks, with which users highlight important concepts and themes and where these are implemented in the code base. The thematic relation is made up of a tag in the center, representing a theme or concept, and connection lines to all types that use the corresponding term.

The relation visually links types and tags.



**Figure 5.2:** The first rough concept drawing of thematic relations. Boxes represent classes, circles tags.

## Interaction

Tags are converted into thematic relations.

Thematic relations can be created from the tag overlay by *pinning* a tag. By clicking the tag, it is converted into a thematic relation and added to the persistent elements in the diagram. Thus, the thematic relation does not disappear, when the tag overlay is deactivated. When the user identifies a tag that represents an important concept, she can pin it to be shown alongside the classes in the diagram.

The user can manually position pinned tags.

Once pinned, the tag becomes the widget to control the appearance of the thematic relation. Unlike the tags of the tag overlay, pinned tags can be moved and rearranged by the user to resolve overlaps with other diagram elements and allow for manual correction, in case the automatic arrangement creates a wrong emphasis on less important types.

Fonts and colors are customizable.

We also wanted to allow the user to edit the relation further, by making several properties editable. Finally, we decided to provide an interface for color and font customization.

### 5.1.3 Class diagram

We kept the design of type box widgets from the paper prototype.

Since the layout and interaction techniques used for the class diagram in the paper prototype were well received, the Silverlight prototype used an almost identical visualization and interaction scheme. One addition was a small tag cloud at the bottom of each type widget, displaying the ten most important terms used in the identifiers of the corresponding source code. This was meant to provide a developer with a very condensed abstract of the themes and concepts dealt with in each class.

## 5.2 Implementation

Our prototype was created using the SketchFlow prototyping framework of the Silverlight authoring tool *Expression Blend 3*<sup>4</sup>. We used the `edu.cmu.hcii.paint` code base from Ko et al. [2006] and created a class diagram using Relo. For tag metrics we used the term cloud view from Cultivate.

We used Sketchflow, Clutivate and Relo to create a prototype.

The prototype consists of three animations and one interactive mock-up of the tag overlay:

The prototype has four mock-up screens.

- Create a new diagram (animation)
- Expand an existing diagram (animation)
- Filtering using the tag overlay (interactive)
- Creating and customizing a thematic relation (animation)

The following sections will focus on these individual parts of the prototype.

### 5.2.1 Creating a new diagram

The first animation of the prototype shows, how the user can create a new diagram (figure 5.3). We showcase the drag-and-drop interface that was missed by our testers in the paper prototype.

The user creates a diagram with drag-and-drop.

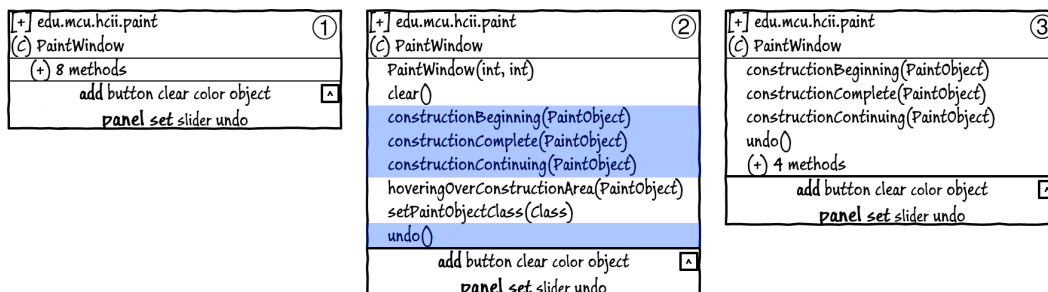
The user drags-and-drops a file from the project explorer of the IDE to an empty Code Gestalt editor window. A minimal widget with a header (including package and type names) and a tag cloud are generated (1).

The editor creates a type box from a dropped source file.

Instead of a list of members, only a button '+' with a label '8 methods' is shown to prevent visual clutter. By clicking the button the widget expands to show a list of all methods,

Members are hidden by default to prevent clutter.

<sup>4</sup>[http://www.microsoft.com/expression/products/blend\\_overview.aspx](http://www.microsoft.com/expression/products/blend_overview.aspx)



**Figure 5.3:** The minimal type widget (1) is expanded to show all methods (2). The user has selected four methods to be added to the SV and collapses the list again to show only the selected ones (3).

from which the user selects those she wants to add to the visualization (2).

The user can add members using standard interactions.

By clicking outside the widget, the list collapses to only those entries chosen by the user (3). The button label changes accordingly to '4 methods', since only four of eight methods remain hidden.

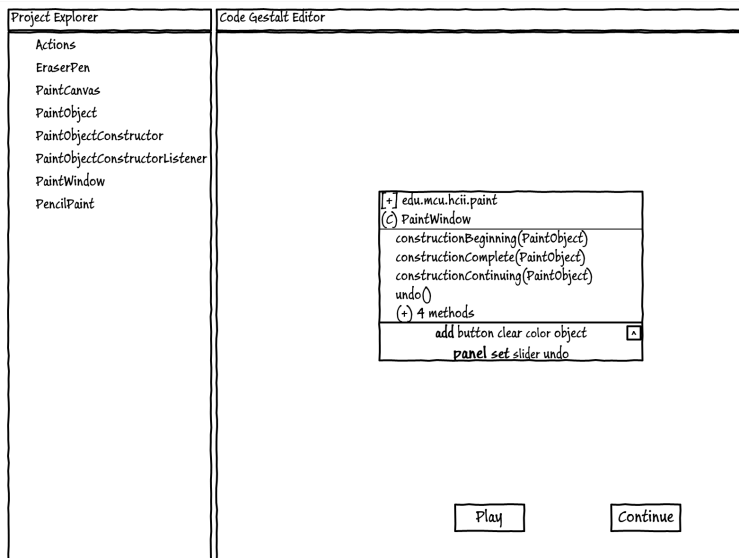
The IDE in our mock-up is highly abstracted.

The complete interface is shown in figure 5.4. We use a very simple layout of a left sidebar listing all files in the project and a large area on the right shared by Code Gestalt and source code editors.

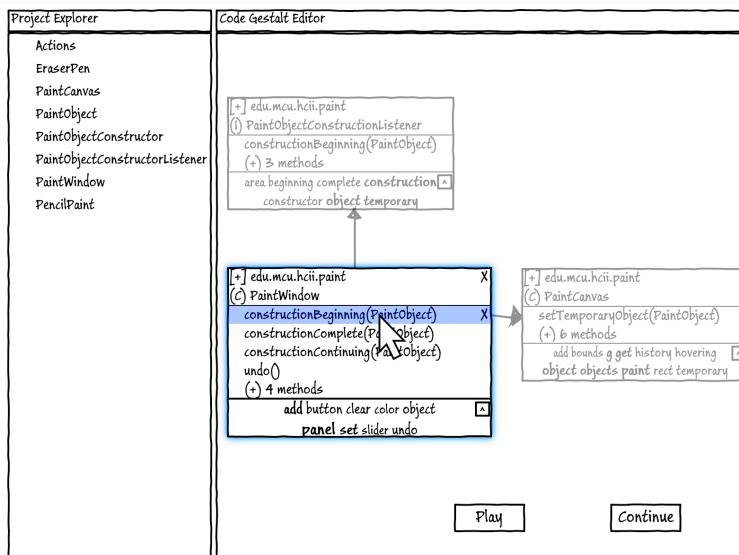
## 5.2.2 Expanding an existing diagram

Diagrams are expanded using context sensitive live previews.

The second animation shows, how the user is offered context sensitive options to expand an existing diagram. Figure 5.5 illustrates, how the current input focus toggles live previews of related entities. The user has selected the method `constructionBeginning`, which is an implementation of the same method in the interface `PaintObjectConstructorListener`. Moreover, the method calls another method, namely `setTemporalObject` of `PaintCanvas`. Hence, the two methods and a minimal widget of their types are displayed as semi-transparent live previews, while the method is selected.



**Figure 5.4:** The user has created a new diagram by dragging 'PaintWindow' to the editor and adding four methods to the class.



**Figure 5.5:** Context sensitive expansion options for `constructionBeginning`. An implemented method is shown above (`constructionBeginning`) and a called method on the right (`setTemporaryObject`).

Method calls are shown horizontally, inheritance vertically.

Note that the new elements are placed on different axis: Inheritance/implementation relations are shown on a vertical, the call relations on a horizontal axis. The user may click the previews to make them persistent in the diagram, thus expanding it. In the animation, the user chooses to make `PaintCanvas` persistent by clicking the ghosted preview. This technique is almost identical to the one shown in the paper prototype (see section 4.2).

### 5.2.3 Tag overlay

Using Cultivate and Relo we created an interactive mock-up of the tag overlay.

The center piece of the prototype is the interactive mock-up of the tag overlay. We created a class diagram with Relo and used it as basis for the visualization. For each of the types in the diagram we generated a tag cloud with Cultivate's cloud view and used these to create an approximation of what the overlay would look like for the 13 most frequent terms. The result is shown in figure 5.6.

The mock-up allows for interactive highlighting.

The prototype allows the user to perform highlighting of types by selecting tags and vice versa as described in section 5.1.1.

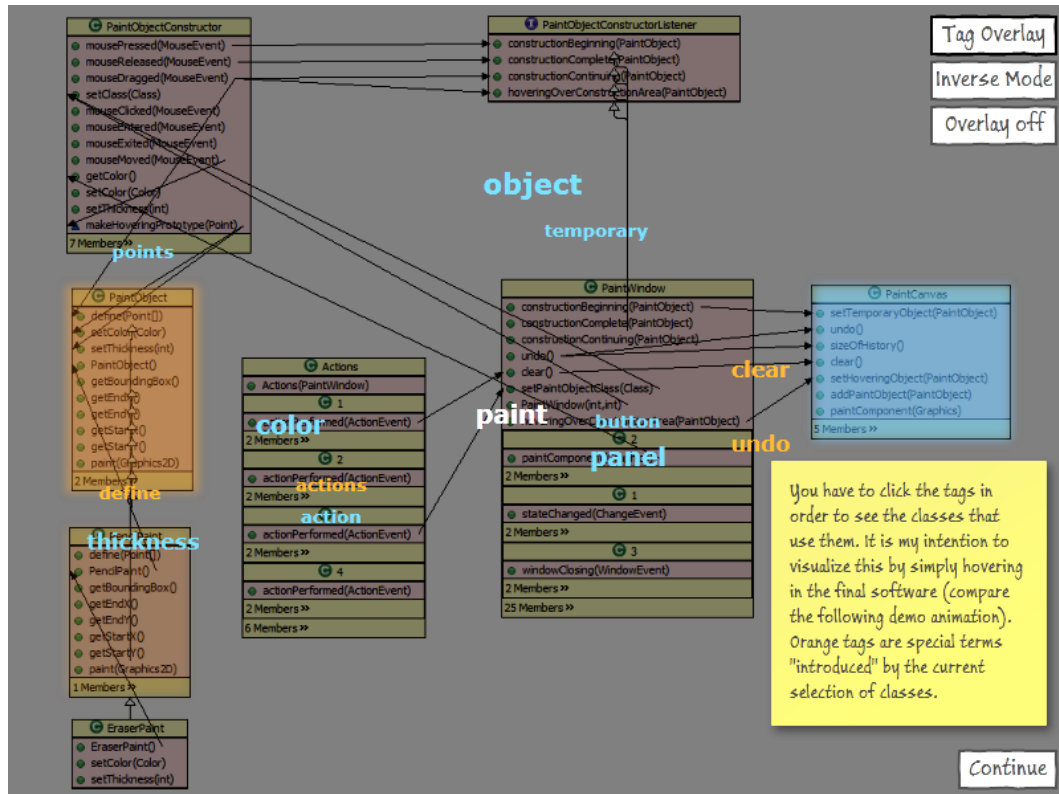
### 5.2.4 Thematic relations

Bézier splines represent thematic relations.

Manually drawn Bézier splines represent thematic relations in our prototype. The shapes connect all types using a term with the corresponding tag. The different weights assigned to the term by individual types is represented by the diameter of the shape of the thematic relation around each type (see figure 5.7).

Colors and fonts are changed from a radial menu.

We allow for tag font and relation color customization. Figure 5.8 shows a radial context menu that is displayed around a pinned tag, when the user clicks it. A half-circle in the upper half showcases a number of different fonts, which may be used to customize the appearance of the tag. At the bottom a color palette is displayed. The user may select a color from this palette to change the fill color of the



**Figure 5.6:** Our mock-up of the tag overlay. The user has selected ‘paint’, which is used by `PaintObject`, `PencilPaint`, and `PaintCanvas`. The orange (as opposed to the blue) highlight indicates that the term is introduced by `PaintObject` and `PencilPaint`.

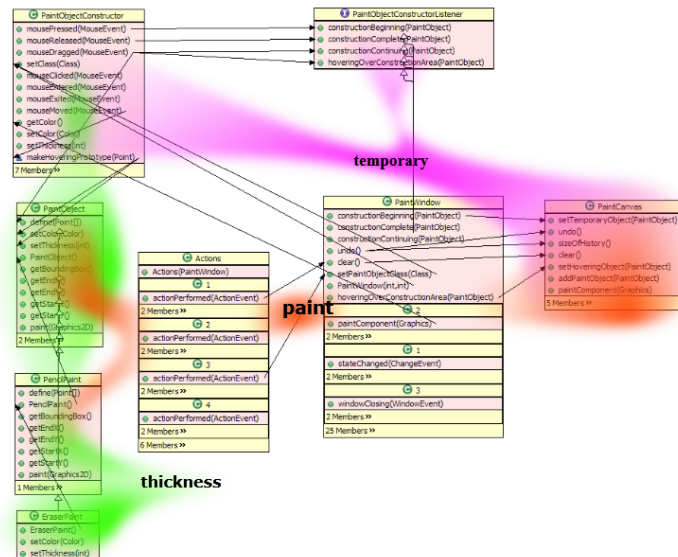
Bézier geometry. This customization allows users to create easily recognizable landmarks and emphasize important relations in the SV.

The Bézier geometry creates a visual landmark that allows for quick orientation in the otherwise monotonous diagram. The visualization was streamlined and simplified for real-time rendering in the final Eclipse implementation (see section 6.2.6), but the concept remained unchanged.

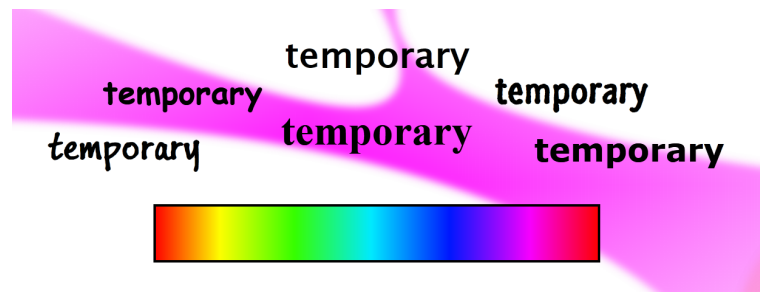
The Silverlight prototype contains an animation that shows the user interaction of selecting three tags and pinning them to the diagram. The user resolves an overlap manually, with the shape adjusting accordingly. Finally, she uses

Landmarks help users to orient in the diagram.

An animation shows the customization of a relation.



**Figure 5.7:** Three thematic relations in the Silverlight prototype: ‘temporary’, ‘paint’ and ‘thickness’.



**Figure 5.8:** Customization options for a thematic relation, here ‘temporary’. The original tag widget is shown in the center, a selection of different font styles can be chosen and the color may be modified using the bottom palette.

the radial menu to change the color of the thematic relation and the font of the tag (see section 5.1.2)



## 5.3 Evaluation

The prototype was made available online and a link was distributed through mailing lists at the *RWTH Aachen University* and the *University of Bonn* on March 17<sup>th</sup>, 2010. The build-in features of the SketchFlow player allow for annotations and free-hand drawings in the prototype. Users were encouraged to leave such feedback. We offered a download key of the video game *World of Goo*<sup>5</sup> to be given to a random participant as incentive. However, we only got feedback from one user in the time until March 31<sup>st</sup>, 2010.

The prototype was available online, but we got little response.

To gather more feedback on the prototype, we evaluated it with one research assistant and one student assistant (who were both not involved with the project) during think-aloud sessions. Also, we presented the prototype at the *Institut für Informatik III* of the University of Bonn and gathered feedback from an open discussion.

We gathered feedback from other CS students and researchers.

Overall, the prototype was well received. Some shortcomings were mentioned by testers that were due to the limitations of the prototype. E.g. the '+' button was not identifiable as such, since it was realized with a label. Another common problem was the high speed of the animations and sparse explanation. This is something we will keep in mind for future prototypes of the kind. Also, the class diagram created by Relo was perceived as cluttered and unattractive.

Testers had minor complaints about the presentation.

Apart from such problems that are clearly traceable to the limited capabilities of an animation prototype, all testers not familiar with Cultivate had problems with the concept of the two different tag and highlight colors. Even after explaining the meaning, this was not conceived as intuitive. However, the newly introduced concepts of filtering the code base visually using the tag overlay and the introduction of thematic relations were received positively by all reviewers.

Testers were irritated by tag colors. The tag overlay and thematic relations were well received.

---

<sup>5</sup><http://worldofgoo.com/>

Hence, we decided to concentrate our further efforts on creating a working visualization tool to evaluate Code Gestalt in practice.

## Chapter 6

# Eclipse implementation

*“Borgus frat! ‘Let’s see what she’s got,’ said the captain. And then we found out, didn’t we?”*

*—Montgomery Scott about the new starship Enterprise (Star Trek V: The Final Frontier)*

After evaluating the Silverlight prototype, we were confident that our design was mature enough to be implemented as actual SV tool. The centerpieces are the tag overlay and the thematic relations. Both these new concepts are derived from our work on the online survey and the paper prototype.

We will shortly revisit design changes between the Silverlight prototype and our final Eclipse plug-in in section 6.1. The technical aspects and practical considerations needed to implement Code Gestalt are discussed in section 6.2. Finally, section 6.3 presents our two-step evaluation of Code Gestalt.

### 6.1 Design

The design of Code Gestalt was already well established as augmented class diagram editor by both the paper and Silverlight prototypes (see chapters 4 and 5). The Eclipse

We used the Silverlight prototype as design for an Eclipse plug-in.

We present design, implementation and evaluation of the plug-in.

We unified the color of tags.

implementation follows the Silverlight prototype closely, since it was intended to be a working tool to evaluate the validity of both the tag overlay and the thematic relations. However, a change by design is the removal of differently colored tags, since this metric was not well perceived by the testers.

Some features from the Silverlight prototype had to be adapted<sup>x</sup> for Eclipse.

We needed to simplify and alter some visuals and interactions from the Silverlight prototype to comply with the UI guidelines of Eclipse and work around some framework limitations. We will discuss these changes and our solutions in detail in the following section.

Diagrams behave like any other project file.

From the paper prototype we kept the concept that diagrams follow a default *document life cycle* of open, edit, and save. Also, the diagrams should integrate with the IDE project (and file) structure, so they are compatible with version control systems and are available to all project developers.

## 6.2 Implementation

### 6.2.1 Framework

Code Gestalt is an Eclipse plug-in to visualize Java code.

Code Gestalt was implemented as a plug-in for the cross-platform IDE *Eclipse*<sup>1</sup>. We decided on Java as the programming language to create visualizations for, since among the numerous languages supported by Eclipse, the tool set for *Java*<sup>2</sup> is the best documented and most comprehensive one (Eclipse itself is mostly written in Java [Clayberg and Rubel, 2008]).

Code Gestalt adds a new file type and editor to the IDE.

There are two principal options to extend the Eclipse IDE with windowed areas suitable for displaying large visuals: *views* and *editors*. While views are meant to display complementary information (call hierarchies, errors, properties, etc.), editors are reserved for modifying resources that follow the document life cycle. Hence, Code Gestalt is imple-

<sup>1</sup><http://www.eclipse.org/>

<sup>2</sup><http://www.oracle.com/us/technologies/java/index.html>

mented as an editor, while most other visualizations from section 2 are implemented as views (if they are available as Eclipse plug-in).

The graph-based nature of Code Gestalt made it an easy choice to use the *Graphical Editing Framework*<sup>3</sup> (GEF) to implement our editor. GEF enforces the separation of model, view (called *Figure*), and controller (called *EditPart*). The framework is specialized on interactions for graph editors and the representation of (nested) model data. It also allows for the easy implementation of commands that support undo.

We use the GEF as application framework.

We use the *Java Development Toolkit*<sup>4</sup> (JDK) to extract a model from the Java source code and Cultivate to generate tag metrics (used for tag clouds, the tag overlay and thematic relations).

JDK provides us with the Java model, Cultivate with the tag metrics.

### 6.2.2 Other resources

We loosely followed Moore et al. [2004] in the implementation of Code Gestalt, but also used several articles and tutorials, most prominently the shape diagram editor<sup>5</sup>, UML diagram<sup>6</sup>, and drag-and-drop<sup>7</sup> *Eclipse Corner* articles. For implementation details we also referred to the open source code bases of Relo, Cultivate, and the examples provided by the GEF project in the package `org.eclipse.gef.examples`. Very late in the development process we also used Clayberg and Rubel [2008] for reference.

The implementation was guided using external references.

Although Code Gestalt's basic graph editing capabilities very closely resemble those of Relo, Code Gestalt uses a discrete code base. Our implementations of model extraction from Java code, interaction design, and rendering were de-

Code Gestalt does not use code from Relo.

<sup>3</sup><http://www.eclipse.org/gef>

<sup>4</sup><http://www.eclipse.org/jdt/>

<sup>5</sup><http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html>

<sup>6</sup><http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html>

<sup>7</sup><http://www.eclipse.org/articles/Article-GEF-dnd/GEF-dnd.html>

veloped from scratch. The biggest commonality with Relo on the source code level is the use of the GEF.

### 6.2.3 Eclipse integration

Code Gestalt follows the Eclipse User Interface Guidelines.

One of the development goals was to seamlessly integrate Code Gestalt with the target IDE. Our intend is to minimize teething troubles caused by the introduction of users to a new and unfamiliar user interface. Figure 6.1 shows how Code Gestalt integrates with the Eclipse workbench. We tried to adhere to the *Eclipse User Interface Guidelines*<sup>8</sup> as much as possible. Accordingly, the icons introduced by Code Gestalt to the Eclipse IDE were designed with *Expression Design*<sup>9</sup> and *Paint.NET*<sup>10</sup> using the guidelines on UI graphics.

