
On the Evolution of Source Code and Software Defects

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Marco D'Ambros

under the supervision of
Prof. Dr. Michele Lanza

October 2010

Dissertation Committee

Prof. Dr. Carlo Ghezzi Politecnico di Milano, Italy
Prof. Dr. Cesare Pautasso Università della Svizzera Italiana, Switzerland

Prof. Dr. Harald C. Gall University of Zurich, Switzerland
Prof. Dr. Hausi A. Müller University of Victoria, Canada

Dissertation accepted on 19 October 2010

Prof. Dr. Michele Lanza
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Michele Lanza
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Marco D'Ambros
Lugano, 19 October 2010

To Anna

Abstract

Software systems are subject to continuous changes to adapt to new and changing requirements. This phenomenon, known as software evolution, leads in the long term to software aging: The size and the complexity of systems increase, while their quality decreases. In this context, it is no wonder that software maintenance claims the most part of a software system's cost. The analysis of software evolution helps practitioners deal with the negative effects of software aging.

With the advent of the Internet and the consequent widespread adoption of distributed development tools, such as software configuration management and issue tracking systems, a vast amount of valuable information concerning software evolution has become available. In the last two decades, researchers have focused on mining and analyzing this data, residing in various software repositories, to understand software evolution and support maintenance activities. However, most approaches target a specific maintenance task, and consider only one of the several facets of software evolution. Such approaches, and the infrastructures that implement them, cannot be extended to address different maintenance problems.

In this dissertation, we propose an *integrated* view of software evolution that combines different evolutionary aspects. Our thesis is that an integrated and flexible approach supports an *extensible* set of software maintenance activities. To this aim, we present a meta-model that integrates two aspects of software evolution: source code and software defects. We implemented our approach in a framework that, by retrieving information from source code and defect repositories, serves as a basis to create analysis techniques and tools. To show the flexibility of our approach, we extended our meta-model and framework with e-mail information extracted from development mailing lists.

To validate our thesis, we devised and evaluated, on top of our approach, a number of novel analysis techniques that achieve two goals:

1. Inferring the causes of problems in a software system. We propose three retrospective analysis techniques, based on interactive visualizations, to analyze the evolution of source code, software defects, and their co-evolution. These techniques support various maintenance tasks, such as system restructuring, re-documentation, and identification of critical software components.
2. Predicting the future of a software system. We present four analysis techniques aimed at anticipating the locations of future defects, and investigating the impact of certain source code properties on the presence of defects. They support two maintenance tasks: defect prediction and software quality analysis.

By creating our framework and the mentioned techniques on top of it, we provide evidence that an integrated view of software evolution, combining source code and software defects information, supports an extensible set of software maintenance tasks.

Acknowledgements

Life is relationship and relationships shape everything we do. The work presented in this thesis would not have been possible without the support and influence of a number of people who I had the fortune and the privilege to meet.

First of all, I would like to thank my advisor Michele Lanza for his guidance and support during my Ph.D. Many thanks also for your patience in dealing with my strange and improvised ideas (e.g., “lets write a paper”—a couple of days before the deadline). Michele, what I learned from you during these years goes far beyond the work presented in this thesis and research in general. I wish you, Marisa, Alessandro, and the new member of the family all the best.

Special thanks go to the members of my dissertation committee—Harald Gall, Carlo Ghezzi, Hausi Müller, and Cesare Pautasso—for dedicating a part of their precious time to evaluate my work, and for providing very valuable and detailed feedback. There is something special that I really appreciated from each of you: Harald, I learned a lot from you on how to be very focused and professional, while also having fun; Carlo, you are able to be among the world-wide top scientists in the software engineering scene and, at the same time, such a cool guy; Hausi, I really admire the energy and passion that you put in whatever you do, from reviewing my thesis to playing soccer and climbing the great wall; Cesare, thanks for your positive and constructive feedback which boosted my motivation in the last moments before the defense.