Code highlighting in Java editors is not changed by Code Gestalt.

We did not implement syntax highlighting in the Java Editor based on thematic relation colors as we did with group colors in the paper prototype. This decision is based on the mixed feedback we got from the evaluation of the paper prototype (see section 4.3.1). A second reason to not implement the feature was the open question of how to achieve consistent highlighting of code, when multiple Code Gestalt diagrams were opened or types were part of multiple thematic relations.

#### Diagram creation

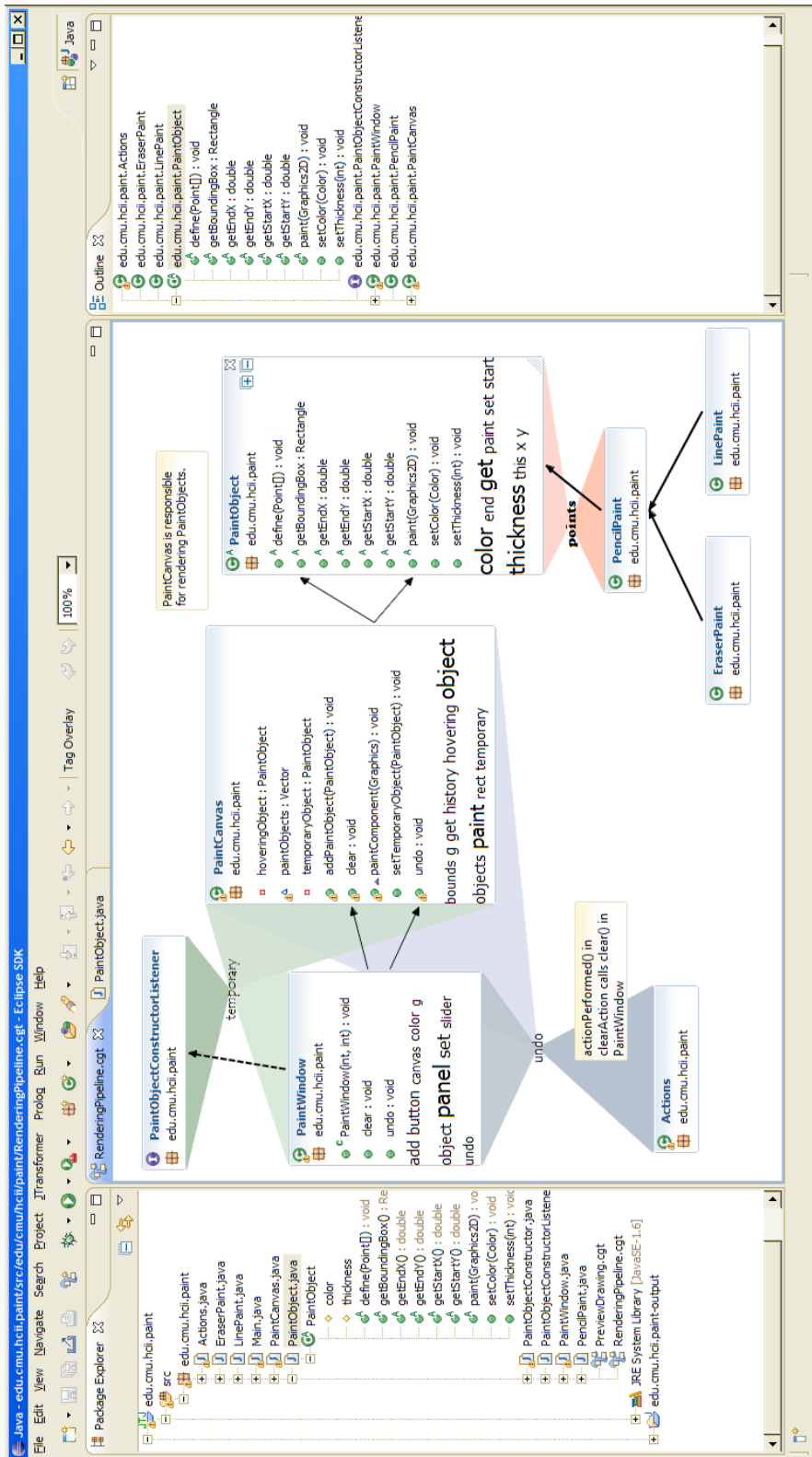
Code Gestalt diagrams are provided through default Eclipse wizards.

Code Gestalt is made available to the user through the new file type *Code Gestalt Diagram* that integrates with all default mechanisms in Eclipse to create new files. Since users asked for this in the paper prototype (see section 4.3), we also added a button for access to the *New Code Gestalt Diagram* wizard (all files in Eclipse are created through so called *wizards*). Figure 6.2 shows the interface to create a new Code Gestalt diagram file. The layout follows other Eclipse wizard very closely to achieve consistency in the user interface.

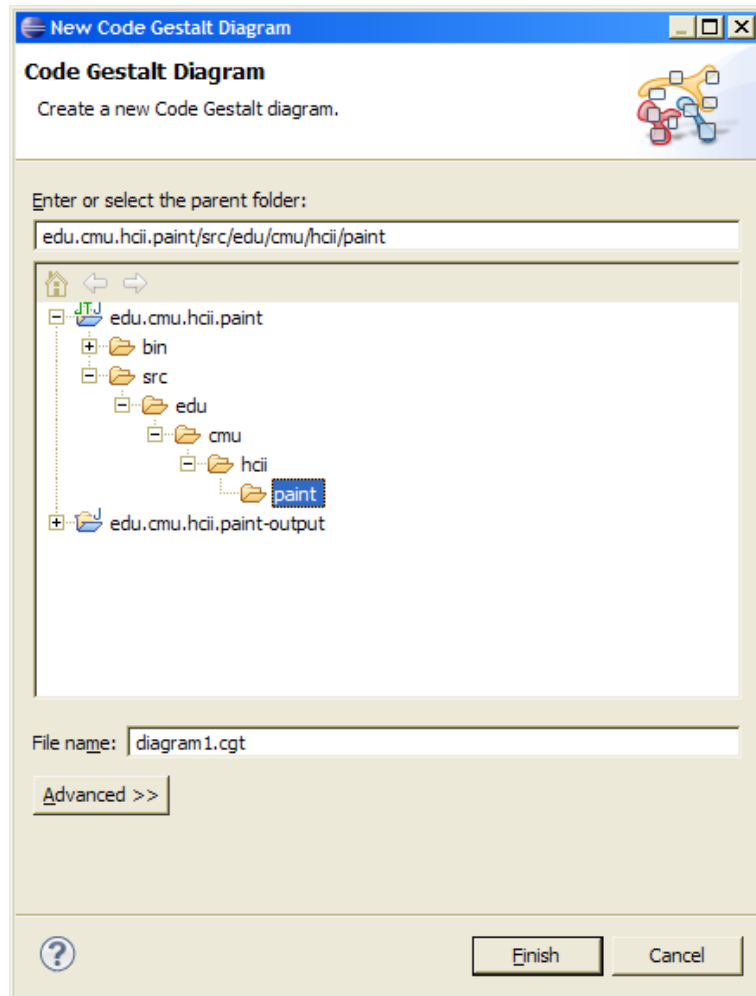
<sup>8</sup>[http://wiki.eclipse.org/User\\_Interface\\_Guidelines](http://wiki.eclipse.org/User_Interface_Guidelines)

<sup>9</sup>[http://www.microsoft.com/expression/products/Design\\_Overview.aspx](http://www.microsoft.com/expression/products/Design_Overview.aspx)

<sup>10</sup><http://www.getpaint.net/>



**Figure 6.1:** The IDE Eclipse with the Code Gestalt Plugin. The screen shot shows an open Code Gestalt editor in the center of the workspace. Code Gestalt integrates with the tool bar (sixth icon from the left), the project (files `PreviewDrawing.cgt` and `RenderingPipeline.cgt` in Package Browser) and the Outline (on the right).



**Figure 6.2:** The dialog to create a new Code Gestalt diagram.

Users can create diagrams from selected Java entities.

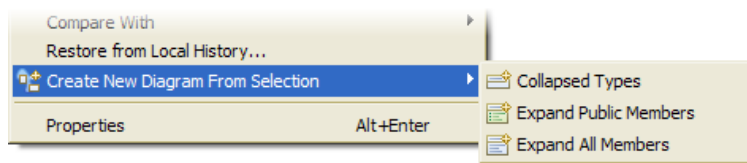
The diagrams created in this manner are empty and need to be populated by the user by dragging-and-dropping code entities from a view. An alternative is the creation of a new diagram from the context menu of one or more selected Java entities (see figure 6.3). This feature was requested by a tester, when tracking down bugs in an early build of the plug-in. We provide different levels of pre-population of the generated type boxes:

- *Collapsed Types*: Only the selected elements will be inserted in the new diagram. I.e., no members other



than those explicitly selected are added to the type member lists.

- *Expand Public Members*: All elements selected and all public members of selected types will be inserted in the new diagram.
- *Expand All Members*: All members from all selected types will be inserted in the new diagram.



**Figure 6.3:** The granularity options for creating a new diagram from a selection, e.g., in the package explorer.

## Editor

The Code Gestalt editor follows the open, edit, and save document life cycle. It can be deployed anywhere, other Eclipse editors can. Multiple instances for multiple documents can be open simultaneously. This allows for a multitude of possible workspace setups. E.g., tabbed editors, side-by-side of Java code and Code Gestalt diagram, or even multi-monitor configurations.

The editor introduces a very limited set of additional UI elements to the Eclipse tool bar:

- a button to toggle the tag overlay
- undo and redo buttons
- a drop-down list for zooming

Users can arrange Code Gestalt editors freely on their workspace.

Code Gestalt adds only few UI elements to Eclipse.

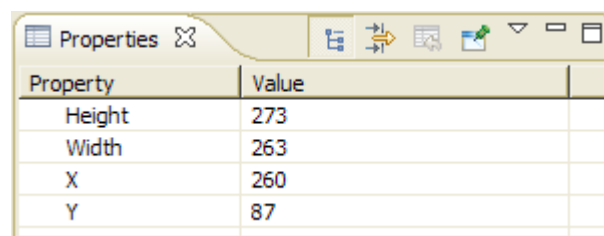
Different locations in the UI provide access to editor commands.

The same set of commands is also available through the main menu. The context menu of the editor supports undo and redo. All user actions in Code Gestalt are executed through GEF commands. Our implementation fully supports undo and redo.

### Complementary Eclipse views

Code Gestalt supports Eclipse's outline and properties views.

Like many other Eclipse editors Code Gestalt supports the use of two standard views, the *outline* and the *properties* views. The outline contains a hierarchical view of the contents in the current selection. In the case of Code Gestalt all types and members are shown which are visible in the diagram (see figure 6.1). The properties view allows for manual editing of diagram elements, usually exact positioning as in figure 6.4.



Property	Value
Height	273
Width	263
X	260
Y	87

Figure 6.4: Properties view for a selected type

### Drag-and-drop

By popular demand we implemented drag-and-drop as interaction to edit diagrams.

Code Gestalt supports drag-and-drop of Java entities from any Eclipse view. When a selection of elements is dropped to the diagram, Code Gestalt automatically creates a type box for the Java entities. However, the Code Gestalt editor does not accept drag-and-drop of source code or diagram elements from other Code Gestalt editors. This is a limitation that will be removed in future versions (compare section 6.3.4).

## Code editors

The user can always access the source code of any visualized Java entity by double-clicking it. The respective code is opened in an Eclipse Java editor and can be reviewed and edited as usual.

Double-clicking diagram elements opens a code editor.

### 6.2.4 Class diagram editor

#### Type boxes

The Code Gestalt editor allows users to create class diagrams. We represent classes as boxes that contain a list of members and a tag cloud displaying the ten most frequent terms (see figure 6.5). The tag cloud is sorted alphabetically, since this layout is the quickest to parse [Halvey and Keane, 2007].

Boxes in the diagram represent types.

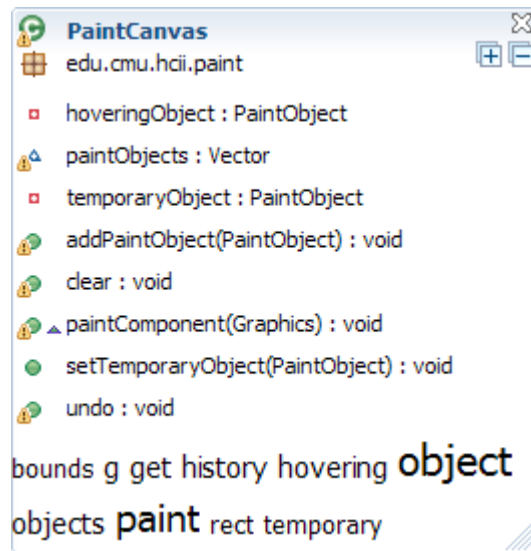
Which types and which members are visible is completely up to the user to decide. Based on the results from the online survey, we let users stay in control over the scope and level of detail. This way, Code Gestalt diagrams are less prone to clutter and the diagram designer can keep the visualization focused.

The user controls the scope of the diagram.

Therefore, users add types to the diagram by simple drag-and-drop from any other Eclipse view, such as the package explorer. Methods can be added using one of several interactions:

Members can be added automatically using several presets.

1. *Drag-and-drop*: If the type of the member is not included in the diagram, it will be added with only the dragged member(s) visible. If the type is already present, the user must drop the member on that type.
2. *Contextual buttons*: When a type is selected, two buttons in the top right corner of the box allow for quickly expanding the type to show all members or collapsing it to show none (see figure 6.5).



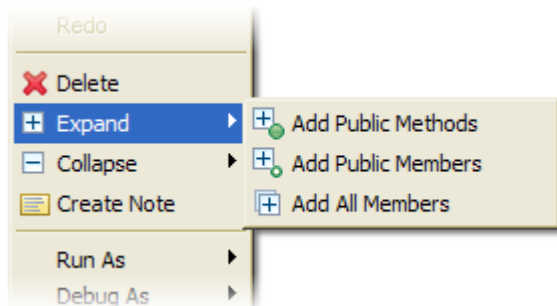
**Figure 6.5:** The representation of a type in Code Gestalt. In the header we have the type name and package with their default Eclipse icons. In this case the class `PaintCanvas` from the package `edu.cmu.hcii.paint` contains warnings as indicated by the yellow sign. In the top right we present the user with contextual buttons for closing, expanding and collapsing the type box. The header is followed by a list of members. Default Eclipse icons identify the nature of the members. At the bottom we display a tag cloud of the ten most frequent terms. A contextual resize handle is shown in the bottom right corner.

3. *Context menu presets:* To assist users in quickly filtering the list of available members, we provide six convenience presets via the context menu (see figures 6.6 and 6.7).

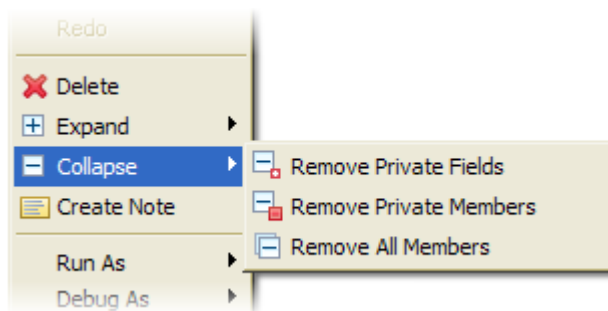
### Inheritance and call relations

Call relations pass horizontally, inheritance relations vertically.

Code Gestalt supports two structural relation types, namely inheritance and call relations. As in previous prototypes, inheritance relations are connecting top and bottom edges of type boxes, call relations the left and right edges of methods. Similar to types and members, we designed vi-



**Figure 6.6:** Options in the context menu of types for adding members.



**Figure 6.7:** Options in the context menu of types for removing members.

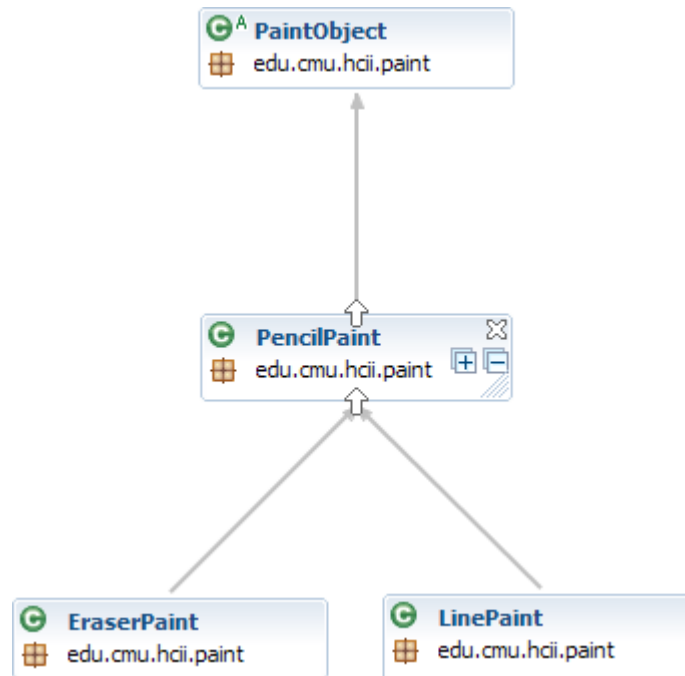
ualization and interaction techniques to both prevent clutter and support the user in quickly creating and finding relevant entities at the same time.

Thus, we visualize relations in two steps. When the user selects a diagram element, Code Gestalt automatically parses the inheritance or call hierarchy (depending on the selection being a type or method) and displays any relation to other diagram elements as gray arrows (see figure 6.8). We call these faint and transient objects *live previews*.

The preview relations are visible as long as a connected element has a selection focus. This way, the user is informed about the presence of a relation by Code Gestalt. If the user feels a relation to be important for the aspect of the software he wants to visualize, he can make the relations persistent by clicking them. Also, all incoming or outgoing relations

Code Gestalt searches for relations and displays live previews of them.

Users decide which relations to add to the persistent diagram by clicking previews.

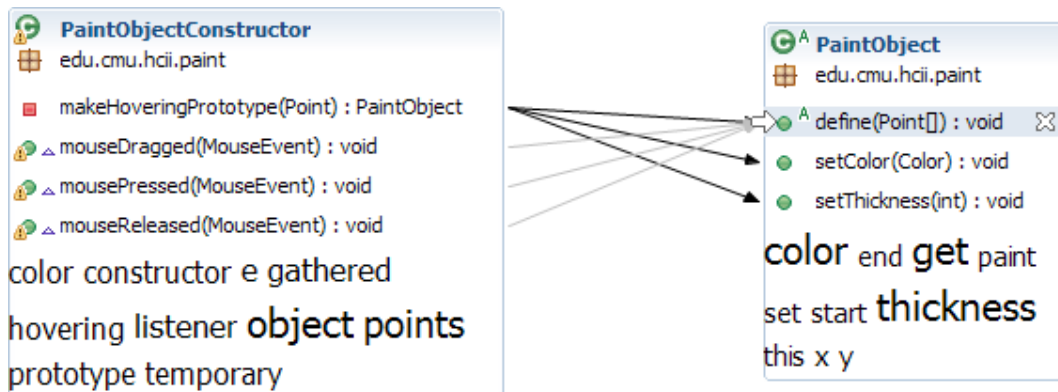


**Figure 6.8:** The selected type `PencilPaint` extends `PaintObject` and is extended by `EraserPaint` and `LinePaint`. The user can click any individual relation to make it persistent. The two arrow symbols in the top and bottom center of the `PencilPaint` type box are contextual buttons that allow the user to make all incoming or outgoing preview relations persistent.

can be made permanent at once by clicking contextual anchor buttons. Persistent relations are rendered black and stay visible, regardless of the current selection (see figure 6.9).

Live previews only work for relations to existing diagram elements.

Due to time constraints we were unable to implement live previews for types and methods which are not already included in the diagram, as shown in the third use case of the paper prototype (section 4.2.3) and the second animation of the Silverlight prototype (section 5.2.2). When prioritizing features, we felt that to our research goals the preview technique was secondary to the tag overlay and thematic relations and could be studied, at least superficially, using the implementation for relations presented in this section.



**Figure 6.9:** Preview and persistent call relations. The method `define` in `PaintObject` is selected, which is called by all visible methods in `PaintObjectConstructor`.

### 6.2.5 Tag overlay

The tag overlay can be toggled by the user with a dedicated tool bar button. When the overlay is active, the terms from the source code identifiers are shown in front of the faded class diagram (figure 6.10). As presented in section 5.1.1, the tags are spatially arranged, so that they are positioned near those types in which the corresponding terms are most frequently used.

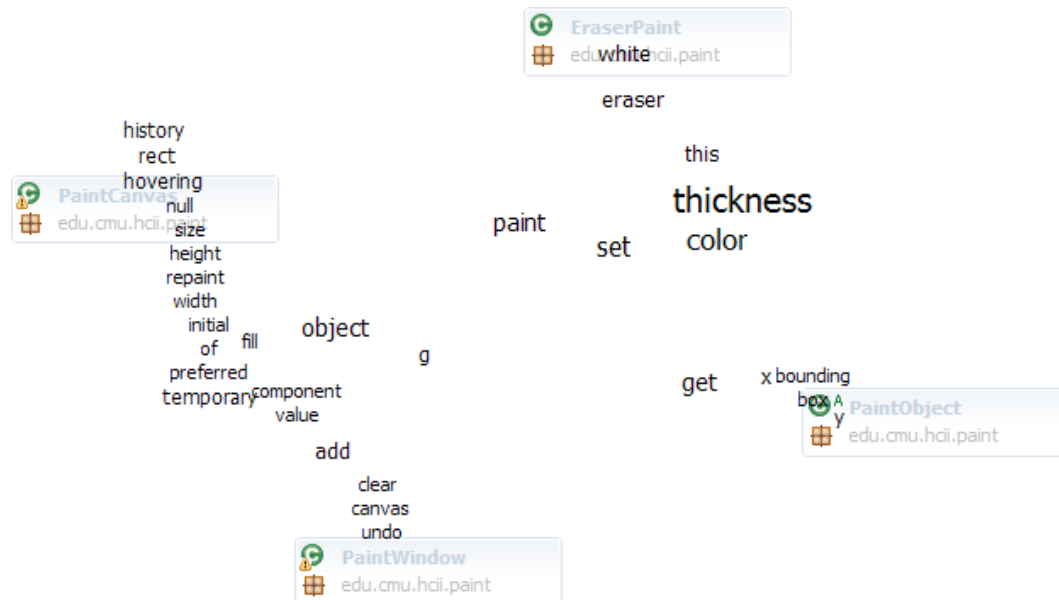
The implementation of the tag overlay is based on tag metrics obtained from Cultivate. We calculate a tag cloud for each type, mapping each term to its occurrence frequency. We use this information to determine the ‘center of gravity’ and font-size for the tags in the overlay.

Provided, that a consistent and reasonable naming scheme was used in the creation of the source code, the tag overlay is a thematic map. The position of terms hint at the distribution of responsibilities, concepts and themes in the source code. While the class diagram uses syntactical analysis to visualize structural features of the source code, the tag overlay uses vocabulary analysis to allow for a more semantical approach to the code base.

The tag overlay fades out the class diagram and displays a tag cloud in front.

Cultivate provides the tag metrics.

Given a consistent naming scheme, the tag overlay provides a thematic map.



**Figure 6.10:** The tag overlay for the four types `EraserPaint`, `PaintCanvas`, `PaintObject`, and `PaintWindow`. `thickness` is the most frequent term.

### Sweepline algorithm

Multiple tags might overlap.

Since multiple terms tend to appear in the same types in similar frequency, it is quite probable that positioning tags around their center of gravity (using the term frequency assigned by the individual types as weights) will cause overlaps.

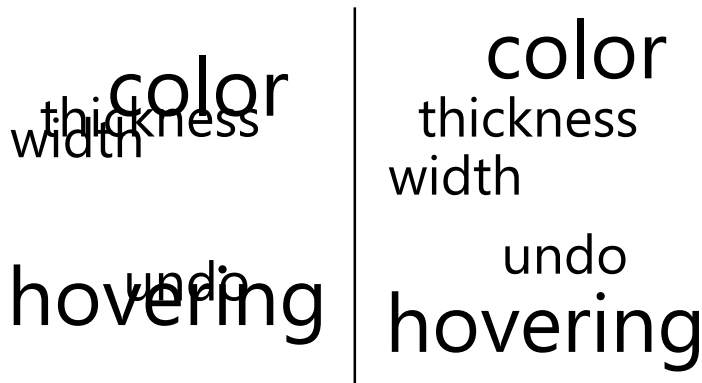
We resolve overlaps with a sweepline algorithm.

We resolve this problem using a custom sweepline algorithm. The algorithm groups tags that directly or indirectly overlap, and ‘stacks’ them on top of each other around a common center (figure 6.11). Thus, tags are no longer positioned exactly at the center of gravity in favor of readability.

An ideal solution is not efficiently computable.

This solution of course has its limitations, since two stacks of previously non-overlapping tags might overlap after the transformations of the algorithm are applied to the tags. However, the solution for this problem is known to be NP-hard [Marks and Shieber, 1991]. In practice our algorithm resolves overlaps to a degree, where there is little to no tag overlapping, so we consider this solution an acceptable and efficient compromise.





**Figure 6.11:** The swepline algorithm detects tag overlaps (left) and resolves them by stacking the tags on top of each other around the common center (right).

## Highlighting

As in the Silverlight prototype, we implemented a two-way highlighting function in Code Gestalt. Through this feature we allow for visual searches. The user may select a tag in order to visualize the term frequency in the types of the diagram (figure 6.12), or select a type to visualize the term frequencies for that type (figure 6.13). Code Gestalt creates a heat map of relates diagram and tag overlay elements. The intensity of the highlight color is proportional to the respective frequency.

Users can create heat map highlights for tags and types.

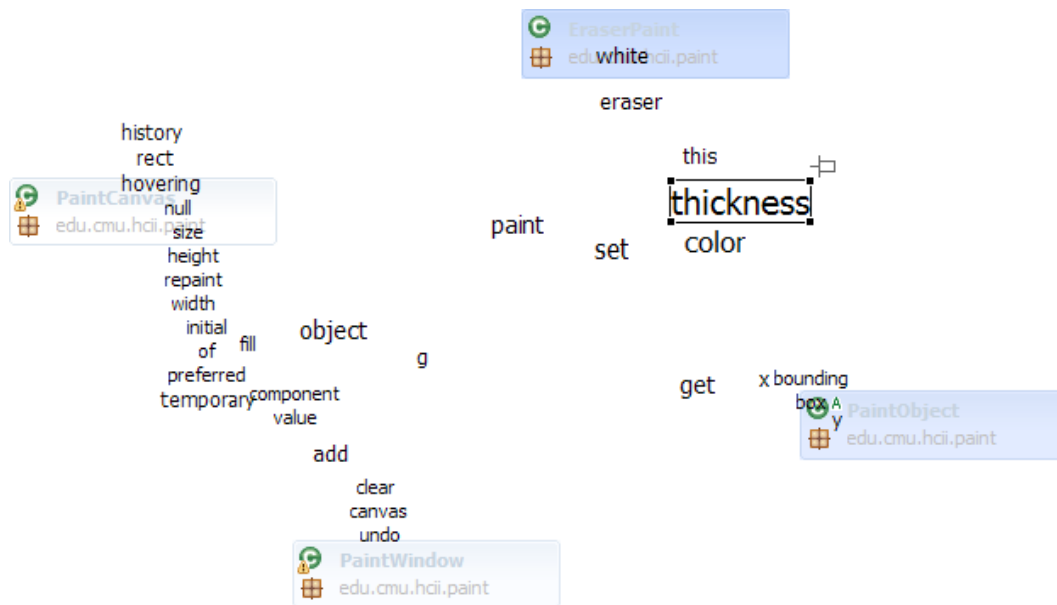
### 6.2.6 Thematic relations

The user creates thematic relations from tags in the tag overlay. Since in GEF single-clicking is reserved for selecting items, we chose another way to convert tags to thematic relations: a pin-shaped button. Hence, we speak of *pinning* a tag to the diagram.

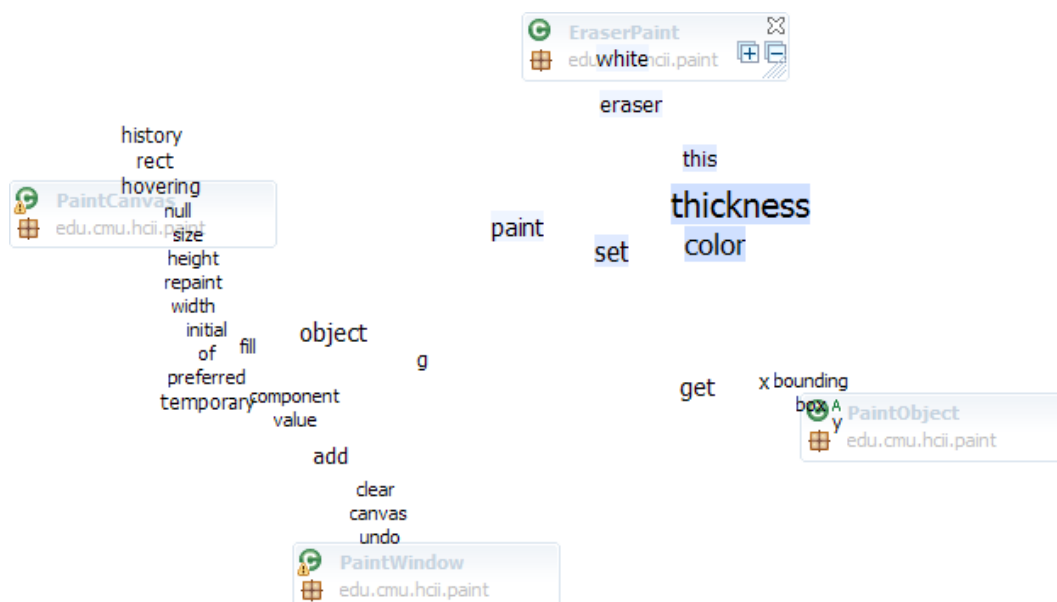
Tags are converted into thematic relations using a pin button.

A pinned tag becomes the center of the created thematic relation (see figure 6.14). Its position is now editable by the user and is no longer changed automatically, when other parts of the diagram are rearranged. This way, we want to keep the SV predictable and controllable for the user. Also,

Users may rearrange pinned tags.

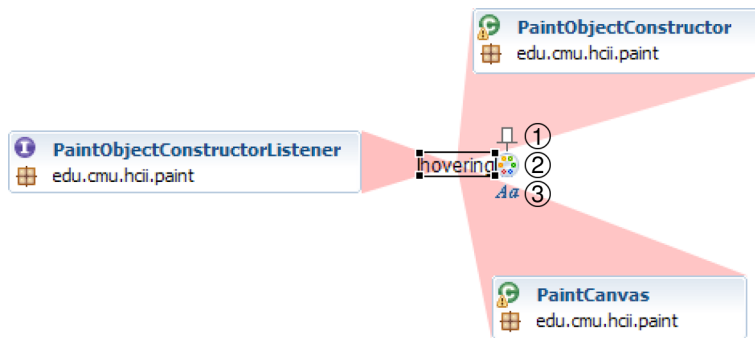


**Figure 6.12:** By selecting the tag 'thickness', the term frequencies are visualized as highlight in the type boxes.



**Figure 6.13:** The user has selected EraserPaint and highlights tags based on the term frequency in its identifiers.

the user can rearrange tags to resolve overlaps and change the emphasis on different parts of the diagram.



**Figure 6.14:** The thematic relation can be manipulated using contextual controls. The tag of the relation can be unpinned, removing the relation from the diagram (1), the thematic relation can be recolored (2), and the font of the tag may be changed (3).

Due to limitations in the GEF framework and real-time rendering requirements, the geometry of the thematic relations is a lot simpler than in the Silverlight prototype. Instead of Bézier splines, we use cones to connect types with the center tag. We do not visualize different term frequencies by the diameter of the connection geometry, but rather the opacity of the cone segments. So cones connecting types that have a high term frequency are more intense than cone segments to types with a lower term frequency.

We replaced Bézier splines with cones of varying opacity.

Figure 6.14 shows the context sensitive controls that replace the radial menu from the Silverlight prototype. We found this interface to be more consistent with the way we allow interaction with type boxes and notes. Next to a selected tag, we display three contextual buttons. The pin allows the user to unpin the tag from the diagram, which also removes the thematic relation. The two other buttons open dialogs to pick the color of the thematic relation and the font for the tag.

Contextual buttons allow for color and font customization.

Note, that the types connected by a thematic relation form a group, very much like the groups in the paper prototype (see section 4.2.5). However, groups in the paper prototype were created independent from filters (although filters

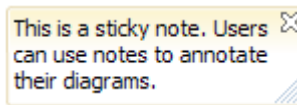
Thematic relations constitute auto-updated groups.

could help users to identify types as potential group members). Thus, it was not possible to create groups that would automatically update, when the source code changed or a new type was added. The thematic relation however is updated automatically. E.g., the opacity of cone segments is updated, when the source code changes. Similarly, newly added types that contain matching terms are automatically connected to existing thematic relations.

### 6.2.7 Notes

Users can create notes to annotate diagrams.

To allow users to freely annotate diagrams Code Gestalt provides notes. These are simple boxes that a user may type into (see figure 6.15). New notes can be created from the editor context menu. The note widget only supports plain text.



**Figure 6.15:** Notes allow users to annotate Code Gestalt diagrams

## 6.3 Evaluation

Testers were members of the CS departments from two universities.

We evaluated Code Gestalt in late 2010 at the *RWTH Aachen University* and the *University of Bonn* by means of a user study. In the following we will present the goals, design, and results of that study.

### 6.3.1 Goals

Let us revisit the original design goals from section 1.1:

1. Use semi-automation features (live previews, drag-and-drop diagram creation, etc.) to support users in creating diagrams and prevent the creation of invalid diagrams that do not match a given code base, but let users stay in control of the diagram scope.
2. Employ a new visualization metaphor that allows the user to harness the human intelligence present in the source code (through code vocabulary analysis) and to better organize the diagram.
3. Achieve good usability and fast diagram creation to compete with pen and paper techniques.
4. The created SVs should be meaningful for any programmer, not only Code Gestalt users, so they can be used for inter-developer communication and project documentation.

We defined four primary goals and requirements for Code Gestalt.

To evaluate these goals, we decided to perform a two-step user study. In the first step, 16 users were asked, to complete four code visualization tasks using pen and paper and Code Gestalt. The competitiveness of Code Gestalt was quantitatively evaluated by measuring completion rates, completion time, and errors made by the users. To assess the usability of Code Gestalt we used a standardized *System Usability Scale* (SUS) questionnaire from Brooke [1996] (Goal 3).

We evaluate Code Gestalt with a user study and a follow-up online survey.

Goal 1 is partially enforced by our implementation of Code Gestalt and we asked testers to evaluate some related features using a questionnaire. Similarly, we asked testers to assess the features related to the tag overlay and thematic relations to evaluate point 2.

The study helps us to evaluate Code Gestalt as a system.

The diagrams created by the testers were then used for a second step of the evaluation. We designed two online surveys, in which testers were asked to compare pen and paper sketches against Code Gestalt diagrams in several cate-

The survey helps us to evaluate Code Gestalt as an SV.

gories. These results give us the means to evaluate goals 2 and 4.

### 6.3.2 Test design

Testers performed four tasks in 10 minutes each.

Each test session was scheduled to require at most 90 minutes. After discussing the test situation and explaining the informed consent form (see appendix C.1.1), we gave the testers a short introduction to the Eclipse IDE and the Code Gestalt plug-in. The testers were given a 10 minutes training period to get some hands-on experience with the IDE and SV tool. The testers were allowed and encouraged to study the source code of the project to be visualized in the next phase of the test.

Code Gestalt was compared to pen and paper sketching.

The main part of the user study was a series of four test tasks that asked the participants to understand and visualize different aspects and features of the `edu.cmu.hcii.paint` package<sup>11</sup> (see appendix C.1.2). Two tasks were to be performed with pen and paper (sketching) and two with Code Gestalt each. For sketching, we provided the testers with blank Din A4 paper sheets, a black pen (0.5 mm tip) and four highlighter pens (yellow, orange, magenta, and green; 2–5 mm tip). Both the order of tasks and tools was counterbalanced to prevent learning effects. We allowed participants to use the full feature set of Eclipse to explore the code. They were also allowed to use any number of sheets or Code Gestalt diagrams to note intermediate results, but were asked to turn in one sheet or diagram file at the end of each test task.

After 10 minutes testers had to turn in one task solution.

The testers had 10 minutes to complete each task after reading the description and clearing any questions with the principal investigator. The testers could indicate that they finished a task early, otherwise they were interrupted after 10 minutes and asked, if they considered the diagram to be complete or not.

---

<sup>11</sup>We use a modified source from Ko et al. [2006]

The final part of the test session was the completion of a survey form, including the SUS questionnaire, questions to rate existing features and open ended questions, comparing the Code Gestalt user experience to that of pen and paper.

Testers filled out a complimentary questionnaire.

Two weeks after the start of the first session participants were contacted by e-mail to complete one of two surveys. For each original test task, we selected two pairs of sketches and Code Gestalt diagrams that had to be compared in four categories. The pairs were selected, so that no tester had to review her own solution for any task. Consequentially, both surveys were send to eight participants each. Both surveys also included a list of requested features proposed by the testers during the study that should be rated by the testers for usefulness.

We send participants a follow-up online survey 14 days after principal testing.

### 6.3.3 Results

In the following we present the results from the user study, the complementary questionnaire and the online survey.

#### Population

Our 16 testers were recruited from the computer science departments of the RWTH Aachen University and the University of Bonn. Our test population covered a wide range of different experience levels, from novices with less than a year of programming practice to Java veterans who worked with early public versions of the programming language. For details refer to appendix C.2.1.

The participants covered a wide range of programming and SV experience.

#### Completion rates and completion time

The testers had 10 min to complete each task. When the time was up, the testers were interrupted and asked, if they had finished the sketch or SV. We tested the null hypothesis  $H_1$ : 'The completion rates for sketching and using Code Gestalt do not differ.' The differences between sketching

We found no significant difference in the completion rates of the two systems.

and the use of Code Gestalt were not significant according to Mann-Whitney tests for each of the four tasks. In other words, we cannot refute the null hypothesis.

Sketching was significantly faster for task #4.

We also tested the null hypothesis  $H_2$ : ‘The completion times for sketching and Code Gestalt do not differ.’ For the comparison of task completion times we only consider the times recorded for completed test tasks. We detected a significant difference between sketching and the use of Code Gestalt for test task #4,  $U = 9.00, z = -1.98, p < .05$ , with a large effect of  $r = -.53$ , meaning Code Gestalt users required significantly more time to complete the task. Hence, we can refute the null hypothesis for task #4. For details on the tests of completion rates and completion time please refer to appendix appendix C.2.2.

There was some inaccuracy in time measurement.

**Threats to validity** Time measurement was not precise for all test runs, when the principal tester failed to start the clock. In these cases we used an approximation of the completion time obtainable from screen recording or notes. Also, many testers reacted on the time limit and tried very hard to complete the task in the given time interval. One tester stated that he felt uncomfortable with the time limit, but wanted to continue the test nonetheless.

### Errors

We counted errors in five categories.

Our participants created diagrams that were meant to explain certain features in the `edu.cmu.hcii.paint` code base. For each task we created a list of code entities that had to be visualized for a diagram to be considered complete. These lists can be found in appendix C.1.4. When a sketch or diagram was missing an element from a list, we counted it as error in the respective category. Using the lists we check for the presence of five types of entities:

- Types
- Attributes
- Methods



- Call relations
- Inheritance relations

Using Mann-Whitney tests we tested the null-hypotheses  $H_3$ : 'The error count for sketching and Code Gestalt do not differ.' for each of the five categories and each of the four tasks. We can refute the null hypothesis in four of the 20 tested cases. Code Gestalt lowered error rates for types highly significantly for task #1,  $U = 8.0, z = -2.56, p < .01$ , and significantly for task #2,  $U = 9.5, z = -2.41, p < .05$ . In both cases we found a large effect with  $r_{\#1} = -.64$  and  $r_{\#2} = -.60$ . Likewise, Code Gestalt users made significantly fewer method errors in task #2,  $U = 12.0, z = -2.15, p < .05$ , again we found a large effect  $r_{\#2} = -.54$ . Finally, sketching produced highly significantly fewer call relation errors in task #3,  $U = 6.5, z = -2.92, p < .01$ , the effect was large with  $r_{\#3} = -.73$ . For details refer to appendix C.2.3.

Code Gestalt improved correctness of types and methods, sketching correctness of call relations.

### Qualitative feedback

We prepared two complementary open ended questions for testers to report advantages and disadvantages of Code Gestalt and pen and paper.

**Which aspects of Code Gestalt did you like? Did it offer any advantages over pen-and-paper sketches?** All 16 participants responded to this question. Seven testers found Code Gestalt diagrams to be advantageous, because they were easy to edit and correct. Six users perceived Code Gestalt to be faster than pen and paper, especially when it came to copying identifiers and signatures. Also, six testers stated that at least one of the new features (tag overlay, thematic relations, tag cloud) supported them in understanding the code base. An additional tester stated that paper would not teach a user anything about the code. The resulting diagrams were considered to be clearer than sketches by five participants. Four testers commended the automation features (live previews). Similarly, the usability was

Many participants felt that working with Code Gestalt was faster than pen and paper. The tag overlay and thematic relations helped them understand the code.

highlighted by four participants, with special emphasis on the drag-and-drop feature. Four users also attested good IDE integration. Four testers stated that they found the resulting diagram esthetically pleasing. The ability to open a Java editor displaying the code of a double-clicked diagram element was positively mentioned by three users. Two participants commented on the better legibility of Code Gestalt diagrams in comparison to handwriting in sketches. Again two users appreciated the integration of diagrams with the project.

We also got some individual responses. Very concisely, these were similarity to UML class diagrams, high information density, usefulness of call relations, simplicity, flexibility, interactivity, layout consistency, and WYSIWYG.

Sketching offers more flexibility and allows for parallelism.

**Which aspects of Code Gestalt did you dislike? Which advantages did pen-and-paper sketches offer?** We got responses from 13 users to this question. Five participants felt sketching was more flexible than using Code Gestalt, with two users noting the ability of drawing arbitrary relations on paper. Two testers attributed poor performance to our test system. Again, two users were missing those relation previews for elements not included in the diagram, with one additional participant stating that Code Gestalt was a good add-on, but highly dependable on other Eclipse views. Two users preferred sketches over Code Gestalt because of speed benefits, with one tester mentioning the benefits of parallelism, when sketching on paper. The tag cloud in type boxes was explicitly dismissed by two participants.

Single users gave feedback on a multitude of mostly UI related limitations.

A quick overview of other singular responses:

- Code Gestalt diagrams can clutter easily.
- Code Gestalt is an “unintuitive” name.
- Code Gestalt lacks expressiveness.
- Sketches allow the portrayal of objects without source code backing.
- ‘Remove All But Selected Members’ command is missing.

- Code Gestalt should automatically add inheritance relations.
- Annotations in handwriting are faster than Code Gestalt annotation tool.
- Code Gestalt annotation tool is too simplistic.
- Dragging handle area on type boxes is not always clear.
- Code Gestalt does not allow for inter-type relations.<sup>12</sup>

### Questionnaire

All participants were asked to fill out a survey at the end of the session. The questionnaire had three parts:

Testers filled out a three-part questionnaire.

1. 10 questions taken from SUS, complemented by four questions specially tailored to determine Code Gestalt's competitiveness in relation to sketching; answers could be given on a five-point Likert scale ('strongly disagree' to 'strongly agree')
2. 16 questions to determine the usefulness of individual Code Gestalt features; answers could be given on a five-point Likert scale ('not useful' to 'very useful')
3. two open ended questions asking the testers to describe advantages of Code Gestalt and sketching over the other

The complete questionnaire form can be found in appendix C.1.3. We map the answers from the five-point Likert scale to a normalized  $[-1..1]$  range, where  $-1$  represents 'strongly disagree' or 'not useful',  $1$  'strongly agree' or 'very useful', and  $0$  a neutral answer.

---

<sup>12</sup>We do not know how the tester arrived at this conclusion, since Code Gestalt allows and actively searches for call relations within types and shows previews of them.

According to SUS, Code Gestalt has good usability.

**System Usability Scale** From the 16 testers Code Gestalt received a mean SUS score of 79.5 ( $Mdn = 77.5, s = 8.4$ ), putting Code Gestalt's usability in the 'good' to 'excellent' range [Bangor et al., 2009].

### Additional usability questions

Testers agreed on Code Gestalt being a practical alternative to sketching.