To my colleagues, the former and current members of the REVEAL research group: It has been a pleasure and a privilege to work with you all. I always enjoyed the easy atmosphere in our office, the jokes, and the sense of playfulness alternated with hard working sessions. Romain “Omar” Robbes, you were an example for me, and I learned a lot from you. Thanks for being always kind and offering your help all the time. Mircea “the long” Lungu, I am glad that I met such a sympathetic and cheery person as you are. I will always remember the awesome trips we did around the world, and I am certain that our friendship will last no matter where we will be or what we will do. With Ricky “the vettel” Wettel I always had fun time, exchanging jokes, reading funny blogs, and watching nonsensical clips. I already miss the time we have spent together. My best wishes to you and Simi, I am sure that you will be fantastic parents. Lile “the thermostat” Hattori, thanks for accepting my continuous jokes and for the Eclipse-related help... without forgetting the precious (?) Google stickers. Fernando “beautiful coder” Olivero, your peacefulness and passion for OO programming is contagious. It was fun playing soccer together, and I am looking forward to visiting you in the southernmost city of the world and having a full-fledged asado together (and perhaps a cup of tea, lad). Alberto “12 O’clock” Bacchelli, even if our personalities are opposite (*i.e.*, the chaotic vs the precise), we got along well from the very first moment. You are a very generous person whom one can always rely on. Working with you was always instructive and pleasant, both for teaching and for research: I especially enjoyed our missions impossible such as FASE, QSIC, ICPC. I really hope that we will continue to collaborate, and I wish you and Sara a marvelous future together. Tommaso “graspa” Dal Sasso, you just

joined the REVEAL group, but your sharp sympathy is already spreading.

I am also grateful to all the people that I had the pleasure to meet during my staying in Lugano: Mehdi, Shima, Alessio, Paolo, Alessandra, Domenico, Mostafa, Nicolas, Marco, Jeff, Giovanni, Francesco, Cédric, Dmitrijs, Edgar, Milan, Morgan, Sasa, Cyrus, Adina, Antonio, and many others that are too numerous to list. To our amazing secretaries, Cristina, Danijela, Elisa and Nina: Thanks for always trying to fulfill my improbable requests while keep smiling. Many thanks go to a number of people that I regularly met at conferences and workshops: Martin, Ahmed, Ettore, Carl, Jonathan, Andi, Denys, Rocco, Filippo, Max, Doru, and many others. We had fruitful discussions, which influenced my work, and a great deal of fun too.

I owe very special thanks to many people that do not have anything to do with my research, but contributed to this work by making my life much better. I am grateful to Zamo and Lukino, my best friends: You were, are, and will always be a firm reference point for me. To my friends Polly, Carmelo, Sara, Marcoalto, Elegance, Vivandiere, Vittorio, Alice, Emi, Simo, Ema, Samba, Pablo, Franco, Dany, Michela, Panji, Davide, Giovanni, Betty, Giulio, Giuseppe, Barla, Lucky, Salvo, Pigei, Antonio, Cucia, Raffo, Vito, Nunzia, Rossella, Paolo, Orla, Griso, Ampio, Stefano, Tala, Nico, Luca, Elsa, and many others that are too numerous to mention: In different ways and time you all had played an important role in my life. Thanks for all the crazy adventures and cool stuff that we did together, which were always cause of fun, joy and enthusiasm. To the young families that share with us the experience of having a baby: Vale, Dany and Samu (the eagle king); Fra, Peppe, Gaia and Alice; Sheila, Vale and Marco; Vitto, Ricky, Giovanni and Lucia; Anna, Poldis and Francy. I would like to thank you for your support, the precious advises, and especially for the good time spent together. To all the kids, boys, and girls that I had the honor to accompany during a small stretch of their path: Thanks for teaching me so well the importance of playing, and not taking ourselves too seriously.

Very special thanks go to the members of my amazing family, which I am really proud to be part of. To my mom and dad, thanks for your unconditional love and support, and for everything I have learned from you, which made me the way I am. My aunt Anna (“super”) was always part of the family, and I considered you as a wise older sister. Paola, my oldest sister, you have been an example for me, since when I was a kid. When you and your family—Ben, Michael, Johannes, and Anne—visited us, you were always bringing good cheer and a great deal of fantasy. Luca, you are a special brother and a mould-breaker person. Despite the thousands of kilometers that separate us, I could always rely on you in the most important moments of my life. To my sister Chiara, thanks for everything you did for me, and for being always available and helpful. With my youngest sister Marta I have had a special relationship. Your spontaneity and cheerfulness make the people around you happy. I wish you and Paoleous all the very best. My gratitude goes also to my extended family. Andrea, Michela, Samu, Franco, and Mariangela: Your warmth and sympathy made me feel part of the family from the very first moment.