The testes were given the following four statements expanding the default SUS set. The questions and descriptive statistics of the answers are given in table 6.1. While our testers had mixed impressions on Code Gestalt's flexibility, they perceived its predictability, competitiveness with sketching, and clearness of the SV favorably.

#	Statement	<i>M</i>	<i>Mdn</i>	<i>s</i>
11	I found Code Gestalt to be very flexible.	0.03	0.00	0.39
12	I could not predict the outcome of my interactions with Code Gestalt.	-0.44	-0.50	0.48
13	I believe using Code Gestalt is a practical alternative to creating diagrams by hand.	0.66	1.00	0.47
14	I expect the diagrams created with Code Gestalt to be confusing for most programmers.	-0.63	-1.00	0.50

**Table 6.1:** Response to the four statements on a scale from -1 ('strongly disagree') to 1 ('strongly agree').

### Individual features

We asked users to rate individual Code Gestalt features.

In the second part of the survey we asked the participants about the usefulness of individual features to review our design decisions. Each feature could be rated on a five-point Likert scale from 'not useful' to 'very useful'. The features with the descriptive statistics of the results are re-

ported in table 6.2. Numbers refer to the question in the questionnaire from appendix C.1.3.

The individual features of Code Gestalt were rated ‘useful’ (0.2 – 0.6) and ‘very useful’ (0.6 – 1.0) with the exception of the tag cloud included in each type box. The features specifically tied to the tag overlay and the thematic relations (#22–#26) received ratings in the range from ‘useful’ to ‘very useful’.

The tag overlay and thematic relations were rated useful.

#### 6.3.4 Second online survey

To determine, if we succeeded in delivering an SV that offers some degree of ‘human insight’ and is meaningful on its own without a tool and IDE allowing for further exploration, we designed two surveys, pitting a sketch against a Code Gestalt diagram for each test task. The participants of our survey were contacted by e-mail to evaluate the pairs against four categories on a seven-point Likert scale:

We let users compare pairs of sketches and Code Gestalt diagrams.

- Which diagram is clearer?
- Which diagram is more understandable?
- Which diagram would you rather use as aid to solve the task?
- Which diagram is better suited to document the source code?

To find pairs of sketches and diagrams that were roughly comparable, we computed the squared differences of errors and clutter for each pair in all five error categories. The amount of clutter was determined by counting the number of code artifacts in each category that were unnecessary to solve the task but included in the diagram. We allowed for some variation by adding optional entities that were not deemed essential, but also no clutter. These are the items in parenthesis found in appendix C.1.4. We reviewed the pairs with the lowest squared sums, but did not follow this ad-hoc heuristic blindly. We devised two sets of four pairs

We used squared differences of errors and clutter and manual filtering to find comparable pairs.

#	Feature	<i>M</i>	<i>Mdn</i>	<i>s</i>
15	Similarity to UML class diagrams	0.73	1.0	0.32
16	Generation of diagrams from a Project Explorer selection	0.96	1.0	0.13
17	Adding types and methods using drag-and-drop	0.84	1.0	0.40
18	Integration of Eclipse symbols and markers for Java entities	0.87	1.0	0.30
19	Adding relations using previews for selected entities	0.81	1.0	0.25
20	Number and selection of 'expand' and 'collapse members' commands for types via context menu	0.43	0.5	0.50
21	Inclusion of a tag cloud in type boxes	-0.03	0.0	0.77
22	Visualization of tags in an overlay	0.29	0.5	0.75
23	Highlighting of tags by selecting types in the tag overlay	0.35	0.5	0.67
24	Highlighting of types by selecting tags in the tag overlay	0.41	0.5	0.66
25	Pinning tags from the overlay to the diagram	0.68	1.0	0.46
26	Visualizing the influence of a tag using a 'fan' in the background of the diagram	0.85	1.0	0.24
27	Adding notes	0.56	0.5	0.44
28	Opening a Java Editor by double-clicking a diagram entity	0.96	1.0	0.13
29	Use of selection-dependent buttons (e.g. 'Close' and 'Change Color')	0.54	0.5	0.50
30	Limitation to one box per Java type	0.33	0.5	0.67

**Table 6.2:** Usability of 16 Code Gestalt features on a scale from -1 ('not useful') to 1 ('very useful')

each, so that each set would only include diagrams from one half of our test group. That way, we prevented that testers reviewed their own diagrams.

The survey was concluded by a list of suggested features, which users made during the test sessions and the open-ended questions of the test questionnaire. The participants were asked to rate each feature on a five-point Likert scale. For further details refer to appendix D.1.

Participants rated the importance of proposed features.

LimeSurvey recorded 14 responses from the 16 testers we invited to take the survey between December 16<sup>th</sup>, 2010 and January 2<sup>nd</sup>, 2011.

### Comparison of sketches with Code Gestalt diagrams

Code Gestalt diagrams are considered slightly clearer ( $M = .24$ ;  $Mdn = .33$ ,  $s = .53$ ) and better suited for documentation ( $M = .23$ ;  $Mdn = .33$ ,  $s = .51$ ) than sketches. They are however not perceived as offering a better means of understanding source code ( $M = -.01$ ;  $Mdn = 0$ ,  $s = .52$ ) or better task support ( $M = .04$ ;  $Mdn = 0$ ,  $s = .52$ ) than sketches.

Code Gestalt diagrams have minor advantages over sketches.

### Qualitative feedback

12 testers left feedback in an open-ended question asking the participants to draw the balance on the four pairs they evaluated and describe in their own words which visualizations were more appropriate under which aspects. The testers were encouraged to give feedback beyond the four comparison categories of the survey.

We asked testers to compare the pairs in their own words.

Four participants noted that sketches were more suitable for conveying ideas and unusual circumstances, since Code Gestalt was always bound closely to the actual implementation, but sketches could omit implementation details and take a more abstract point of view on the code base. The clearness of Code Gestalt diagrams was preferred over sketches by four testers. In this context testers noted that

Some positive feedback did not only consider the individual diagram, but also interactions possible using Code Gestalt.

the undo and editing features of the tool made it easy to keep a diagram clear, even if the diagram creator had to make corrections to the diagram after an error. Three participants expressed their preference for sketches when it came to the possibility of annotating the diagram. However, two testers mentioned that the readability of Code Gestalt annotations was better than those from hand writing. Sketches were more capable to portray the sequence of actions according to two testers. Interestingly, two participants considered sketches to be more precise and expressive than Code Gestalt diagrams and two users made the opposite statement. Code Gestalt was also deemed more appropriate for documenting code by two participants. Two testers mentioned the high usability and the speed advantages of Code Gestalt each, although this was not directly prompted by the question.

Individual comments are concerned with relation customization, clutter, and comprehensibility.

We also got a number of individual remarks: One participant attributed the difference between the diagram pairs more to the skill of the creators than the capabilities of the tools used to make them. The visualization options for terms were positively attributed to Code Gestalt by one tester. Another user mentioned that Code Gestalt type boxes tended to be cluttered by unnecessary members, as users did not bother to remove unneeded members. The complementary use of both systems was preferred by one participant. While one tester felt that Code Gestalt made it hard to detect coherences, another user stated that Code Gestalt diagrams were more comprehensible than sketches. One tester noted that the flexibility of relations in sketches were only preferable, because Code Gestalt did not allow users to label them, yet.

### Requested features

Users were asked to rate proposed Code Gestalt features for importance.

We compiled a list of features that were requested by testers during the initial study and its questionnaire. We presented the list in the online survey and asked the participants to rate the importance of each feature on a five-point Likert scale from 'completely insignificant' to 'very important'. Rating each feature was optional, thus we recorded varying response counts, identified by  $n$  in table 6.3.



Feature Description	<i>n</i>	<i>M</i>	<i>Mdn</i>	<i>s</i>
Automatic search for and preview relations to elements that are not yet included in the diagram	13	0.69	1.00	0.43
Labeling of relations	14	0.61	0.50	0.45
New relation type 'override/implements' between methods in a type hierarchy	14	0.54	0.50	0.31
Make tag cloud at bottom of type box optional (default: disabled)	13	0.54	0.50	0.52
New relation type 'dependence' between types	9	0.50	0.50	0.35
'Open Call Hierarchy' and 'Open Type Hierarchy' commands in diagram context menu	14	0.39	0.50	0.35
'Remove members from inverse selection' command in context menu	14	0.29	0.50	0.54
Add members to types using a text box with incremental search	13	0.23	0.50	0.53
Manual relation drawing tools	14	0.21	0.50	0.80
Drag-and-drop of members to any location and automatically add them to the correct type box	14	0.18	0.50	0.50
Framework specific relations (e.g. from a button to an action)	10	0.30	0.25	0.35
Allow highlighting of tags/types in tag overlay for multiple elements	10	0.25	0.25	0.42
Drag-and-drop source code to diagram	14	0.04	0.25	0.66
New relation 'Access' from methods to fields	12	0.17	0.00	0.33
Support 'Link with Editor' feature of Eclipse Package Explorer	11	0.17	0.00	0.50
JavaDoc tool tips	11	0.14	0.00	0.39
Preview relations during drag-and-drop from Package Explorer	13	0.12	0.00	0.58
Full support for anonymous and local types	11	0.09	0.00	0.38
Assign random colors to thematic relation fans on creation	13	0.08	0.00	0.49
Routing-algorithm for relations (to minimize overlap)	11	0.05	0.00	0.42
'Send to Diagram' command in context menu of Package Explorer and Java Editor	14	0.04	0.00	0.66
'Find' command for diagram	13	0.00	0.00	0.50
Show thematic relation fan in tag overlay	12	0.00	0.00	0.37
Show context sensitive controls (i.e. 'Expand All', 'Change Color') on mouse-hover (instead of selection)	14	-0.04	0.00	0.57
Automatic layout (optional)	12	-0.04	0.00	0.62
'Find' command for tag overlay	14	-0.07	0.00	0.39
Show tag overlay only while holding down a key (quasi-mode)	14	-0.25	0.00	0.51
Rich Text for sticky notes	14	-0.50	-0.50	0.48

**Table 6.3:** Importance of suggested features for future versions of Code Gestalt on a scale from -1 ('completely insignificant') to 1 ('very important')

Users want more flexibility with regard to relations and more live previews.

The feature with the highest importance was the live preview of relations to types not already included in the diagram. Many testers told us, they wanted this feature, so they did not need to use Eclipse's type hierarchy and call hierarchy views anymore, which introduced a lot of time overhead, when using Code Gestalt. Moreover, users wanted additional relation types and customization options, as well as additional automation and IDE integration. There was also a high demand for making the type box tag cloud an optional element and disable it by default. This was expected, since we noticed during the user study that users tended to ignore the tag clouds and use the tag overlay instead, when they wanted access to the vocabulary of the source code.

Explicit searching was not a feature in high demand.

Interestingly, our users did not put much weight on a traditional search mechanism. This was unexpected, as related work suggested this would be a feature in high demand. The mixed results regarding automatic layout and routing features are however in line with our expectations and the design premise of Code Gestalt.

### 6.3.5 Summary

The user study indicates that most of our design goals were met by the Eclipse implementation of Code Gestalt.

SUS indicated good usability.

From the SUS score and the individual feature evaluation we gather that Code Gestalt has good usability and that the implemented features are perceived as useful by our target group. The null hypothesis regarding task completion times and error counts could not be rejected for the majority of the test conditions. This might indicate that our test group was too small, however, the significant results we found, speak another language.

Code Gestalt tends to improve correctness at the cost of time.

We see indications for Code Gestalt to improve correctness at the cost of time overhead. This is in line with the findings of Park and Jensen [2009]. The easiest of the the four tasks (#4, which had the highest completion rates and lowest completion times) was significantly faster processed by users who sketched the solution. It should be noted that an

overhead is to be expected, since Code Gestalt users have to switch between editors and views in the IDE in order to create the visualization, while pen and paper allow participants to work on the visualization in parallel to using the IDE. The completion times suggest that the overhead is getting less relevant for more complex tasks, where Code Gestalt's ability to support the user in finding key code artifacts plays a more important role. Most interestingly, the subjective perception of participants was inverse to our quantitative results: While only two users stated in an open ended question that sketching was faster than Code Gestalt, six testers stated the opposite.

Code Gestalt significantly improved the solution correctness for tasks #1 and #2. However, for task #3 sketching was more successful in revealing important call relations. The comments gathered during the sessions and from the open-ended questions lead us to believe that live previews for relations to non-included code artifacts would have increased efficiency of Code Gestalt users. Many testers reported independently from another that they would have wanted this feature, so they could have built the diagram without the IDE views to find call and inheritance relations. This is backed up by the importance assigned to the respective features in table 6.3.

We believe that Code Gestalt is in fact competitive to sketching, but has a different trade-off between completion time and correctness. The competitiveness is also strongly agreed upon by the testers (see #13 in table 6.1). We think, we can greatly improve completion rates and times by providing additional live previews as outlined in the previous section.

The usefulness of the newly introduced SV is supported by the results of the user survey. The thematic relations and the pinning interaction are rated 'very useful' ( $Mdn = 1.0$ ), while the tag cloud and the highlighting interactions are rated 'useful' ( $Mdn = 0.5$ ). Qualitative feedback from the study indicates that the tag overlay and thematic relations in fact support programmers in understanding unknown source code.

The time overhead can be reduced by providing more live previews.

Code Gestalt is a competitive to sketching.

The tag overlay and thematic relations support code understanding.

The created SVs have slight advantages over sketches.

Our second survey investigated the expressiveness of Code Gestalt diagrams without the backing of an IDE in direct comparison with sketches. The results show that our users slightly prefer Code Gestalt diagrams over sketches as far as clearness and suitability for documentation are concerned. However, for understanding source code and the support of concrete tasks we have not detected a tendency that would favor one system over the other. These results are encouraging, when we remember that pen and paper are the most often used SV solution (see chapter 3).

The automation features did not take control away from the user.

The features requested by our testers partially consist of features that were part of our design (like the live preview of relations to non-included entities), but had to be cut for time and framework limitations. That our users ask for more automation and system assistance suggests that our implementation was successful in giving the user control, while still automatically searching for options and providing previews. Features that would automate diagram creation at the cost of user control were rated less important.

Multiple terms might be more appropriate for identifying concepts.

The participants also requested features that expand the tag overlay, especially the selection of multiple elements for cross-referencing. This is interesting for two reasons. First, the tag overlay and its highlighting features seem to be natural and intuitive enough for users to internalize the concept. Second, it suggests that a theme or concept is more precisely captured by more than one term. In that case one could also consider thematic relations that are based on multiple terms and logical compositions thereof (AND, OR, etc.).

We identified a case of feature creep in our system.

Finally, we learned a valuable lesson from our users' wish to remove the tag cloud at the bottom of a type box. This UI element was a typical case of *feature creep*. Our paper prototype had no such tag cloud, however it managed to end up in the final version of Code Gestalt, as we were computing the tag cloud for the tag overlay anyway, and thought, without justification, it would be an interesting addition to the type box.

## Chapter 7

# Summary and future work

*“I wanted to write that my work consists of two parts: of the one which is here, and of everything which I have not written. And precisely this second part is the important one.”*

—Ludwig Wittgenstein

In this thesis we presented the SV tool Code Gestalt and its development process. Section 7.1 discusses our approach, implementation, and findings. In section 7.2 we will take a look at the next steps of our research and the open questions that remain to be answered.

### 7.1 Summary and contributions

Visualizing source code is an everyday task for many developers. However, related research and our own observations indicate that software visualization tools are not as widely used as one might expect. In a user survey we found that many programmers use traditional sketching techniques rather than SV tools, although the analog nature of sketches is adverse to modern communication and version control systems.

We investigated sketching as means of code visualization.

SV tools must be carefully designed.

The reasons for users to prefer sketching over SV tools are manifold, but we identified some key aspects from our own investigations and related work. Many SV tools take away control from the human user, thus creating SVs that do not represent the developers concept of the visualized code base. Where the user is able to focus on important aspects of a code base, most SV tools are quite indifferent and cannot emphasize regions of interest. Also, speed, IDE integration, and usability are key components in the acceptance of an SV tool.

Code Gestalt allows users to harness source code vocabulary for code exploration.

We designed Code Gestalt to meet these design goals, most prominently developing a new SV that would allow users to create landmarks of focus to emphasize important regions of their diagram. The visualization was also geared toward helping users explore the source code not only by following structural relations (such as call and inheritance relations), but themes and concepts. Our approach aimed at mining the vocabulary of the source code for such thematic information and make it accessible for the user of the SV tool.

Familiarity with class diagrams, IDE integration, and live previews ease learning curve.

To ground the new visualization in a familiar environment, we designed it as augmentation to class diagrams. Also, we designed the user interface in a way that would automatically assist the user whenever possible, without taking away any control over the scope and layout of the class diagram. We realized this by working with live previews that are displayed in context to the current selection. The user immediately sees, what expansion options are available to her and can decide to convert any number of previews into persistent parts of the diagram or ignore them.

The tag overlay and thematic relations are the center pieces of our design.

The design of Code Gestalt was refined twice using a paper prototype and a Silverlight mock-up. During this process, we developed two dependent visualizations based on the vocabulary of the source code:

1. *Tag Overlay*: a layer on top of a class diagram to visually search the vocabulary of the source code and to detect thematic similarities between types

2. *Thematic Relations*: a new relation type for class diagrams, connecting those types who share common terms in their identifiers

We implemented Code Gestalt as plug-in for Eclipse. This implementation was evaluated in a two-step user study. The results from the study suggest good to excellent usability of Code Gestalt in general according to Bangor et al. [2009]. Moreover, we can establish competitiveness of Code Gestalt compared to sketching with pen and paper.

Code Gestalt is competitive with sketching

Testers missed fewer types and attributes in their visualizations compared with sketches, however they missed more call relations and needed more time to create a diagram for simple tasks. We found evidence that the tag overlay and thematic relations help users in understanding source code by providing ways to harness the human intelligence present in the naming of source code identifiers.

Code Gestalt reduced errors and help users understand unknown code.

When we directly compared Code Gestalt diagrams with sketches, testers attributed better clearness and suitability for documentation to Code Gestalt diagrams. Testers did not have clear preference between pen and paper sketches and Code Gestalt diagrams with regard to understandability and use for practical task support.

Code Gestalt diagrams are preferred over sketches for clearness and readability.

## 7.2 Future work

Code Gestalt was well received by our test group. The user test revealed potential for improvement and poses new research questions.

### 7.2.1 Implementation

We gathered suggestions for additional features and feature changes from our participants during the test session and the following survey. The collection of suggestions was then presented to all testers to rate them for importance (see

Users suggested numerous new features.

section 6.3.4). This list is a good indicator for the direction of Code Gestalt's development.

Users would like Code Gestalt to dispense with Eclipse's hierarchy views.

Some of the requested features are identical with what was part of the original design and could not be implemented due to time constraints. We are very interested in bringing Code Gestalt on par with the feature set of Relo, so users can use Code Gestalt without the assistance of Eclipse views to find call and inheritance relations to artifacts not already visualized in the code. We believe this would greatly reduce the time overhead our testers experienced when working with Code Gestalt.

### 7.2.2 Diagram customization

Additional customization options should borrow from sketching.

Users requested the addition of further customization features and better annotation tools. While it is relatively straight forward to add more sophisticated customization options known from text and image processors, the reaction of testers to the suggested Rich Text sticky notes is a clear indicator that this kind of customization is not desired. Instead, we should be looking for customization techniques inspired by what users do when sketching.

We need more detailed data about sketches.

A more detailed analysis of how users annotate sketches is required to find appropriate metaphors and interactions to integrate them with Code Gestalt. An alternative approach is the direct integration of hand drawings as in Lichtschlag and Borchers [2010]. This must be done carefully though, because we do not want to sacrifice the clearness and consistency currently found in Code Gestalt diagrams.

### 7.2.3 Scalability

Code Gestalt's scalability has not yet been assessed.

We used a relatively small code base (ten files) in the evaluation of Code Gestalt. For this project size, the tag overlay is appropriate as our users gave it good usability ratings and were indifferent to a dedicated search feature.



We estimate this to change for larger code bases. The naive approach of positioning tags at the center of gravity might lead to great disparities between the position of a tag and the location of the types where the corresponding concept or theme is implemented. Such disparities would break the map metaphor of the tag overlay, as tags would no longer correctly label areas of interest. We expect to resolve these issues with more sophisticated tag metrics, such as *tf-idf*<sup>1</sup> [Spärck Jones, 1972], and clustering algorithms.

Clustering algorithms and *tf-idf* could improve tag overlay.

#### 7.2.4 Further evaluation

Although our user study has established the usability of Code Gestalt and its new concepts, our setup did not allow us to gather quantitative data on the question, if the tag overlay and thematic relations help users gain more insight into the given code base compared to traditional SVs, like class diagrams. A second user study might be helpful, in which we compare a modified version of Code Gestalt with disabled tag overlay against the full system. In such a study we would give testers tasks that require them to answer conceptual questions about an unknown code base.

Separately investigate Code Gestalt's influence on code understanding.

#### 7.2.5 Multiple selection in tag overlay

An interesting suggestion from our testers was the support of multiple selection in the tag overlay to create more refined highlights. E.g., by selecting multiple tags all types would be highlighted according to their multiplied weights. This feature request suggests that concepts and themes might be better represented using multiple terms. If that is the case, we should investigate thematic relations based on multiple terms and evaluate, if they capture concepts better than the current single-term thematic relations.

Users want to be able to create heat maps from multiple tags.

---

<sup>1</sup>*term frequency-inverse document frequency* determines the weight of tags not only by term frequency in an individual document, but also by considering its importance in relation to other documents.

### 7.2.6 Additional metrics

Other metrics could hold similar potential as the code vocabulary.

Code Gestalt incorporates only a very limited number of metrics and visualizations thereof. Our results show that the tag overlay and thematic relations allow users to manage the information distilled from source code vocabulary. There are however numerous metrics to detect code smells (indicators for code in need of improvement) and other properties that might be equally useful for developers in their endeavor of understanding and illustrating source code.

Use Code Gestalt's design as pattern for new SVs.

Those metrics could be integrated with Code Gestalt by following the design of the tag overlay and thematic relations. The research question at hand is, if our design methodology of using overlays and live previews to keep this new source of information manageable and usable for the user. If so, it might be abstracted to become a successful pattern for building new SVs.

## Appendix A

# Additional online survey materials

*“It’s time to reapprciate the original software:  
paper.”*

—Dale Dauten

In chapter 3 we present the results of the online survey in context of our development of Code Gestalt. This appendix contains the complete set of survey questions and the detailed statistical analysis that led to the results presented in this thesis (see section 3.2).

### A.1 Survey questions

In the following we reprint the questions used in the online survey discussed in chapter 3. The survey consists of five parts:

The questions of the survey are divided in five groups.

1. Background information
2. Usefulness of software visualizations in general
3. Visualization software

4. Manual visualization
5. Visualizations in documentation

Please note that the writing of the survey was less formal than the usual style of this thesis. Users were able to input data using HTML form elements like text boxes, drop-down lists, check boxes, and radio buttons. We do not reproduce these UI elements, but give details on the answers and modalities for each question. Unless otherwise noted, participants had to select one of several predetermined answers.

### **A.1.1 Background**

To get started I'd like to ask some general questions on your background as a programmer.

**Voluntary e-mail contact:** Free text (optional)

Help: This survey contains some open ended questions. Sometimes the answers to these questions may inspire a follow up question or the need for clarification. If you provide an e-mail contact it will be exclusively used for that single purpose and deleted at the end of the evaluation process.

**What is your current occupation?** Programmer; Student; Researcher; Teacher; Other (mandatory)

Help: If the occupation you want to select is not listed, choose 'Other' to manually enter one.

**What is your gender?** Female; Male (mandatory)

Help: Please select the corresponding sex.

**What is your age?** Numerical input (mandatory)

Help: Please enter your age in years.

**For how many years have you gathered programming experience?** Numerical input (mandatory)

Help: Please input the number of years you have been programming regularly.

**How regularly do you read source code in your current occupation?** Daily; Not daily, but several times per week; At least once a week; At least once a month; At least once a quarter; Less than once a quarter, but occasionally; Never (mandatory)

Help: Please give a rough estimate how often you are occupied with reading source code.

**How much of the code you deal with is unknown to you by the time you read it?** < 20%; 20%–40%; 40%–60%; 60%–80%; >80% (mandatory)

Help: E.g., this can be code from a colleague who works on another project, students (e.g. correcting exercises), third party libraries, or legacy projects.

**Visualizations** In this subsection I'd like to ask you about what you think about common and some academic/experimental visualizations of software.

**Please rate the following common visualizations of source code regarding their usefulness.** Useful; Not useful; Don't know/never used (optional) for each of the following: Class Diagram; Call Graph; Flowchart; Data Flow Diagram; Software Layer Diagram; Sequence Diagram; Package Diagram; Other (refer to next question)

Help: In the following Wikipedia articles you can find examples of the diagram types mentioned above (links open in new window): **Class Diagram:** Classes are represented as boxes, usage relations as labeled lines and inheritance relations as arrows. **Call Graph:** Functions/methods are represented as boxes and calls as arrows. **Flowchart:** Beginning and end of a process are represented as rounded rectangles, Input/Output as parallelograms, and data flow as labeled arrows. **Data Flow Diagram:** The arrows indicate the flow of data between files/databases (open boxes). **Software Layer Diagrams:** e.g. visualizing the OSI model or an operation system. This Software Layer Diagram shows how the layers of an operating system encapsulate the functionality of the layers above and below. **Sequence Diagram:** The diagram illustrates the interaction/messages (labeled horizontal arrows) between objects/processes (dotted lines) over time (negative y-axis) and highlights the active phases of each object (vertical bars). **Package Diagram:** are represented as boxes, nested packages are represented as nested boxes. Usage relations are shown as labeled arrows.

**If you evaluated the usability of the ‘Other’ common visualization above, what is it called?** Free text (mandatory, only shown when ‘Other (refer to next question)’ was rated in the previous question)

Help: Please enter the name of the visualization and if possible a link to a sample visualization or a more detailed description of what the visualization looks like.

**Please rate the following visualizations that have been proposed by researchers in recent years based on your first impression by looking at them. Consider whether you think these diagrams are useful in understanding source code.** Useful; Not useful; Don’t know/never used (optional) for each of the following: CodeCity; Thematic Software Maps; 3d Relation Diagram; CallStax

Help: Please just skim quickly over the example pictures. Give an answer based on your first impression. **CodeCity:**

In a CodeCity classes are represented as buildings and packages as districts. The size of the buildings represents statistical information such as number of attributes and number of methods. **Thematic Software Map:** Classes are represented as elevations proportional to their size in lines of code. The distance between classes is determined by their similarity in source code vocabulary. **3d Relation Diagram:** (German website, but English pdf-papers at bottom of page) 3d Relation Diagrams use three dimensions to layout classes, interfaces and packages. Usage, inheritance and inclusion relations are represented by lines, arrows and nested geometry. **CallStax:** CallStax' show the paths of a function call tree as separate 'towers'. Individual functions are represented by blocks of different colors.

**Comments:** Free text (optional)

Help: Did I miss an important visualization type? Do you want to comment on why you think certain visualizations are more useful than others? If so or if you have any other comment on the topic of visualization techniques in general, please feel free to leave a comment right here. All comments will be read and considered.

### A.1.2 Visualization software

The next subsection covers questions on your experience as a user of software visualization systems.

**How often do you use software tools (within or outside your IDE) to create visualizations of your software project or certain source code artifacts like classes?** Daily; Not daily, but several times per week; At least once a week; At least once a month; At least once a quarter; Less than once a quarter, but occasionally; Never (mandatory)

Help: Only give a rough estimate on how often you use software to visualize source code.

**SV users**

The questions in this subsection were only presented to participants, who indicated to use SV tools in the previous question.

**What visualizations do you usually create with a software visualization application?** Class Diagram; Call Graph; Flowchart; Data Flow Diagram; Software Layer Diagram; Sequence Diagram; Package Diagram (mandatory, multiple answers possible)

Help: Please check only those visualizations you actually create regularly.

**For what purpose do you create visualizations of your project using visualization software?** Internal documentation, Documentation for public release, Project presentation, Personal use (source code understanding), Project management; Complexity control; Code reviews; Quality assessment (mandatory, multiple answers possible)

Help: Check only those scenarios you create visualizations for on a regular basis.

**What software do you use most often when creating visualizations?** Free text (mandatory)

Help: This may be an IDE with visualization features, an IDE plug-in or a standalone application. If you use a suite of several smaller tools that need to be used in combination please state the name of the suite.

**Does the above tool work seamlessly and smoothly with your source code and projects?** Mainly yes; Mainly no (mandatory)



Help: Perfectly working tools should require no effort at all to load and parse your code. Very badly working tools will require you to take great lengths in making your project and code compatible with it or it withholds options and functionality e.g. because your project is created with the IDE of another vendor.

**Does the above tool create a visualization to your satisfaction?** Mainly yes; Mainly no (mandatory)

Help: The satisfaction with the visualization tool should include several aspects. Some of the questions you might ask yourself are: Is the visualization customizable or is it random? Is the use of colors tasteful and useful or annoying and confusing? Is the layout clear or confused? Can the visualization be easily printed and exported? Is the visualization understandable or does one need 'rocket science' to make heads and tails of it?

**Does the above tool create the visualization automatically?** Automatic; Semi-automatic; Manual (mandatory)

Help: A short elaboration on the three options: **Automatic** tools should be able to create the complete visualization without user input, although they might offer options that allow the user to change some global features of the visualization or allow for manual edits after the creation of an automatically generated diagram. **Semi-automatic** tools should have a basic understanding of the code and offer features to generate visualizations of at least basic software artifacts, although the user must e.g. select manually what pieces to include in a diagram or how to layout the single parts. **Manual** tools are basically drawing applications. Although they might have some basic support by providing predefined shapes and layouts, they are not capable of parsing a project or source code in order to create any part of a diagram automatically.

**How much time do you normally spend using the tool mentioned above on creating a visualization of your**

**project?** < 5 min, 5–10 min, 15–30 min, 30–60 min, > 60 min (mandatory)

Help: Please give a rough estimate on how much time you spend on creating one visualization.

### **SV non-users**

This question was given to participants who indicated to never use SV tools.

**Why do you not use a visualization software?** I did not know that there are ways to visualize source code; I did not know that there are programs for creating such visualizations; I have no need to create such visualizations; There is no tool that can create the visualizations I need; The visualization tool I need is not affordable; Usability issues; Undesired/unsatisfying results; Too time consuming (mandatory, multiple answers possible)

Help: Please check only those answers that specifically hinder you from using a visualization software, not shortcomings that do not usually keep you from using software.

### **All participants**

All participants were give the following question.

**Comments:** Free text (optional)

Help: Please leave any comments that you think are important with regard to visualization software.

### A.1.3 Manual visualization

This subsection takes a look at how you visualize source code using sketches.

**How often do you use sketches (paper, white board, flip chart, Tablet PC etc.) to create visualizations of your software project or certain source code artifacts like classes?**

Daily; Not daily, but several times per week; At least once a week; At least once a month; At least once a quarter; Less than once a quarter, but occasionally; Never (mandatory)

Help: Only give a rough estimate on how often you use sketches to visualize source code.

#### Sketching participants

The following questions were only given to those participants, who indicated in the previous question to sketch at least occasionally.

**What aspects of your software project or code artifact do you usually cover in your sketches?**

Class hierarchy; Class dependency; Membership; Data flow; Software layers; Function/method calls; Framework/package/namespace hierarchy; Framework/package/namespace dependency; Sequence and timing (mandatory, multiple answers possible)

Help: Please check only those properties you actually sketch regularly.

**What materials do you use to create sketches?** Paper; Pen/pencil; Multiple colors; White board; Flip chart; Ruler; Set square; Compass; Eraser/sponge; Computer graphics tablet; Magnets; Sticky notes (mandatory, multiple answers possible)

Help: Please check only those materials you use regularly to create sketches.

**For what purpose do you create sketches of your project or software artifacts?** Internal documentation; Documentation for public release; Project presentation; Personal use (source code understanding); Project management; Complexity control; Code reviews; Quality assessment (mandatory, multiple answers possible)

Help: Check only those scenarios you create sketches for on a regular basis.

**How much time you usually spend on creating a sketch?** < 5 min; 5–10 min; 15–30 min; 30–60 min; > 60 min (mandatory)

Help: Please give a rough estimate on how much time you spend on creating one sketch.

### **Non-sketching participants**

The following questions were only given to those participants, who indicated to never sketch.

**Why do you not use sketches to visualize your code?** I did not know that there are ways to visualize source code; I did not think of doing sketches to visualize code; I have no need to create such sketches; I can not sketch the visualizations I need (e.g. they are too complicated); I can not afford the materials; I find sketching too hard to do; Undesired/unsatisfying results; Too time consuming (mandatory, multiple answers possible)

Help: Please check all answers that hinder you from drawing sketches to illustrate your code.

## All participants

This question was again given to all participants.

**Comments:** Free text (optional)

Help: Please leave any comments that you think are important with regard to sketches of code.

### A.1.4 Documentation

In this subsection we will discuss how visualizations of code are used in source code documentation.

**Do you initially consult the documentation of a previously unknown project to find visualizations in order to understand it?** Mainly yes; Mainly no (mandatory)

Help: Also consider such cases where you wanted to look at a visualization, but there was no documentation to start with.

**Do you use visualizations of a documentation regularly when you are more familiar with a project?** Mainly yes; Mainly no (mandatory)

Help: This does not only include foreign code you came to understand, but also your own projects.

**Comments:** Free text (optional)

Help: Please leave any comments that you think are also important in your use of documentation visualizations.

## A.2 Results and analysis

In this section we offer the reader a detailed report on the data gathered from the survey and its statistical analysis that is omitted from section 3.2 for better readability.

### A.2.1 Background

The following data are the basis for the results presented in section 3.2.1.

A majority of the participants were male students.

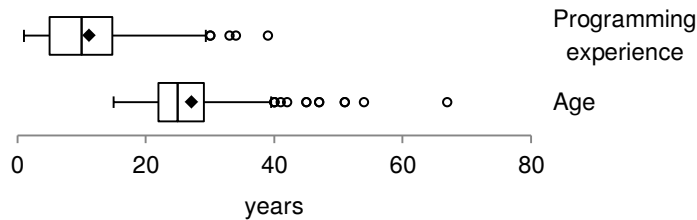
**What is your current occupation?** Roughly half of the 128 participants were students (67 participants or 52.34%). The second biggest group were programmers (43 / 33.59%) followed by researchers (11 / 8.59%). Two participants were teachers (1.56%), four participants came from other professions in the field of software development (3.12%).

**What is your gender?** Only two participants were female (1.56%), the rest was male (126 / 98.44%).

The participants cover a wide range of age and programming experience.

**What is your age?** Figure A.1 illustrates the distribution of the participants' age and programming experience. Participants were 27.17 years old on average ( $Mdn = 25y, s = 8.39y$ ). There were 13 participants who were 40 years of age and older. These are indicated as outliers in the box plot diagram.

**For how many years have you gathered programming experience?** Our sample group had a mean of 11.18 years of programming experience ( $Mdn = 10y, s = 7.41y$ ). Five participants had 30 or more years of programming experience. These are indicated as outliers in the box plot diagram in figure A.1.



**Figure A.1:** Box plots of the survey participants' age and programming experience ( $N = 128$ ).

**How regularly do you read source code in your current occupation?** Table A.1 shows the distribution of how frequently our participants read source code. Almost two-third (62.50%) of the participants were reading source code on a daily basis.

Two thirds of the participants read source code every day.

Answer	Count	Percentage
Daily	80	62.50%
Not daily, but several times per week	32	25.00%
At least once a week	14	10.94%
At least once a month	1	0.78%
At least once a quarter	1	0.78%
Less than once a quarter, but occasionally	0	0.00%
Never	0	0.00%
$N$	128	100.00%

**Table A.1:** Frequency of source code reading

**How much of the code you deal with is unknown to you by the time you read it?** The distribution of the amount of unknown code read by our sample group is shown in table A.2. Only every sixth participant (17.97%) was working with 60% or more unknown source code, but every third (33.59%) dealt with 20% or less.

Only every sixth participant works with 60% or more unknown code

Answer	Count	Percentage
< 20%	43	33.59%
20% – 40%	30	23.44%
40% – 60%	32	25.00%
60% – 80%	7	5.47%
> 80%	16	12.50%
<i>N</i>	128	100.00%

**Table A.2:** Percentage of unknown source code dealt with

### A.2.2 Visualizations

The following data are the basis for the results presented in section 3.2.2.

**Please rate the following common visualizations of source code regarding their usefulness.** The mean scores for all participants were presented in 3.2.2. In the following we will focus on how the usefulness varied among the participants, when we take their background into consideration:

- perceived usefulness between occupations
- correlation with programming experience
- correlation with code reading regularity
- correlation with percentage of unknown code read

Students rated call graphs and software layer diagrams less useful than programmers.

Since our data for individual occupations are not normally distributed (*Kolmogorov-Smirnov* and *Shapiro-Wilk* tests for each occupation significant with  $p < .05$ ), we used a non-parametric *Kruskal-Wallis* test to determine significant differences in the perceived usefulness (see table A.3). Call graphs were better received by programmers ( $M = 0.33, s = 0.104, Mdn = 0$ ) than students ( $M = 0.01, s = 0.094, Mdn = 0$ ). This difference is significant with  $U =$



$-1123.500, z = -2.085, p < .05, r = -.20$ . Similar differences occurred for the rating of software layer diagrams,  $U = -1105.000, z = -2.201, p < .05, r = -.21$ . All diagrams but the package diagrams received positive mean scores among all occupations. The package diagram received negative mean usefulness ratings from all groups.

Visualization	$H(4)$	$p$	Significance
Call graph	10.74	.02	$p < .05$
Class diagram	2.33	.70	<i>ns</i>
Data flow diagram	8.29	.07	<i>ns</i>
Flowchart	0.94	.94	<i>ns</i>
Package diagram	3.72	.47	<i>ns</i>
Sequence diagram	3.53	.50	<i>ns</i>
Software layer diagram	9.86	.03	$p < .05$

**Table A.3:** Kruskal-Wallis test comparing perceived usefulness of common visualizations between occupations. We found significant differences for call graphs and software layer diagrams. In both cases, programmers perceived the diagrams to be more useful than students.

When looking at the data by programming experience in years, we used *Kendall's* and *Spearman's* tests to find significant two-tailed correlations as shown in table A.4. We found a significant negative correlation  $\tau = -0.16, p < .05; r_s = -.19, p < .05$  between programming experience and the perceived usefulness of class diagrams.

The perceived usefulness of class diagram decreases with growing programming experience.

The responses can also be organized by the regularity by which the participants read source code. Again, partitioned by the regularity of code reading the data are not normally distributed. Thus we use Kruskal-Wallis (table A.5) and *Janckheere-Terpstra* (table A.6) tests for analysis.

We have a highly significant result for software layer diagrams with  $H(4) = 11.53, p < .01$ . The *Jonckheere-Terpstra* test shows a highly significant trend, indicating that the more often developers read source code, the less useful software layer diagrams become,  $J = 1990, z = -2, 81$ . Similarly, we detected a significant trend for package diagrams,  $J = 1751, z = -2.004$ , however the Kruskal-Wallis test fails to detect significant differences between the

The more frequently developers read source code, the less useful software layer diagrams are perceived.

Visualization	Kendall's $\tau$		Spearman's $r_s$		Significance
	$\tau$	$p$	$r_s$	$p$	
Call graph	.05	.51	.07	.46	<i>ns</i>
Class diagram	-.16	.03	-.19	.03	$p < .05$
Data flow diagram	.10	.15	.13	.14	<i>ns</i>
Flowchart	-.05	.53	-.06	.53	<i>ns</i>
Package diagram	-.06	.37	-.08	.37	<i>ns</i>
Sequence diagram	.02	.82	.02	.81	<i>ns</i>
Software layer diagram	.07	.33	.09	.31	<i>ns</i>

**Table A.4:** Correlation between programming experience and perceived usefulness of common visualizations. Programming novices perceive class diagrams significantly more useful than veterans.

Visualization	$H(6)$	$p$	Significance
Call graph	8.00	.06	<i>ns</i>
Class diagram	6.67	.18	<i>ns</i>
Data flow diagram	2.26	.78	<i>ns</i>
Flowchart	8.00	.06	<i>ns</i>
Package diagram	6.56	.12	<i>ns</i>
Sequence diagram	2.31	.80	<i>ns</i>
Software layer diagram	11.53	.01	$p < .01$

**Table A.5:** Differences in perceived diagram usefulness between participants by code reading regularity. We detected significant differences between groups for software layer diagrams.

groups. Also, the Kruskal-Wallis results for call graphs and flowcharts are interesting, but fail being considered significant. Moreover, we cannot report any significant trends for them.

The percentage of unknown code encountered by the participants did not significantly influence the perceived usefulness of any SV.

A similar investigation, grouping participants by the percentage of unknown code read, did not reveal any significant results (see tables A.7 and A.8). This was a bit of a surprise to us, since we expected that users that need to analyze code would have learned to gain more information from diagrams than those users who primarily deal with known code.

Visualization	<i>J</i>	<i>z</i>	<i>p</i>	Sig.
Call graph	2091.5	0.51	.61	<i>ns</i>
Class diagram	2217.5	0.19	.85	<i>ns</i>
Data flow diagram	2262.0	0.37	.71	<i>ns</i>
Flowchart	2023.5	0.93	.35	<i>ns</i>
Package diagram	1751.0	-2.00	.05	$p < .05$
Sequence diagram	1990.0	-0.75	.45	<i>ns</i>
Software layer diagram	1645.5	-2.81	.01	$p < .01$

**Table A.6:** Trends in perceived diagram usefulness between participants by code reading regularity. Frequent readers of source code perceive software layer diagrams less useful than infrequent readers.

Visualization	<i>H</i> (4)	<i>p</i>	Significance
Call graph	4.20	.39	<i>ns</i>
Class diagram	1.89	.77	<i>ns</i>
Data flow diagram	3.39	.50	<i>ns</i>
Flowchart	4.05	.40	<i>ns</i>
Package diagram	3.80	.44	<i>ns</i>
Sequence diagram	8.08	.08	<i>ns</i>
Software layer diagram	1.57	.82	<i>ns</i>

**Table A.7:** Differences in perceived diagram usefulness between participants dealing with different percentages of unknown code. No significant differences were found. (*p* values are an approximation calculated with IBM SPSS 19)

The participants were allowed to indicate an additional type of common visualization, they felt was missing from the given list. 11 participants used this opportunity, but most entries were irrelevant. Examples include code highlighting, sketches, and visualizations covered in other parts of the survey. Relevant entries were *gource*<sup>1</sup>, *state charts*, and *use case diagrams*. Only individual responses for the usefulness of these visualizations were given, so the results are not usable for analysis.

The selection of common SVs was rather comprehensive.

<sup>1</sup><http://code.google.com/p/gource/>

Visualization	<i>J</i>	<i>z</i>	<i>p</i>	Sig.
Call graph	3317.5	1.10	.27	<i>ns</i>
Class diagram	3198.0	0.75	.45	<i>ns</i>
Data flow diagram	3075.0	-0.01	.99	<i>ns</i>
Flowchart	3258.0	0.90	.37	<i>ns</i>
Package diagram	2826.5	0.96	.34	<i>ns</i>
Sequence diagram	3111.0	0.36	.72	<i>ns</i>
Software layer diagram	3345.0	1.23	.22	<i>ns</i>

**Table A.8:** Trends for perceived diagram usefulness between participants dealing with different percentages of code. No significant differences were found. (*p* values are an approximation calculated by IBM SPSS 19)

**Please rate the following visualizations that have been proposed by researchers in recent years based on your first impression by looking at them. Consider whether you think these diagrams are useful in understanding source code.** Again, we investigated, how different groups perceive the usefulness of the presented SVs. Table A.9 shows the results of a Kruskal-Wallis test to find differences between participants of different occupations.

Visualization	<i>H</i> (4)	<i>p</i>	Significance
3d relation diagram	5.05	.29	<i>ns</i>
CallStax	1.12	.90	<i>ns</i>
Code city	2.42	.69	<i>ns</i>
Thematic software map	8.07	.08	<i>ns</i>

**Table A.9:** Kruskal-Wallis test comparing perceived usefulness of common visualizations between occupations. A Mann-Whitney test discovered a significant difference between students and researchers regarding thematic software maps.

Researcher find the thematic software map more useful than students.

Using a Mann-Whitney test we found a highly significant difference in the perceived usefulness of thematic software maps between students ( $M = -0.3, s = 0.78$ ) and researchers ( $M = 0.9, s = 0.91$ ),  $U = 169, z = -2.73, p < .01, r = -.24$ , indicating that researcher find the thematic software map more useful than students.