Last but not least, my love and gratefulness go to my son Matteo: Thanks for your big and curious eyes, for tending your little hands to me, and especially for reminding me every day that happiness resides in the most simple things. To the new baby who is flourishing with his/her mom: I can hardly wait to meet and hold you in my arms!

And most of all, to the person whose smile changed my life: Anna, this work is dedicated to you, as it would have been impossible to accomplish it without your love and support. I cannot imagine my life without you: My love and gratitude extend beyond words and reasons.

Marco D'Ambros
October 2010

Contents

Contents	ix
List of Figures	xv
List of Tables	xix
I Prologue	1
1 Introduction	3
1.1 Our Thesis	5
1.2 Contributions	6
1.2.1 Modeling	6
1.2.2 Analysis	6
1.2.3 Tools	7
1.2.4 A Side Track: Collaborative Software Evolution Analysis	7
1.3 Roadmap	8
2 State of the Art	11
2.1 Genesis of our Approach	11
2.1.1 The Seventies	11
2.1.2 The Eighties	12
2.1.3 From the Nineties to Mining Software Repositories	12
2.1.4 The Advent of Mining Software Repositories	13
2.1.5 Problems of Software Development Tools for MSR	14
2.1.6 Summing Up and Contextualizing our Approach	16
2.2 Modeling Software Evolution	18
2.2.1 Summing Up	22
2.3 Software Evolution Visualization	22
2.3.1 Software Visualization	22
2.3.2 Evolutionary Visualization	25
2.3.3 Summing Up	27
2.4 Change Coupling Analysis	28
2.4.1 Change Coupling Visualization	28
2.4.2 Summing Up	29
2.5 Defect Prediction and Analysis	30

2.5.1	Defect Analysis	30
2.5.2	Defect Prediction	32
2.6	Design Flaw Detection and Analysis	35
2.6.1	Summing Up	36
2.7	Summary	36
3	Modeling and Supporting Software Evolution	39
3.1	Modeling an Evolving Software System	39
3.1.1	Modeling Source Code	40
3.1.2	Modeling Software Defects	42
3.1.3	Mevo: A Meta-Model of Evolving Software Systems	46
3.2	Our Approach in Action	47
3.2.1	Retrieving the Data and Populating the Models	48
3.2.2	Pre-processing and Linking the Data	48
3.2.3	Limitations	52
3.3	Applications	54
3.4	Tool Support	56
3.4.1	Churrasco's Architecture	57
3.4.2	Discussion	60
3.5	Summary	62
II	Inferring Causes of Problems	63
4	Analyzing Integrated Change Coupling Information	67
4.1	The Evolution Radar	68
4.1.1	Example	70
4.1.2	Change Coupling Measure	70
4.1.3	Interaction	71
4.1.4	Discussion	74
4.2	Using the Evolution Radar for Retrospective Analysis	74
4.2.1	Azureus	74
4.2.2	ArgoUML	80
4.2.3	Discussion	83
4.3	Supporting Maintenance with the Radar	83
4.3.1	Integration in the IDE	84
4.3.2	Experimental Evaluation	86
4.3.3	Lessons Learned	88
4.4	Summary	88
5	Analyzing the Evolution of Software Defects	89
5.1	Visualizing a Bug Database	90
5.1.1	The System Radiography View	90
5.1.2	The Bug Watch View	93
5.2	Experiments	97
5.2.1	Analysis in the Large	97
5.2.2	Analysis in the Small	100
5.3	Discussion	102

5.4	Summary	102
6	Code and Bug Co-Evolution Analysis	103
6.1	Visualizing Evolving Software	104
6.2	Co-Evolutionary Patterns	108
6.2.1	Persistent	109
6.2.2	Day-Fly	110
6.2.3	Introduced for Fixing	111
6.2.4	Stabilization	112
6.2.5	Addition of Features	113
6.2.6	Bug Fixing	114
6.2.7	Refactoring / Code Cleaning	115
6.2.8	Summing Up	116
6.3	Experiments	117
6.3.1	Case Studies	117
6.3.2	Characterizing the Evolution of the Systems	117
6.4	Limitations	123
6.5	Summary	124
III	Predicting the Future	125
7	Predicting Defects with the Evolution of Source Code Metrics	129
7.1	Motivations	130
7.2	Experiments	131
7.2.1	Prediction Task	131
7.2.2	Evaluating the Approaches	131
7.2.3	Tool Support	133
7.3	Bug Prediction Approaches	133
7.3.1	Change Metrics	133
7.3.2	Previous Defects	134
7.3.3	Source Code Metrics	135
7.3.4	Entropy of Changes	135
7.3.5	Churn of Source Code Metrics	137
7.3.6	Entropy of Source Code Metrics	139
7.4	Benchmark Dataset	140
7.5	Results	142
7.6	Threats to Validity	145
7.7	Summary	146
8	Improving Defect Prediction with Information Extracted from E-Mails	147
8.1	Methodology	148
8.1.1	Extending Mevo to Model E-Mail Data	149
8.1.2	Popularity Metrics	150
8.2	Experiments	151
8.2.1	Correlations Analysis	152
8.2.2	Defect Prediction	153
8.3	Discussion	155