The programming experience does not significantly correlate with the perceived usefulness, as shown by the correlation tests in table A.10. Also, the regularity with which the participants read source code does not impact their rating of the diagrams significantly, the results of a Kruskal-Wallis test are shown in table A.11. Similarly, there were no significant differences in the perceived usefulness between participants grouped by the percentage of unknown source code read. The results from a Kruskal-Wallis test are shown in table A.12.

Programming experience, code reading frequency, and the percentage of unknown code dealt with do not significantly impact the perceived usefulness.

Visualization	Kendall's $\tau$		Spearman's $r_s$		Significance
	$\tau$	$p$	$r_s$	$p$	
3d Relation Diagram	-.09	.21	-.12	.19	<i>ns</i>
CallStax	-.02	.77	-.03	.78	<i>ns</i>
Code city	-.09	.21	-.12	.19	<i>ns</i>
Thematic software map	.11	.13	.14	.12	<i>ns</i>

**Table A.10:** Correlation between programming experience and perceived usefulness of experimental visualizations. No significant results were found.

Visualization	$H(6)$	$p$	Significance
3d relation diagram	1.14	.97	<i>ns</i>
CallStax	1.43	.95	<i>ns</i>
Code city	6.68	.48	<i>ns</i>
Thematic software map	1.05	.99	<i>ns</i>

**Table A.11:** Differences in perceived usefulness of experimental visualizations between participants by code reading frequency. No significant results were found.

### A.2.3 Visualization software

**How often do you use software tools (within or outside your IDE) to create visualizations of your software project or certain source code artifacts like classes?** Table A.13 shows the answers to this question. Half the participants from our survey use SV tools only several times per year or never.

Visualization	$H(4)$	$p$	Significance
3d relation diagram	4.89	.30	<i>ns</i>
CallStax	1.96	.75	<i>ns</i>
Code city	2.60	.64	<i>ns</i>
Thematic software map	4.70	.32	<i>ns</i>

**Table A.12:** Differences in perceived usefulness of experimental visualizations between participants by percentage of unknown code. No significant results were found.

Answer	Count	Percentage
Daily	5	3.91%
Not daily, but several times per week	9	7.03%
At least once a week	12	9.38%
At least once a month	14	10.94%
At least once a quarter	23	17.97%
Less than once a quarter, but occasionally	24	18.75%
Never	41	32.03%
$N$	128	100.00%

**Table A.13:** Frequency of SV tool usage. Half of our participants use SV tools very scarcely or never.

Users and non-users were given different questions.

Depending on the answer to this question, participants were given different follow-up questions. We will present the results for questions that followed for SV tool users (87 participants) first, then the question for non-users (41 participants).

### SV tool users

Class diagrams are by far the most often created SV.

**What visualizations do you usually create with a software visualization application?** For this question participants were allowed to choose any number of given SV types and up to one additional type using a text box. Again, we offered the seven common diagram types from section 3.2.2.

Note that the percentages in table A.14 do not sum up to 100%, since each user could select several SVs.

Answer	Count	Percentage
Class diagram	71	81.61%
Flowchart	19	21.84%
Sequence diagram	18	20.69%
Call graph	16	18.39%
Software layer diagram	13	14.94%
Data flow diagram	11	12.64%
Packet diagram	6	6.90%
Other	10	11.49%
<i>n</i>	87	100.00%

**Table A.14:** Most frequently created SV types using software tools

Other SV types given were relational database diagrams ( $n_r = 3$ ), state charts ( $n_s = 2$ ), and several individual diagrams.

**For what purpose do you create visualizations of your project using visualization software?** In order to determine the right use case scenario for Code Gestalt we wanted to know about the situations, in which SV tools were used. Again, participants were allowed to select multiple reasons, as shown in table A.15.

SVs are created mostly for internal documentation and personal use.

**What software do you use most often when creating visualizations?** The participants used a large variety of different tools. Table A.16 lists all tools used by at least two participants. Note that only 87 of 128 participants answered this question. Also, several participants used the text box to indicate more than one visualization tool, so the values do not sum up to 100%.

Visual Studio, Doxygen, and Eclipse are the most popular SV tools.

Answer	Count	Percentage
Internal documentation	64	73.56%
Personal use (source code understanding)	57	65.52%
Documentation for public release	29	33.33%
Project presentation	21	24.14%
Code reviews	20	22.99%
Project management	20	22.99%
Complexity Control	17	19.54%
Quality assessment	15	17.24%
Other	5	5.75%
<i>n</i>	87	100.00%

**Table A.15:** The reasons for which users create SVs. The most prominent uses are inter-developer communication and personal understanding.

Most SVs work on given source code without major problems.

Many participants entered more than one tool, thus biasing our results.

**Does the above tool work seamlessly and smoothly with your source code and projects?** 63% of the participants chose 'Mainly yes', 37% 'Mainly no', 41 participants were not asked, since they selected 'Never' in the first question of this part of the survey.

For the most popular tools we looked at their individual scores. Please note that these results are biased by the fact that several participants indicated more than one tool. We decided to count answers for each indicated tool. The results (normalized to 100% per group) are given in table A.17.

**Does the above tool create a visualization to your satisfaction?** 79% participants chose 'Mainly yes' and 21% 'Mainly no'. The individual results are shown in table A.18. Again, these results are biased, since some answers could not be unequivocally assigned to a single tool.



Answer	Count	Percentage
Visual Studio	11	12.64%
Doxygen	10	11.49%
Eclipse	8	9.20%
Dia	7	8.05%
StarUML	5	5.75%
Xcode	5	5.75%
Visio	4	4.60%
bouml	3	3.45%
Custom software solution	3	3.45%
Unnamed IDE	3	3.45%
Magicdraw	3	3.45%
OmniGraffle	3	3.45%
ArgoUML	2	2.30%
Enterprise Architect	2	2.30%
JavaDoc	2	2.30%
Paint	2	2.30%
Rational Software Architect	2	2.30%
Umbrello	2	2.30%
Other	30	34.48%
<i>n</i>	87	100.00%

**Table A.16:** List of primarily used SV tools. The SV user base is very fragmented.

Answer	<i>n</i>	Mainly Yes	Mainly No
Visual Studio	11	91%	9%
Doxygen	10	80%	20%
Eclipse	8	75%	25%
Dia	7	43%	57%
StarUML	5	20%	80%
Xcode	5	60%	40%
Overall	87	63%	37%

**Table A.17:** The compatibility of SV tools with the code bases of our participants.

Answer	<i>n</i>	Mainly Yes	Mainly No
Visual Studio	11	91%	9%
Doxygen	10	70%	30%
Eclipse	8	88%	12%
Dia	7	86%	14%
StarUML	5	40%	60%
Xcode	5	40%	60%
Automatic	39	85%	15%
Semi-automatic	25	77%	23%
Manual	36	74%	26%
Overall	87	79%	21%

**Table A.18:** User satisfaction with results produced by SV tools.

The results from automatic tools are as satisfying as manually created SVs.

The categorization in automatic, semi-automatic, and manual tools was provided by the participants' answers to the following question. Interestingly, there is no significant difference between automatic, semi-automatic, and manual SV tools,  $H(2) = 1.28, p = .53$ , and also the slight decrease in our sample group is not a significant trend,  $J = 1143, p = .266, z = -1.11$  (two-tailed Jonckheere-Terpstra test).

Participants rated the automation level of their most often used tool.

**Does the above tool create the visualization automatically?** The participants were given three possible answers with the following explanation:

- *Automatic* tools should be able to create the complete visualization without user input, although they might offer options that allow the user to change some global features of the visualization or allow for manual edits after the creation of an automatically generated diagram.
- *Semi-automatic* tools should have a basic understanding of the code and offer features to generate visualizations of at least basic software artifacts, although the user must, e.g., select manually what pieces to include in a diagram or how to layout the single parts.

- *Manual* tools are basically drawing applications. Although they might have some rudimentary support by providing predefined shapes and layouts, they are not capable of parsing a project or source code in order to create any part of a diagram automatically.

This categorization was apparently not sharp enough, since only the users of *Dia*<sup>2</sup> could unanimously classify it as manual tool. The results are given in table A.19. Bear in mind that we needed to map answers to more than one tool in some cases.

Users had very different perceptions of the automation of many tools.

Answer	<i>n</i>	Automatic	Semi-automatic	Manual
Visual Studio	11	36%	64%	0%
Doxygen	10	70%	20%	10%
Eclipse	8	38%	62%	0%
Dia	7	0%	0%	100%
StarUML	5	39%	25%	36%
Xcode	5	39%	25%	36%
Overall	87	39%	25%	36%

**Table A.19:** The categorization of SV tools by level of automation. The categorization is not always clear.

**How much time do you normally spend using the tool mentioned above on creating a visualization of your project?** The participants were given five time intervals to choose from. Table A.20 shows the results for all tools grouped by automation and the frequently mentioned individual tools.

Interestingly, StarUML ( $Mdn = '> 60 \text{ min}'$ ), a dedicated tool to design UML diagrams, performs significantly slower than the more general purpose tool Dia ( $Mdn = '5-15 \text{ min}'$ ),  $U = 4, p = .04, r = -.68$ , or sketches (see next section),  $U = 46, p < .001, r = -.30$ . These findings are probably biased, as StarUML was rarely mentioned exclusively, but

Dedicated UML drawing tools are not always faster than general purpose tools.

<sup>2</sup><http://live.gnome.org/Dia>

Answer	<i>n</i>	< 5 min	5-15 min	15-30 min	30-60 min	> 60 min	<i>Mdn</i>
Visual Studio	11	45%	27%	9%	9%	9%	5-15 min
Doxygen	10	20%	30%	20%	20%	10%	5-30 min
Eclipse	8	25%	53%	13%	0%	0%	5-15 min
Dia	7	0%	14%	72%	14%	0%	5-15 min
StarUML	5	0%	0%	20%	20%	60%	> 60 min
Xcode	5	40%	40%	20%	0%	0%	5-15 min
Automatic	39	47%	21%	12%	14%	6%	5-15 min
Semi-automatic	25	18%	36%	18%	14%	14%	5-15 min
Manual	36	3%	23%	42%	16%	16%	15-30 min
Overall	87	24%	26%	24%	15%	11%	15-30 min

**Table A.20:** SV tool time requirement per visualization. We found a significant trend that more automation decreases the time required to produce SVs.

in combination with several other tools (Doxygen, Netbeans UML Plugin<sup>3</sup> & MagicDraw<sup>4</sup>, and PowerPoint & Rational).

In more general terms, comparing the time required by tool automation support, there are significant differences with  $H(2) = 12.46, p < .05$ . We also found a highly significant trend with  $J = 1687.50, z = 3.58, r = .38, p < .001$ , meaning that for tools with less automation, the median time required to produce diagrams increases (medium effect). We also tested the differences between the most automatic tool (Doxygen, *Mdn* = '5–30 min') and the most manual (Dia, *Mdn* = '5–15 min') for which we have at least five samples. Their comparison shows that we cannot transfer the general trend to any two tools from different ends of the spectrum of automation:  $U = 28.5, p = .54, ns$ .

More automation leads to faster diagram creation. But there are exceptions.

### Non-users

This question was given to those participants, who indicated to never use an SV tool.

**Why do you not use a visualization software?** We offered eight reasons and the ability to type in a ninth. Participants could select multiple options. Table A.21 gives the results to this question. Please note that percentages do not sum up to 100%, since more than one answer could be given. The question was only answered by 41 participants.

Many users avoid SV tools, because they do not need them, or they are too time consuming.

Two additional reasons given in the 'Other' text box were variations of 'I did not know that there are programs for creating such visualizations' and 'Undesired/unsatisfying results' and were merged with these cases.

---

<sup>3</sup><http://netbeans.org/community/releases/55/uml-download.html>

<sup>4</sup><http://www.magicdraw.com/>

Answer	Count	Percentage
I have no need to create such visualizations	21	51.22%
Too time consuming	21	51.22%
Undesired/unsatisfying results	16	39.02%
There is no tool that can create the visualizations I need	12	29.27%
I did not know that there are programs for creating such visualizations	6	14.63%
I did not know that there are ways to visualize source code	3	7.32%
Usability issues	3	7.32%
The visualization tool I need is not affordable	0	0.00%
Other	0	0.00%
<i>n</i>	41	100.00%

**Table A.21:** Reasons for not creating SVs. Participants could give multiple answers.

#### A.2.4 Manual visualization

The following data are the basis for the results reported in section 3.2.4.

**How often do you use sketches (paper, white board, flip chart, Tablet PC etc.) to create visualizations of your software project or certain source code artifacts like classes?** Table A.22 shows the responses to this question. Half the participants create a sketch at least once a month.

#### Sketching participants

The following questions were only given to the 112 participants who sketch at least occasionally.

Answer	Count	Percentage
Daily	14	10.94%
Not daily, but several times per week	24	18.75%
At least once a week	19	14.84%
At least once a month	27	21.09%
At least once a quarter	12	9.38%
Less than once a quarter, but occasionally	16	12.50%
Never	16	12.50%
<i>N</i>	128	100.00%

**Table A.22:** Frequency of sketching. More than half of our participants sketch at least once a month or more often.

**What aspects of your software project or code artifact do you usually cover in your sketches?** The participants were given nine aspects to choose from and the option to enter a tenth in a text box. The results are shown in table A.23. Note that the percentages do not sum up to 100%, since the 112 participants, who were given this question, could select an arbitrary number of answers.

Sketches often depict the relations between classes.

Answer	Count	Percentage
Class hierarchy	74	66.07%
Class dependency	61	54.46%
Data flow	46	41.07%
Function/method calls	46	41.07%
Sequence and timing	44	39.29%
Software layers	24	21.43%
Membership	13	11.61%
Framework/package/namespace hierarchy	10	8.93%
Framework/package/namespace dependency	8	7.14%
Other	13	11.61%
<i>n</i>	112	100.00%

**Table A.23:** Most used aspects in sketches. Participants could give multiple answers.

Participants also sketch the workings of algorithms and data storage.

Actually, 14 participants provided information in the ‘Other’ text box, but one was identified as data flow aspect and merged with that category. Five participants visualize algorithms and four data storage aspects. One participant uses sketches for testing locking requirements, for hardware setup and peripheral devices, for interactions between program parts using the actor model each. One participant sketches aspects “je nachdem”<sup>5</sup>.

The tools used most often for sketching are pen and paper, followed by whiteboards.

**What materials do you use to create sketches?** We use a very wide definition of sketches. Thus, we provided a wide range of options to choose from for this question. Participants could select any number of tools and materials. Please note that the results in table A.24 are only answered by 112 participants, and that multiple answers could be chosen. Thus, the percentages do not sum up to 100%.

Answer	Count	Percentage
Pen, pencil	111	99.11%
Paper	110	98.21%
Whiteboard	50	44.64%
Multiple colors	40	35.71%
Eraser, sponge	36	32.14%
Ruler	12	10.71%
Sticky notes	11	9.82%
Flip chart	7	6.25%
Set square	7	6.25%
Magnets	6	5.36%
Computer graphics tablet	5	4.46%
Compass	0	0.00%
Other	5	4.46%
<i>n</i>	112	100.00%

**Table A.24:** Materials used for sketches. Participants could give multiple answers.

<sup>5</sup>English: “it depends”



Among ‘other’ materials were two entries mentioning different software tools (*OmniGraffle*<sup>6</sup>, *Paint*<sup>7</sup>, and *Gimp*<sup>8</sup>). Two users employ plain text editors. One participant uses “verschiebbare Schnipsel”<sup>9</sup>, which we merged with the ‘sticky notes’ category. One user employs mind maps and the software *CUEcards*<sup>10</sup>.

Developers repurpose many different applications to create sketches.

**For what purpose do you create sketches of your project or software artifacts?** We gave the participants the same options as for the related SV question. The results are shown in table A.25, again note that only 112 participants responded and could give more than one answer each.

A majority creates sketches for personal use and understanding.

Answer	Count	Percentage
Personal use (source code understanding)	95	84.82%
Internal documentation	49	43.75%
Project management	28	25.00%
Complexity control	20	17.86%
Project presentation	20	17.86%
Code reviews	19	16.96%
Quality assessment	8	7.14%
Documentation for public release	6	5.36%
Other	11	9.82%
<i>n</i>	112	100.00%

**Table A.25:** Tools and materials used for creating sketches. Participants could give multiple answers.

**How much time you usually spend on creating a sketch?**

The same time intervals as for the corresponding SV question were offered. The percentages in table A.26 are based on the 112 participants who answered this question. More than half of the participants create a sketch in 15 min or less. When we compare these results with the times for the creation of SVs, we find no significant difference:  $U = 4263, z = -1.56, n.s.$

Sketches often take less than 15 minutes to create.

<sup>6</sup><http://www.omnigroup.com/products/omnigraffle/>

<sup>7</sup><http://windows.microsoft.com/en-US/windows7/products/features/paint>

<sup>8</sup><http://www.gimp.org/>

<sup>9</sup>English: “movable snippets”

<sup>10</sup><http://www.mhst.net/cuecards/>

Answer	Count	Percentage
< 5 min	26	23.21%
5-15 min	44	39.29%
15-30 min	25	22.32%
30-60 min	13	11.61%
> 60 min	4	3.57%
<i>n</i>	112	100.00%

**Table A.26:** Time requirement per sketch. Half of our participants draw up a sketch in 15min or less.

### Non-sketching participants

The following question was displayed for the 16 participants, who indicated in the first question of this part that they never sketch.

Sketches are avoided, when deemed unnecessary or time consuming.

**Why do you not use sketches to visualize your code?** We suggested eight reasons and a text box for users to provide a ninth. Table A.27 show the results. Percentages are based on 16 out of 128 participants who could select an arbitrary number of reasons.

The 'Other' reason given by one user was the fact that he needed to provide SVs for documentation purposes and therefore it was not economical for him to also create sketches.

### A.2.5 Software visualization in documentation

In this section we give the results for the last two questions, as reported in section 3.2.5.

**Do you initially consult the documentation of a previously unknown project to find visualizations in order to understand it?** 76 participants chose 'Mainly yes'

Answer	Count	Percentage
I have no need to create such sketches.	9	56.25%
Too time consuming	7	43.75%
I find sketching too hard to do.	5	31.25%
Undesired/unsatisfying results	3	18.75%
I did not think of doing sketches to visualize code.	2	12.50%
I can not sketch the visualizations I need (e.g., they are too complicated).	2	12.50%
I did not know that there are ways to visualize source code.	1	6.25%
I can not afford the materials.	0	0.00%
Other	1	6.25%
<i>n</i>	16	100.00%

**Table A.27:** Reasons for not creating sketches. Participants could give multiple answers.

(59.38%), while 52 chose 'Mainly no' (40.62%), the two answers offered to that question.

**Do you use visualizations of a documentation regularly when you are more familiar with a project?** 37 participants chose 'Mainly no' (28.91%), 91 'Mainly yes' (71.09%).



## Appendix B

# Paper prototype user test

*“If you were really great and powerful,  
you’d keep your promises!”*

*—Dorothy challenges the Wizard of Oz  
(The Wizard of Oz, 1939)*

The following guide was used by the instructor to perform the user test of the paper prototype (see section 4.3).

- Show mock-up for use-case 1
  - Ask: “How many diagrams are contained in this project?”
  - Ask: “What is probably visualized in each of the diagrams?”
  - Ask: “What would you do to open the diagram offering the most abstract view on the project?”
- Show mock-up for use case 2
  - Ask: “How would you create a new diagram containing the method `fireResizePerformed?`”
  - Ask: “What would you do to manipulate the newly created type box in the diagram?”
- Show mock-up for use case 3(a)

- Ask: “How would you expand the diagram with the parent class of `JResizeHandle`?”
- Show original situation on mock-up of use case 3(a)
- Ask: “How would you expand the diagram with the methods calling `fireResizePerformed`?”
- Ask: “How would you delete `mouseDragged`?”
- Ask: “Which methods are called by `fireResizePerformed`?”
- Ask: “How would you expand the diagram with the method `resized` of `ResizeListener`?”
- Ask: “What is visualized by the preview?”
- Ask: “How would you expand the diagram with the child class of `ResizeListener`?”
- Ask: “Why was the method `resized` added to `JStatusBar`?”
- Ask: “Do you consider this behavior reasonable?”
- Ask: “How intuitive is the interaction? What changes do you propose?”
- Show mock-up for use case 3(b)–3(d)
- Show mock-up of complete diagram and filter sidebar
  - Ask: “What functionality do you expect from this sidebar?”
  - Ask: “What would you do to find all listeners?”
- Show mock-up of diagram with filtered elements
  - Ask: “How would you restrict the search to only show calls that use ‘Listner’ as a parameter name?”
- Show mock-up of diagram with filtered relations
  - Ask: “How would you save a filter?”
- Turn sidebar
  - Ask: “How would you disable the filter?”
  - Ask: “How would you re-enable the filter?”

- Ask: “How would you change the filter scope to look for types?”
- Show mock-up of diagram with filtered types
  - Ask: “How would you change the search term?”
  - Ask: “How would you disable all filters?”
- Ask: “How intuitive is the interaction? What changes do you propose?”
- Show mock-up of palette
  - Ask: “How would you create a group of listener types?”
  - Show mock-up of diagram with group
  - Ask: “How would you set the color of the group to green?”
  - Ask: “How would you assign an icon of an ear to the group?”
  - Ask: “How would you assign a name to the group?”
- Show mock-up with updated IDE
  - Ask: “Do you consider the changes to the package explorer and code editor to be reasonable?”





## Appendix C

# Additional user study materials

*“In the end we retain from our studies only that  
which we practically apply.”*

—Johann Wolfgang von Goethe

In this appendix we present the interested reader with details on the user study described in 6.3. We reproduce original forms used for the study in section C.1 and give details on the results in section C.2.

We presents details on the user study.

### C.1 User study forms

We used several forms for the evaluation of the Eclipse plug-in Code Gestalt (see section 6.3). In the following you find the original forms handed to the testers, namely the consent form (section C.1.1), task descriptions (section C.1.2), and final questionnaire (section C.1.3). We also reproduce the checklist used to determine error and clutter counts for the created sketches and diagrams (section C.1.4).

### **C.1.1 Consent form**

Each tester signed a consent form.

The consent form on the following page was given to each participant at the beginning of a test session. After explaining to the tester and answering any questions regarding the study, the form was signed by both the participant and the principal investigator.