8.4	Threats to validity	156
8.5	Related work	157
8.6	Summary	157
9	On the Relationship Between Change Coupling and Software Defects	159
9.1	Measuring Change Coupling	160
9.2	Case Studies	163
9.3	Correlation Analysis	163
9.3.1	Results	164
9.3.2	Discussion	166
9.4	Regression Analysis	167
9.4.1	Results	168
9.4.2	Discussion	168
9.5	Threats to Validity	170
9.6	Summary	170
10	On the Relationship Between Design Flaws and Software Defects	171
10.1	Design Flaws and Detection Strategies	172
10.2	Experimental Setup	173
10.3	Experiments	175
10.3.1	Correlation Analysis	176
10.3.2	Delta Analysis	179
10.3.3	Wrapping Up	181
10.4	Threats to Validity	181
10.5	Summary	182
IV	Epilogue	183
11	Conclusion	185
11.1	Contributions	186
11.1.1	Modeling and Supporting Software Evolution	186
11.1.2	Analyzing Software Evolution to Infer Causes of Problems	186
11.1.3	Analyzing the Evolution of a Software System to Predict Its Future	187
11.1.4	Tools	188
11.2	Limitations and Future Work	189
11.3	Closing Words	191
V	Appendices	193
A	Supporting Collaborative Software Evolution Analysis	195
A.1	Churrasco's Collaboration Support	196
A.1.1	The Visualization Module	196
A.1.2	The Annotation Module	198
A.2	Collaboration in Action	200
A.2.1	Analyzing ArgoUML	200
A.2.2	First Collaboration Experiment: Analyzing JMol	202

A.2.3	Second Collaboration Experiment: Collaborative Restructuring	204
A.3	Discussion	206
A.3.1	Tool Building Issues	206
A.3.2	Wrapping Up	207
A.4	Related Work	207
A.5	Summary	208
B	The Meta-Base	209
B.1	Object Persistence	209
B.2	Generating Descriptors with the Meta-base	213
B.3	Example	214
B.4	Summary	215
	Bibliography	217

Figures

1.1	The roadmap of our work	9
2.1	A word cloud obtained from the abstracts of the 2008, 2009 and 2010 editions of the Mining Software Repositories conference	14
2.2	An overview of the history of software evolution and MSR	16
2.3	Related work in the context of our thesis roadmap	17
3.1	An example of source code snapshots and SCM meta-data	40
3.2	A class diagram of the versioning system meta-model	41
3.3	Distribution of Mozilla’s bugs according to their lifetime. More than 50% of bugs lived more than six months.	43
3.4	An example of a bug report	44
3.5	A class diagram of the bug meta-model	44
3.6	The Bug Status Transition Graph	45
3.7	Mevo: A meta-model of an evolving software system. It is composed of three linked parts: The versioning system, the source code and the bug meta-models.	46
3.8	The overall view of our approach in action, divided in data retrieval and pre-processing (links inference) and applications	47
3.9	Reconstructing CVS transactions: Fixed and sliding time window	49
3.10	Possible linkings among the different parts of the Mevo meta-model in our implementation	49
3.11	An example of package nesting. With our linking approach the Java class Layout is translated in the path <code>./org/uml/ui/Layout.java</code> to be found in the versioning system repository.	50
3.12	Linking multiple versions of the Layout FAMIX class with artifact versions of the versioning system model	50
3.13	Linking versioning system artifacts with bug reports: We first detect bug ids in commit comments and then check that the fix was committed (commit timestamp) after the bug was reported (bug creation timestamp).	51
3.14	Linking fixes (commits fixing a bug) with a bug re-opened several times: If there are multiple fixes between two re-opened events, we filter them out, as the data is not consistent (a bug must be re-opened before being fixed again).	52
3.15	The main components of the Churrasco framework and the analysis tools built on top of it	56
3.16	The architecture of Churrasco	57
3.17	The Churrasco web portal	59

3.18	A screenshot of the Churrasco web portal showing a visualization of ArgoUML . . .	61
4.1	Principles of the Evolution Radar	68
4.2	An example Evolution Radar applied on the <code>core3</code> module of Azureus	70
4.3	An example of misleading results when considering the entire history of artifacts to compute the change coupling value: <code>file1</code> and <code>file2</code> are not coupled during the last year.	71
4.4	Evolution of the change coupling with the <code>Model</code> module of ArgoUML. The Evolution Radar keeps track of selected files along time (yellow border).	73
4.5	Spawning an auxiliary Evolution Radar	73
4.6	Evolution Radars of the <code>org.gudy.azureus2.ui</code> package of Azureus	75
4.7	Evolution Radars for <code>GlobalManagerImpl</code> and <code>DownloadManagerImpl</code>	76
4.8	Evolution Radar for Azureus's <code>aelitis.core</code> package and details of the coupling between this package and the class <code>DHTPlugin</code> , February – August 2006	78
4.9	Evolution Radars for the <code>core3</code> package of Azureus. They are helpful to detect the transition from the old <code>MainWindow.java</code> to the new one.	79
4.10	Evolution Radars applied to the <code>Explorer</code> module of ArgoUML	80
4.11	Details of the change coupling between ArgoUML's <code>Explorer</code> module and the classes <code>FigActionState</code> , <code>FigAssociationEnd</code> and <code>FigAssociation</code>	81
4.12	Evolution Radars of the <code>Explorer</code> and <code>Diagram</code> modules of ArgoUML from June to December 2004	82
4.13	The Evolution Radar integrated in the System Browser IDE. It shows the change coupling between the <code>CodeCityGlyphs</code> module and the rest of <code>CodeCity</code> 's modules.	84
4.14	The change coupling evolution of the class <code>AbstractGlyph</code>	87
5.1	The principles of the System Radiography visualization	91
5.2	A System Radiography of Eclipse from October 2001 to July 2009. Only bugs with new, assigned or reopened statuses are considered.	92
5.3	Counting open bugs over time in a row of the System Radiography view	93
5.4	A Bug Watch Figure visualizing Bug 5119 of Mozilla between Oct 19, 1999 and Oct 16, 2001	94
5.5	A Bug Watch visualization of the bugs affecting the JDT product of Eclipse. The reference time interval is 20/2/2003 – 5/25/2004. We represent each bug as a Bug Watch Figure.	95
5.6	Clustering bugs according to the similarity of their life cycles. The clustering eases comparing bugs and spotting “exceptional” bugs.	96
5.7	A System Radiography of Mozilla from June 1999 to April 2003. We consider bugs with <i>new</i> , <i>assigned</i> or <i>reopened</i> statuses only.	98
5.8	An extract of a System Radiography of Mozilla from June 1999 to April 2003, zoomed on the y axis. We consider open bugs with <i>critical</i> or <i>blocker</i> severity only.	99
5.9	A Bug Watch visualization of the bugs affecting the <code>Browser::Networking</code> component of Mozilla. The reference time interval is November 2002 – April 2003.	101
6.1	One way of visualizing the co-evolution of code and bugs	104
6.2	The structure of a Discrete Time Figure	105
6.3	The color mapping used in the Discrete Time Figure	105
6.4	Introducing phases in a Discrete Time Figure	106