# Informed Consent Form

Code Gestalt Evaluation

Christopher Kurtz  
Media Computing Group  
RWTH Aachen University  
christopher.kurtz@rwth-aachen.de

**Purpose of the study:** The goal of this study is to evaluate the *Eclipse* plug-in *Code Gestalt*. Participants will be asked to use *Code Gestalt* as well as pen and paper to create diagrams that explain how several aspects of a sample program work. Both the time needed to complete these drawings and their accuracy will then be used in an analysis to evaluate the software.

**Procedure:** You will be asked to understand software functions in a given Java painting application and then draw diagrams either by hand or using *Code Gestalt* to illustrate these functions. A video of the UI will be captured in background along with an Audio recording. This study should take about an hour to complete, depending on your proficiency with *Java* and *Eclipse*. After the study, we will ask you to fill out a questionnaire to evaluate the tested system.

In about two weeks we will contact you again via e-mail. The e-mail will contain several diagrams drawn by other users and the painting application source code. You will be asked to compare pairs of pen and paper sketches with Code Gestalt diagrams using another questionnaire. It is possible, that during this second phase other participants of the survey will get access to anonymized versions of the diagrams you created.

**Risks/Discomfort:** You may become fatigued during the course of your participation in the study. You will be given several opportunities to rest, and additional breaks are also possible. There are no other risks associated with participation in the study. Should completion of either the task or the questionnaire become distressing to you, it will be terminated immediately.

**Benefits:** The results of this study will be useful in understanding which features are required for a software visualization application to be competitive with traditional pen and paper sketches.

**Alternatives to Participation:** Participation in this study is voluntary. You are free to withdraw or discontinue the participation.

**Cost and Compensation:** Participation in this study will involve no cost to you. There will be snacks and drinks for you during and after the participation. Also you may enter a lottery for six copies of the computer game *World of Goo*.

**Confidentiality:** All information collected during the study period will be kept strictly confidential. You will be identified through identification numbers. No publications or reports from this project will include identifying information on any participant. If you agree to join this study, please sign your name below.

I have read and understood the information on this form.

I have had the information on this form explained to me.

---

Participant's Name

---

Participant's Signature

---

Date

---

Principal Investigator

---

Date

If you have any questions regarding this study, please contact Christopher Kurtz at christopher.kurtz@rwth-aachen.de

### **C.1.2 Test tasks**

The testers were given four tasks in counterbalanced order.

The following four pages contain the test tasks to be completed by the participants over the course of the study. Note that the order and assignment of tools was counterbalanced, so different users were getting tasks in different order and were to complete them with different tools.

We internally numbered the test tasks in the order of which they are presented on the following pages:

1. Switching Between Tools
2. Drawing Tool Preview
3. Changing Drawing Tool Attributes
4. Erase the Drawing Area

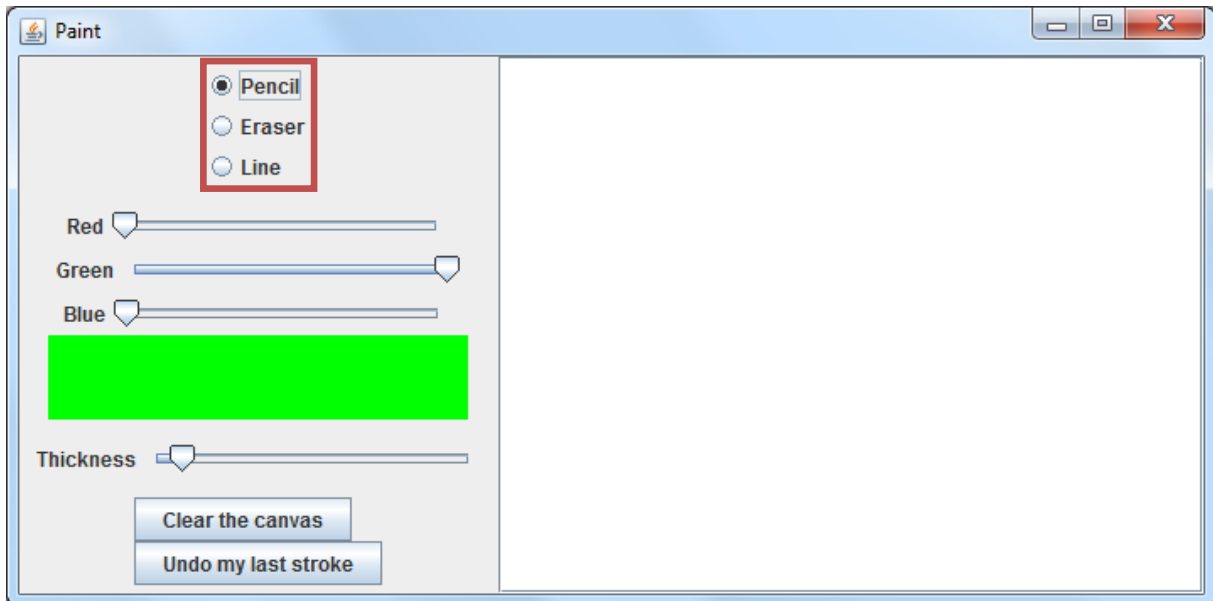
These are the numbers referred to in section 6.3.3.

## Switching Between Tools

In *Paint* the user can choose between three drawing tools: *Pencil*, *Eraser* and *Line*. In order to enable another programmer to create additional drawing tools, you need to communicate to her how the software realizes the support for them.

Study the source code of *Paint* and draw a diagram explaining how the different drawing tools (pen, eraser, line) are implemented. The diagram should explain how switching between tools is realized and contain the relevant classes and methods.

Hint: *Paint* uses *Java*'s ability for dynamic binding and reflection.

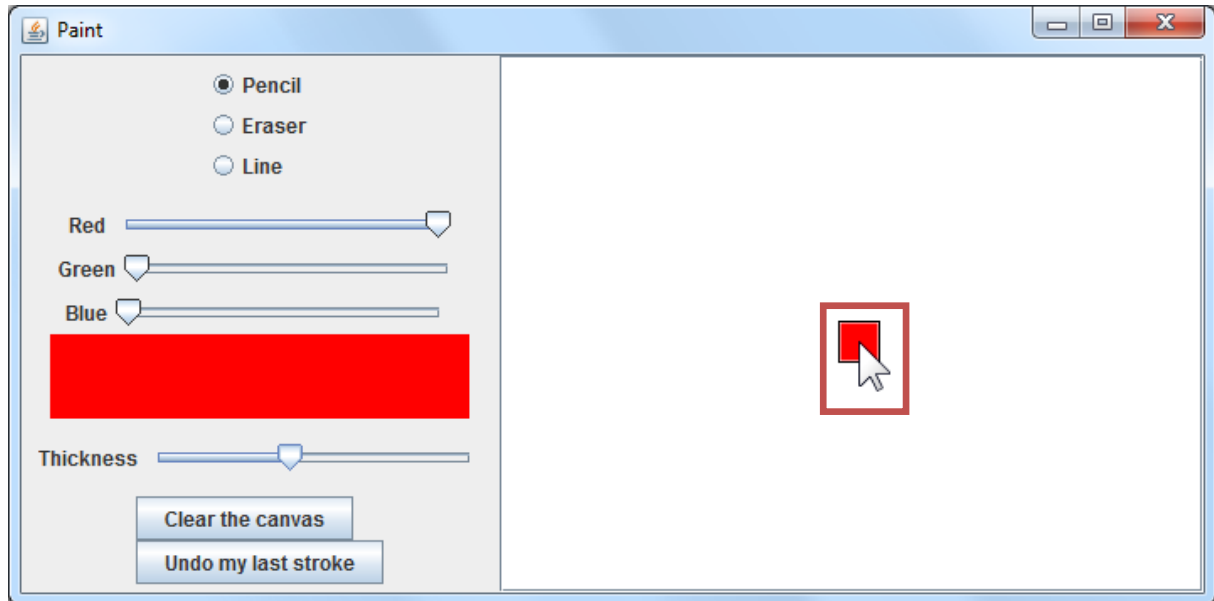


## Drawing Tool Preview

In *Paint* the user is given a preview of the shape and color of the selected tool when moving the mouse over the drawing area. You want to visualize how this function is realized.

Study the source code of *Paint* and draw a diagram explaining how the preview works. The diagram should incorporate all method calls needed beginning by the mouse cursor hovering over the drawing area and ending with the code that renders the shape.

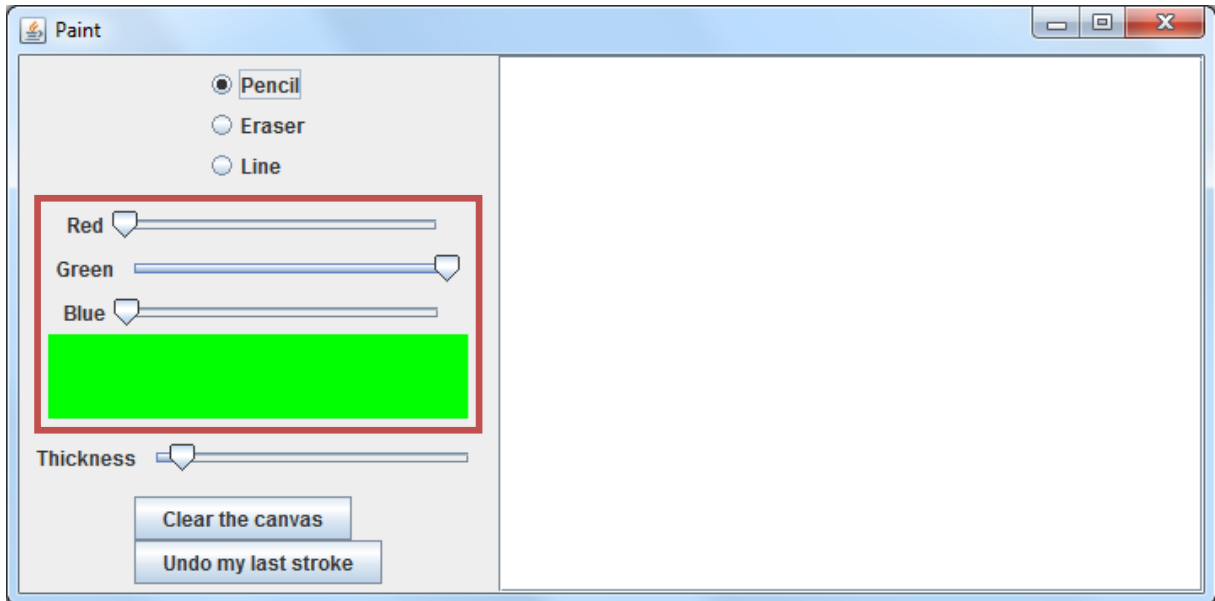
Hint: You do not need to understand or illustrate how the switching between drawing tools works.



## Changing Drawing Tool Attributes

There are three sliders in *Paint* with which the user can manipulate the RGB values of the drawing color. You are to explain to a fellow programmer how the change of the sliders in the UI results in the drawing tool to produce pixels of a different color, so he may implement other controls with which the user can manipulate attributes of the drawing tools.

Study the source code of *Paint* and draw a diagram explaining how changing the sliders impacts the drawing tools. The diagram should contain all relevant code artifacts involved with communicating the color set by the UI up to the ultimate manipulation of the image by the drawing tool.

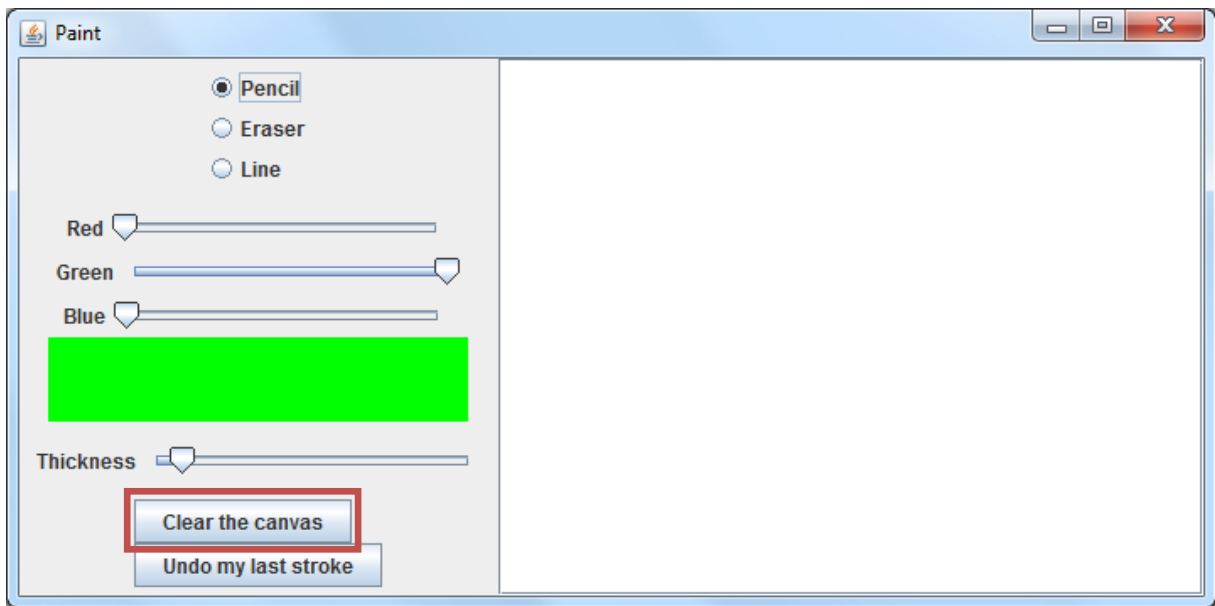


## Erase the Drawing Area

*Paint* has a button labeled *Clear the canvas* that erases the contents of the drawing area. You want to understand, how clicking the button results in the drawing area to change its contents, so you can implement other global image manipulation features like filters.

Study the source code of *Paint* and draw a diagram that illustrates how clicking the button changes the drawing area. The diagram should contain the involved code artifacts.

Hint: *Java Swing* allows for programmers to directly assign actions to buttons without a mediating listener object.





### **C.1.3 User study questionnaire**

At the end of each test session, participants were given the questionnaire reproduced on the following three pages. The first ten statements are an adaptation of the SUS questionnaire, replacing 'the system' with 'Code Gestalt'.

The testers filled out a questionnaire at the end of the session.

## Code Gestalt Questionnaire

Participant ID:

Please choose one response per statement by marking the corresponding square with an "X".

	Strongly Disagree				Strongly Agree
1. I think that I would like to use Code Gestalt frequently.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2. I found Code Gestalt unnecessarily complex.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3. I thought Code Gestalt was easy to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4. I think that I would need the support of a technical person to be able to use Code Gestalt.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5. I found the various functions in Code Gestalt were well integrated.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6. I thought there was too much inconsistency in Code Gestalt.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7. I would imagine that most people would learn to use Code Gestalt very quickly.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8. I found Code Gestalt very cumbersome to use.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9. I felt very confident using Code Gestalt.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10. I needed to learn a lot of things before I could get going with Code Gestalt.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
11. I found Code Gestalt to be very flexible.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
12. I could not predict the outcome of my interactions with Code Gestalt.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
13. I believe using Code Gestalt is a practical alternative to creating diagrams by hand.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
14. I expect the diagrams created with Code Gestalt to be confusing for most programmers.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Continued overleaf

Please rate the following features of Code Gestalt for their usefulness:

	Not useful				Very useful
15. Similarity to UML class diagrams	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
16. Generation of diagrams from a Project Explorer selection	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
17. Adding types and methods using drag-and-drop	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
18. Integration of Eclipse symbols and markers for Java entities	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
19. Adding relations using previews for selected entities	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
20. Number and selection of "expand" and "collapse members" commands for types via context menu	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
21. Inclusion of a tag cloud in type boxes	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
22. Visualization of tags in an overlay	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
23. Highlighting of tags by selecting types in the tag overlay	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
24. Highlighting of types by selecting tags in the tag overlay	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
25. Pinning tags from the overlay to the diagram	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
26. Visualizing the influence of a tag using a "fan" in the background of the diagram	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
27. Adding notes	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
28. Opening a Java Editor by double-clicking a diagram entity	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
29. Use of selection-dependent buttons (e.g. "Close" and "Change Color")	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
30. Limitation to one box per Java type	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please use the spaces to answer the following two open-ended questions.

1. Which aspects of Code Gestalt did you like? Did it offer advantages over pen-and-paper sketches?

2. Which aspects of Code Gestalt did you dislike? Which advantages did pen-and-paper sketches offer?

**Thank you for participating!**

### C.1.4 Error and clutter evaluation scheme

We used the following list of required and optional code artifacts to determine the error count for all user created diagrams in section 6.3.3 and the error and clutter counts for section 6.3.4. The items in parenthesis are optional and were neither considered an error when missing nor clutter when present.

We used a check list to count errors and clutter.

#### Task #1

- Types

- Actions
- PaintWindow
- PaintObject
- PencilPaint
- LinePaint
- EraserPaint
- PaintObjectConstructor
- (JRadioButton)
- (AbstractAction)
- (Class)

- Attributes

- PaintWindow
  - \* actions
  - \* pencilButton
  - \* eraserButton
  - \* lineButton
  - \* objectConstructor
  - \* (toolButtonGroup)
- Actions
  - \* paintWindow
  - \* eraserAction
  - \* lineAction
  - \* pencilAction

- PaintObjectConstructor
  - \* paintObjectClass
  - \* (temporaryObject)

- **Methods**

- Actions **or** AbstractAction
  - \* Actions() **or** actionPerformed()
- PaintWindow
  - \* setPaintObjectClass()
  - \* (PaintWindow())
- PaintObjectConstructor
  - \* setClass()

- **Calls**

- actionPerformed() **or** Actions() → setPaintObjectClass()
- setPaintObjectClass() → setClass()
- (newInstance())
- (PaintWindow() → Actions())

- **Inheritance**

- PencilPaint → PaintObject
- LinePaint → PencilPaint
- EraserPaint → PencilPaint

## **Task #2**

- **Types**

- PaintWindow
- PaintObjectConstructorListener
- PaintObjectConstructor
- PaintCanvas
- PaintObject
- (MouseListener)
- (MouseMotionListener)
- (JPanel)

- Attributes

- PaintWindow
  - \* objectConstructor
  - \* canvas
- PaintObjectConstructor
  - \* constructionListener
- PaintCanvas
  - \* hoveringObject

- Methods

- PaintWindow
  - \* hoveringOverConstructionArea()
  - \* (PaintWindow())
- PaintObjectConstructorListener
  - \* hoveringOverConstructionArea()
- PaintObjectConstructor
  - \* mouseMoved()
  - \* makeHoveringPrototype()
  - \* (PaintObjectConstructor())
  - \* (mouseDragged())
  - \* (mousePressed())
  - \* (mouseReleased())
  - \* (mouseExited())
- PaintCanvas
  - \* setHoveringObject()
  - \* paintComponent()
  - \* (addMouseListener())
  - \* (addMouseMotionListener())
- PaintObject
  - \* paint()
  - \* (define())
  - \* (setColor())
  - \* (setThickness())

- Calls

- mouseMoved() → hoveringOverConstructionArea()

- (mouse\* () → hoveringOverConstructionArea ())
- mouseMoved () →  
makeHoveringPrototype ()
- (mouse\* () → makeHoveringPrototype ())
- hoveringOverConstructionArea () →  
setHoveringObject ()
- paintComponent () → paint ()
- (makeHoveringPrototype () → define ())
- (makeHoveringPrototype () →  
setColor ())
- (makeHoveringPrototype () →  
setThickness ())
- (PaintWindow () →  
PaintObjectConstructor ())

- Inheritance

- PaintWindow → PaintObjectConstructor

### Task #3

- Types

- PaintWindow
- PaintObjectConstructor
- PaintObject
- (ChangeListener)
- (JSlider)

- Attributes

- PaintWindow
  - \* colorChangeListener
  - \* rSlider
  - \* gSlider
  - \* bSlider
  - \* objectConstructor
- PaintObjectConstructor
  - \* color



- PaintObject
  - \* color

- Methods

- ChangeListener
  - \* (stateChanged())
- PaintObjectConstructor
  - \* setColor()
  - \* mousePressed()
- PaintObject
  - \* setColor

- Calls

- stateChanged() → setColor()
- mousePressed() → setColor()

- Inheritance

- None

#### Task #4

- Types

- PaintWindow
- Actions
- PaintCanvas
- (AbstractAction)
- (JButton)

- Attributes

- PaintWindow
  - \* actions
  - \* clearButton
  - \* canvas
- Actions
  - \* clearAction
  - \* (paintWindow)

- Methods

- `PaintWindow`
  - \* `clear()`
  - \* `(PaintWindow())`
- `PaintCanvas`
  - \* `clear()`
- `Anonymous, AbstractAction, or Actions`
  - \* `actionPerformed()` or `Actions()`

- Calls

- `actionPerformed()` or `Action()` → `PaintWindow.clear()`
- `PaintWindow.clear()` → `PaintCanvas.clear()`

- Inheritance

- None

## C.2 Results and analysis

The following data and their analysis are the basis of the results reported in section 6.3.3. All data is based on  $N = 16$ , unless noted otherwise.

### C.2.1 Population

Most participants were male student assistants.

Two participants were female (12.5%), 14 were male (87.5%). Four testers were doctoral students/research assistants (25%), eight student assistants (50%), three diploma thesis students (18.75%), and one tester was a Bachelor student (6.25%). Participants had a mean of 10.03 years of general programming experience ( $Mdn = 8.25, s = 6.34$ ), 3.69 years Java programming experience ( $Mdn = 2.5, s = 3.90$ ), and 3.01 years of experience with the Eclipse IDE ( $Mdn = 2.5, s = 2.30$ ). Seven testers had none, six only ephemeral experience with SV tools. The remaining four

participants had a mean of 2.94 years of SV experience ( $Mdn = 2.5, s = 2.63$ ). Our participants spent a mean of 14.47 hours a week on programming tasks ( $Mdn = 10, s = 14.27$ ), and dealt with a mean of 37.8% of unknown code ( $Mdn = 27.5\%, s = 30.0\%$ ).

### C.2.2 Completion rates and times

We report the completion rates (10 minutes per task) in table C.1. The differences in completion rates are not significant. The results of Mann-Whitney tests for each of the four tasks are listed in table C.2. We found no significant differences.

Task	Sketching	Code Gestalt
#1	50.0%	62.5%
#2	25.0%	25.0%
#3	62.5%	25.0%
#4	87.5%	87.5%
<i>M</i>	56.3%	50.0%

**Table C.1:** Completion rates for each task/system combination.

Task	<i>U</i>	<i>z</i>	<i>p</i>	Significance
#1	28.00	-.49	1.00	<i>ns</i>
#2	32.00	.00	1.00	<i>ns</i>
#3	20.00	-1.46	0.32	<i>ns</i>
#4	32.00	.00	1.00	<i>ns</i>

**Table C.2:** Comparison of completion rates between sketching and Code Gestalt.

We report the completion times for completed tasks in table C.3. Deciding on a statistical test for comparing completion times is not straightforward, as data from some test conditions are normally distributed, whereas others are not (probably due to the small sample size). For reasons of uniformity, we chose to compare completion times

We found no significant differences in the completion rates of sketching and Code Gestalt.

We found one significant differences in the completion times of sketching and Code Gestalt.

using the non-parametrized Mann-Whitney test. The results are shown in table C.4. Sketches were significantly faster completed than Code Gestalt diagrams for task #4,  $U = 9.00, z = -1.98, p < .05$ , the effect size  $r = -.53$  is large.

Task	<i>n</i>	Sketching			<i>n</i>	Code Gestalt		
		<i>M</i>	<i>Mdn</i>	<i>s</i>		<i>M</i>	<i>Mdn</i>	<i>s</i>
#1	4	540	525	42.4	5	526	570	127.0
#2	2	517.5	517.5	116.7	2	570	570	42.4
#3	5	424	420	123.4	2	600	600	0
#4	7	364.3	300	123.6	7	511.4	511.4	49.7

**Table C.3:** Completion times (in seconds) for each task/system combination.

Task	<i>U</i>	<i>z</i>	<i>p</i>	Significance
#1	7.50	-.62	.603	<i>ns</i>
#2	1.5	-.41	1.000	<i>ns</i>
#3	1.00	-1.61	.238	<i>ns</i>
#4	9.00	-1.98	.048	$p < .05$

**Table C.4:** Comparison of completion times between sketching and Code Gestalt.

### C.2.3 Errors

We counted errors for each diagram in five categories.

For each diagram created by our testers we counted the number of missing essential artifacts (errors) according to appendix C.1.4. The descriptive statistics in table C.5 are based on the eight diagrams created by the participants for each test task/system combination.

Again, both Kolmogorov-Smirnov and Shapiro-Wilk tests indicate that most of the data sets are not normally distributed. Thus, we again use Mann-Whitney tests to compare the error counts as reported in table C.6.

Test Task	System	Types		Attributes		Methods		Call Relations		Inheritance Relations					
		<i>M</i>	<i>s</i>	<i>M</i>	<i>s</i>	<i>M</i>	<i>s</i>	<i>M</i>	<i>s</i>	<i>M</i>	<i>s</i>				
#1	Sketch	3.75	4.0	1.39	7.0	1.93	1.88	1.5	0.99	1.25	1.0	0.71	1.88	2.5	1.36
	CG	1.38	1.0	1.69	5.5	2.12	1.13	1.0	1.13	1.13	1.0	0.84	0.75	0.0	1.39
#2	Sketch	3.38	3.5	1.60	3.13	3.0	0.84	5.0	1.85	2.50	3.0	1.41	0.88	1.0	0.35
	CG	1.75	2.0	0.71	1.75	2.0	0.71	3.0	0.76	2.13	2.0	0.35	0.88	1.0	0.35
#3	Sketch	1.00	1.0	1.07	1.00	1.0	1.07	1.50	1.0	0.75	1.0	0.71	0.00	0.0	0.00
	CG	0.88	1.0	0.64	0.88	1.0	0.64	1.75	2.0	1.88	2.0	0.35	0.00	0.0	0.00
#4	Sketch	0.63	0.0	1.06	1.88	2.0	0.84	1.00	1.0	0.63	0.5	0.74	0.00	0.0	0.00
	CG	0.75	1.0	0.46	2.38	2.0	0.92	1.25	1.0	1.25	1.0	0.71	0.00	0.0	0.00

Table C.5: Error count for each of the five categories per task/system combination.

Test Task	Types			Attributes			Methods			Call Relations			Inheritance Relations		
	<i>U</i>	<i>z</i>	<i>p</i>	<i>U</i>	<i>z</i>	<i>p</i>	<i>U</i>	<i>z</i>	<i>p</i>	<i>U</i>	<i>z</i>	<i>p</i>	<i>U</i>	<i>z</i>	<i>p</i>
#1	8.0	-2.56	.007	17.0	-1.59	.124	19.5	-1.37	.241	29.5	-0.28	.984	18.0	-1.62	0.132
#2	9.5	-2.41	.013	21.0	-1.27	.302	12.0	-2.15	.036	22.0	-1.13	.269	32.0	0.00	1.000
#3	32.0	0.00	1.000	26.5	-0.59	.561	27.0	-0.55	.620	6.5	-2.92	.006	32.0	0.00	1.000
#4	23.0	-1.06	.315	22.5	-1.06	.407	25.0	-0.85	.495	17.5	-1.63	.161	32.0	0.00	1.000

**Table C.6:** Error count comparison between systems for each of the five categories per test task.

The tests detected the following significant differences between sketching and Code Gestalt:

- Users of Code Gestalt missed significantly fewer types in tasks #1 and #2,  $r_{\#1} = -.64, r_{\#2} = -.60$ . With  $p < .01$ , the difference for task #1 is highly significant with a large effect.
- Users of Code Gestalt missed significantly fewer methods in task #2,  $r_{\#2} = -.54$ . The effect is large.
- Sketchers missed significantly fewer call relations in task #3,  $r_{\#3} = -.73$ . With  $p < .01$ , this result is highly significant with a large effect.

We found four significant results in the 20 observed cases.





## Appendix D

# Second online survey materials

*“You just don’t get it, do you, Jean-Luc?  
The trial never ends. We wanted to see if you had  
the ability to expand your mind and your horizons.  
And for one brief moment, you did.”*

—Q (*Star Trek: The Next Generation*)

We invited all 16 testers from the user study (see section 6.3) to take a second survey between December 16<sup>th</sup>, 2010 and January 2<sup>nd</sup>, 2011, in which we let them compare four pairs of sketches and Code Gestalt diagrams for each of the test tasks. We present the questions of the survey in appendix D.1, as well as additional data gathered and its analysis in appendix D.2.

We present details on the online survey following the user test.

### D.1 Survey questions

We used the following questions in the second survey for the evaluation of Code Gestalt as described in section 6.3.4.

### D.1.1 Diagram/sketch comparison

Testers compared for pairs of sketches and Code Gestalt diagrams.

Each participant was given a set of four pairs of Code Gestalt diagrams and sketches (one per user study task). The original task text was reproduced (see appendix C.1.2) for each pair of diagrams. The pairs were to be rated in four categories using seven-point Likert scales.

**Please rate the following two diagrams.** Four seven-point Likert scales from 'strongly favor pen and paper' to 'strongly favor Code Gestalt' (mandatory) for each of the following:

- Which diagram is clearer?
- Which diagram is more understandable?
- Which diagram would you rather use as aid to solve the task?
- Which diagram is better suited to document the source code?

**After comparing the four pairs, which type of diagram is particularly well suited for which scenario?** Free text

Help: You may also discuss this from a point of view not expressed in the four previous questions.

### D.1.2 Additional features

Testers rated a list of suggested features for importance.

We compiled a list of all features suggested by participants during the user test. This list was used for the final question.

**During the user study many participants gave suggestions for improvement of Code Gestalt. To guide the further development of Code Gestalt, please rate the importance of the following features:** 28 five-point Likert scales from 'entirely insignificant' to 'very important' (optional)

- 
- Automatic search for and preview relations to elements that are not yet included in the diagram
  - Labeling of relations
  - New relation type 'override/implements' between methods in a type hierarchy
  - Make tag cloud at bottom of type box optional (default: disabled)
  - New relation type 'dependence' between types
  - 'Open Call Hierarchy' and 'Open Type Hierarchy' commands in diagram context menu
  - 'Remove members from inverse selection' command in context menu
  - Add members to types using a text box with incremental search
  - Manual relation drawing tools
  - Drag-and-drop of members to any location and automatically add them to the correct type box
  - Framework specific relations (e.g. from a button to an action)
  - Allow highlighting of tags/types in tag overlay for multiple elements
  - Drag-and-drop source code to diagram
  - New relation 'Access' from methods to fields
  - Support 'Link with Editor' feature of Eclipse Package Explorer
  - JavaDoc tool tips
  - Preview relations during drag-and-drop from Package Explorer
  - Full support for anonymous and local types
  - Assign random colors to thematic relation fans on creation
  - Routing-algorithm for relations (to minimize overlap)

- ‘Send to Diagram’ command in context menu of Package Explorer and Java Editor
- ‘Find’ command for diagram
- Show thematic relation fan in tag overlay
- Show context sensitive controls (i.e., ‘Expand All’, ‘Change Color’) on mouse-hover (instead of selection)
- Automatic layout (optional)
- ‘Find’ command for tag overlay
- Show tag overlay only while holding down a key (quasi-mode)
- Rich Text for sticky notes

## **D.2 Results and analysis**

The following data is the basis of the results reported in section 6.3.4.

### **D.2.1 Comparison of sketches with Code Gestalt diagrams**

Code Gestalt is slightly clearer and more suitable for documentation.

We report the descriptive statistics of the survey results in table D.1. Again, we normalized the Likert scale to the range of  $[-1..1]$ .  $-1$  represents ‘strongly prefer pen and paper sketch’,  $1$  ‘strongly prefer Code Gestalt diagram’, and  $0$  ‘no preference’.

Test Task	Group	$n$	Clearness			Understandability			Task Support			Documentation		
			$M$	$Mdn$	$s$	$M$	$Mdn$	$s$	$M$	$Mdn$	$s$	$M$	$Mdn$	$s$
#1	A	8	0.21	0.17	0.56	0.17	0.00	0.47	0.17	0.33	0.47	0.38	0.50	0.52
	B	6	0.50	0.50	0.35	-0.33	-0.33	0.37	-0.17	-0.33	0.55	0.44	0.33	0.50
#2	A	8	0.50	0.67	0.44	0.54	0.67	0.31	0.58	0.67	0.35	0.67	0.67	0.25
	B	6	0.67	0.67	0.21	0.33	0.33	0.30	0.44	0.33	0.17	0.50	0.50	0.18
#3	A	8	-0.29	-0.17	0.38	-0.50	-0.33	0.36	-0.38	-0.33	0.28	-0.38	-0.33	0.40
	B	6	-0.06	0.17	0.65	-0.55	-0.5	0.27	-0.50	-0.67	0.41	-0.22	-0.33	0.54
#4	A	8	0.33	0.33	0.40	0.25	0.33	0.46	0.17	0.17	0.31	0.25	0.33	0.35
	B	6	0.17	0.17	0.62	-0.11	0.00	0.34	-0.11	-0.17	0.40	0.17	0.17	0.46
All	Both	56	0.24	0.33	0.53	-0.01	0.00	0.52	0.04	0.00	0.50	0.23	0.33	0.51

**Table D.1:** Comparing pairs of diagrams on a scale from -1 ('strongly favor sketch') to 1 ('strongly favor Code Gestalt diagram').



# Bibliography

Klaus Alfert and Alexander Fronk. 3-dimensional visualization of java class relations. In *Proceedings of the 2000 IDPT Conference - The Fifth World Conference on Integrated Design & Process Technology*, 2000.

Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29:33–43, April 1996.

Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4:114–123, May 2009.

Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *In proceedings of the 9th International Workshop on Program Comprehension*, pages 7–17. IEEE Computer Society Press, 2001.

John Brooke. Sus - a quick and dirty usability scale. In *In Usability Evaluation in Industry*, 1996.

Stuart M. Charters, Nigel Thomas, and Malcolm Munro. The end of the line for software visualisation. In *In Proceedings of the 2nd Workshop on Visualizing Software for Analysis and Understanding*, pages 110–112. Society Press, 2003.

Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Pearson Education, Inc., 2008.

Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *In Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

- Cleidson de Souza, Jon Froehlich, and Paul Dourish. Seeking the source: software source code as a social and technical artifact. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work, GROUP '05*, pages 197–206. ACM, 2005.
- Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18: 957–968, November 1992.
- David Erni. Codemap – improving the mental model of software developers through cartographic visualization. Master’s thesis, University of Bern, 2010.
- Andy Field. *Discovering Statistics using SPSS*. SAGE Publications Ltd, 3 edition, 2009. ISBN 978-1-84787-907-3.
- Alexander Fronk, Armin Bruckhoff, and Michael Kern. 3d visualisation of code structures in java software systems. In *Proceedings of the 2006 ACM symposium on Software visualization, SoftVis '06*, pages 145–146. ACM, 2006.
- George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30:964–971, November 1987.
- Orla Greevy, Michele Lanza, and Christoph Wyseier. Visualizing live software systems in 3d. In *Proceedings of the 2006 ACM symposium on Software visualization, SoftVis '06*, pages 47–56. ACM, 2006.
- Martin Halvey and Mark T. Keane. An assessment of tag presentation techniques. In *Proceedings of the 16th 16th International World Wide Web Conference, WWW 2007*, pages 1313–1314, May 2007.
- J. F. Kelley. An empirical methodology for writing user-friendly natural language computer applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems, CHI '83*, pages 193–196, December 1983.
- Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32:971–987, December 2006.



- Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–128, 2008.
- Christian F.J. Lange and Michael R.V. Chaudron. Interactive views to improve the comprehension of uml models - an experimental validation. In *15th IEEE International Conference on Program Comprehension, 2007. ICPC '07.*, pages 221–230, June 2007.
- Michele Lanza. Combining metrics and graphs for object-oriented reverse engineering. Master's thesis, University of Bern, Switzerland, 1999.
- Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29:782–795, September 2003.
- Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.
- Leonhard Lichtschlag and Jan Borchers. Codegraffiti: Communication by sketching for pair programming. In *UIST 2010 Extended Abstracts*, New York, NY, October 2010.
- Rob Lintern, Jeff Michaud, Margaret-Anne D. Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–56, 209, June 2003.
- Jacopo Malnati. X-ray — an eclipse plug-in for software visualization. Bachelor thesis, University of Lugano, Switzerland, 2007.
- Joe Marks and Stuart Shieber. The computational complexity of cartographic label placement. Technical Report TR-05-91, Harvard University Center for Research in Computing Technology, Cambridge, Massachusetts, December 1991.
- Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, pages 181–204, December 1993.

William Moore, David Dean, Anna Gerberm, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Corp., 2004.

Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, March 1990. ISSN 1045-926X.

Inc. Object Management Group. Unified modeling language version 2.3. <http://www.omg.org/spec/UML/2.3/>, May 2010.

Michael J. Pacione, Marc Roper, and Murray Wood. A comparative evaluation of dynamic visualisation tools. In *10th Working Conference Proceedings on Reverse Engineering, 2003. WCRE 2003.*, pages 80–89, November 2003.

Yunrim Park and C. Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2009. VIS-SOFT 2009.*, pages 3–10, September 2009.

Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *CHI '92 Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 83–91, 1992.

Mariam Sensalire and Patrick Ogao. Visualizing object oriented software: Towards a point of reference for developing tools for industry. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007.*, pages 26–29, June 2007.

Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Classifying desirable features of software visualization tools for corrective maintenance. In *Proceedings of the*

- 4th ACM symposium on Software visualization, SoftVis '08*, pages 87–90. ACM, 2008.
- Bonita Sharif and Jonathan I. Maletic. The effect of layout on the comprehension of uml class diagrams: A controlled experiment, 2009.
- Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of CASCON*, pages 209–223, 1997.
- Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Visual Languages and Human-Centric Computing (VL/ HCC)*, pages 4–8. IEEE Computer Society, 2006.
- Daniel Speicher and Sebastian Jancke. Smell detection in context. In *12. Workshop Software-Reengineering (WSR 2010)*, May 2010.
- Daniel Speicher and Jan Nonnen. Consistent consideration of naming consistency. In *12. Workshop Software-Reengineering (WSR 2010)*, May 2010.
- Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.
- John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 3–10, September 1992.
- Margaret-Anne D. Storey. *A Cognitive Framework For Describing And Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, December 1998.
- Margaret-Anne D. Storey and H.A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of International Conference on Software Maintenance, 1995*, pages 275–284, October 1995.
- Richard Wettel. *Software Systems as Cities*. Doctoral dissertation, University of Lugano, Switzerland, 2010.

Cornelia M. Yoder and Marilyn L Schrag. Nassi-shneiderman charts: An alternative to flowcharts for design. *Software Engineering Notes*, 3, November 1978.

Peter Young and Malcolm Munro. A new view of call graphs for visualising code structures. *Computer Science Technical Report*, March 1997.

# Index

3DRD, *see* 3d relation diagram  
3d relation diagram, 22, 44

algorithm visualization, 9, 30  
Apple, 40  
Augur, 34

Bézier spline, 90, 111

call graph, 16, 17  
call relation, 21, 90, 104  
CallStax, 17, 44  
circular bundle view, 18  
city metaphor, 24  
class diagram, 2, 86  
- layout, 34  
- SVs based on, 20  
cloud view, 15, 90  
Code Crawler, 32  
Code Gestalt, 1, 129  
- diagram, 98  
- Eclipse implementation, 95  
- future work, 131  
- paper prototype, 55  
- Silverlight prototype, 79  
CodeCity, 25, 44  
CodeCrawler, 14, 19  
CodeMap, 26  
contextual buttons, 111  
Creole, 13, 32, 33  
CUEcards, 165  
Cultivate, 15, 90, 107  
CVS, 13

data trace, 57  
degrees of freedom, xxx  
Dia, 159  
DIA cycle, 37  
diagram widget, 64

- disharmony maps, 26
- document life cycle, 96
- Doxygen, 47, 161
- drag-and-drop, 75, 87, 102
- dynamic feature interaction view, 19
  
- Eclipse, 47, 96
  - call hierarchy, 126
  - editor, 96
  - Java editor, 103
  - outline view, 102
  - properties view, 102
  - type hierarchy, 126
  - User Interface Guidelines, 98
  - view, 96
  - wizard, 98
- EditPart, 97
- effect, xxxi
- Excel, 41
- execution trace, 18
- Expression Blend, 87
- Expression Design, 98
- Extravis, 18
  
- feature creep, 128
- fisheye view, 11
- framework flow, 59
  
- GEF, *see* Graphical Editing Framework
- Graphical Editing Framework, 97
- grouce, 151
  
- hash map, 80
- heat map, 56, 83, 109
  
- IBM SPSS Statistics, 41
- icon, 56
- IDE, *see* integrated development environment
- inheritance relation, 21, 90, 104
- Institut für Informatik III, 93
- integrated development environment, 32, 98
  
- Java, 80, 96, 115
- Java Development Toolkit, 97
- JDK, *see* Java Development Toolkit
- JTransformer, 15
  
- Kendall's *tau*, xxx
- Kruskal-Wallis test, xxx
  
- lack of insight, 53

- landmark, 56, 91
- LimeSurvey, 40
- lines of code, 10
- live preview, 70, 88, 105, 127
- LoC, *see* lines of code
- local context view, 62
- Logo, 28
  
- MagicDraw, 161
- Mann-Whitney test, xxx
- map metaphor, 24
- massive sequence view, 18
- McCabe complexity, 18
- mean, xxx
- Media Computing Group, 39
- median, xxx
- MetaView, 24
- MetricView, 23
- Microsoft, 40
- model-view-controller pattern, 97
- modification request number, 10
- Moose, 19
- Mozilla Firefox, 64
- MR number, *see* modification request number
  
- Netbeans UML Plugin, 161
- NoA, *see* number of attributes
- NoM, *see* number of methods
- number of attributes, 14
- number of methods, 14
  
- OmniGraffle, 165
  
- Paint, 165
- Paint.NET, 98
- paper prototype
  - design, 66
  - evaluation, 74
  - implementation, 67
- Pascal, 28
- Pearson correlation coefficient, xxx
- polymetric view, 14
  - three-dimensional, 19
- PowerPoint, 161
- probability, xxx
- program execution, 16
- Prolog, 28
  
- quartile, xxx
  
- radial menu, 90

- 
- Relo, 20, 66, 87, 97, 132
  - Rigi, 12
  - RWTH Aachen University, 39, 93, 112
  
  - SA4J, 34
  - Seesoft, 10
  - SHriMP, *see* Simple Hierarchical Multi-Perspective views
  - Silverlight, 79
  - Silverlight prototype
    - design, 80
    - evaluation, 93
    - implementation, 87
  - Simple Hierarchical Multi-Perspective views, 11
  - sketch, 3, 49
  - SketchFlow, 79, 87
  - Smalltalk, 15
  - software visualization, 3, 9
    - dynamic, 35
    - evaluation, 32
    - perceived usefulness of, 41
    - taxonomy, 28
    - tool, 46
  - Source Navigator, 32
  - sourceforge.net, 33
  - Spearman's rank correlation coefficient, xxx
  - special purpose tool sets, 13
  - standard deviation, xxx
  - StarUML, 159
  - state chart, 151
  - structured context diagram, 60
  - Superscape Viscap, 17
  - survey, 30, 39
  - SUS, *see* System Usability Scale
  - SV, *see* software visualization
  - system complexity view, 14, 19
  - System Usability Scale, 113
  
  - tabbed browsing, 64
  - tag, 82
    - overlapping of, 83, 108
    - pinning a, 86, 109
  - tag cloud, 84, 103, 107, 126
  - tag overlay, 6, 130
    - design, 81
    - highlighting, 83, 109, 133
    - implementation, 107
    - prototype, 90
    - sweepline algorithm, 108
  - Tcl/Tk, 12
  - term, 82



- 
- tf-idf, 133
  - thematic heat map, 56
  - thematic relation, 6, 131
    - design, 84
    - implementation, 109
    - prototype, 90
  - thematic software map, 26, 44, 81
  - time consumption problem, 52
  - top-level visualization, 10
  - TraceCrawler, 19
  - TraceScraper, 19
  - type box, 5
    - concept, 64
    - implementation, 103
    - paper prototype, 70
    - Silverlight prototype, 86
  
  - UML-city, 24
  - undo, 97, 102
  - University of Bonn, 93, 112
  - unknown source code, 1
  - use case diagram, 151
  
  - version control system, 3, 10, 67, 129
  - Version Tree, 33
  - virtual reality, 17
  - VisMOOS, 22
  - Visual Studio, 47
  - vocabulary problem, 80
  
  - World of Goo, 93
  
  - X-Ray, 15
  - Xia, 13
  
  - z-score, xxx