6.5	Discrete Time Figures applied to a directory tree. The internal color of the rectangles is not visible, while the color of the boundaries is. Thus, phases and addition/removal events are still intelligible.	107
6.6	An example of a persistent pattern. The entity is also bug persistent.	109
6.7	An intensive and persistent pattern. The entity is also bug persistent.	109
6.8	A day-fly pattern	110
6.9	A dead day-fly pattern	110
6.10	An example of the introduced for fixing pattern	111
6.11	A stabilization pattern	112
6.12	An example of the addition of features pattern with the high stable (commits) – high stable (bugs) pair of phases	113
6.13	An example of the bug fixing pattern with the high stable (commits) – stable (bugs) pair of phases	114
6.14	An example of the refactoring / code cleaning pattern with the high stable (commits) – stable (bugs) pair of phases	115
6.15	A visualization of the libjava module of gcc. The area marked as “B” is a visualization of the files belonging to the java/lang directory.	119
6.16	A visualization of the CalendarClient module of Mozilla	122
6.17	An example of a wrong classification: Bar.java is wrongly defined as an introduced for fixing. The cause of the error is that the developer committed a bug fix together with other changes.	123
7.1	An example of entropy of code changes	136
7.2	Computing metric deltas from sampled versions (FAMIX models) of a system . . .	138
7.3	The types of data used by different bug prediction approaches.	141
8.1	Overall schema of our approach	149
8.2	Linking e-mails and classes	150
9.1	Sample scenario of classes and transactions	161
9.2	Example <i>EWSOC</i> and <i>LWSOC</i> computations	162
9.3	Correlations between number of defects per class and class level change coupling measures, number of changes, and the best object-oriented metric (<i>i.e.</i> , Fan out for Eclipse and ArgoUML, and CBO for Mylyn). The correlations are measured with the Spearman’s coefficient.	164
9.4	Spearman’s correlations between number of major/high priority bugs and change coupling measures	165
9.5	Results of the regression analysis for Eclipse	169
10.1	The Brain Method detection strategy	172
10.2	An example of design flaw matrix for brain methods	175
10.3	Average number of design flaws per class, grouped by flaw	176
10.4	Computing Spearman’s correlations over multiple versions of a system	177
10.5	Spearman’s correlations between number of design flaws and number of post-release defects over multiple versions of software systems	178
10.6	Percentages of systems’ versions with a strong correlation (Spearman’s > 0.4) with post-release defects	179

10.7	Extracting and measuring design flaw addition events	180
10.8	Spearman's correlations between additions of design flaws and number of generated defects	180
A.1	System Complexity and Correlation View principles	197
A.2	A Correlation view applied to the ArgoUML software system. Nodes represent classes, nodes' position represents number of lines of code (x) and number of post-release bugs (y), and nodes' color maps the number of methods.	198
A.3	A screenshot of the Churrasco web portal showing a System Complexity visualization of ArgoUML	199
A.4	An excerpt of a report generated by Churrasco. The entities having one or more annotations are highlighted in red, and the corresponding annotations are provided.	200
A.5	The web portal of Churrasco visualizing System Complexities of the Model package of ArgoUML on the left, and the entire ArgoUML system on the right	201
A.6	Evolution Radars of ArgoUML	202
A.7	A System Complexity of JMol. The color denotes the amount of annotations made by the users. The highlighted classes (green boundaries) are annotated classes.	203
B.1	Types of inheritance in GLORP	212
B.2	Generating and using GLORP descriptors with the Meta-base	213
B.3	The UML class diagram of our example	214
B.4	Examples of generated and populated database tables	215

Tables

1.1	The different aspects of the evolution of source code, software defects and their co-evolution considered in our thesis, and the corresponding maintenance activities supported	5
2.1	A summary of the approaches presented at MSR 2008, 2009 and 2010, classified according to the data they analyzed and the goals they achieved	15
2.2	Approaches modeling software evolution	19
2.3	Software evolution visualization approaches classified according to the abstraction level and the visualized data sources	26
2.4	Confusion matrix: The outcome of classification approaches	32
3.1	Requirements for a software evolution meta-model and framework	39
3.2	Languages and technologies supported in our approach implementation	47
3.3	Commit size in several software projects	54
5.1	Statistics about the bug databases of ArgoUML, Eclipse and Mozilla	90
5.2	The properties of the five areas highlighted in Figure 5.7	97
6.1	The colors used in the Discrete Time Figure and their meanings	108
6.2	The catalog of co-evolutionary patterns	116
6.3	The dimensions of the software systems used for the experiments	117
6.4	Characterizing Apache and gcc in terms of patterns detected	118
6.5	Characterizing the four biggest modules of Mozilla in terms of patterns detected	121
7.1	Categories of bug prediction approaches	133
7.2	Change metrics used by Moser <i>et al.</i>	134
7.3	Class level source code metrics	135
7.4	Systems in the benchmark	140
7.5	Explanative power for all the bug prediction approaches	142
7.6	Predictive power for all the bug prediction approaches	143
8.1	Class-level popularity metrics	150
8.2	Dataset	152
8.3	Spearman's and Pearson's correlation coefficients between number of defects per class and class level popularity metrics (and <i>LOC</i>)	153
8.4	Defect prediction results: Adjusted R^2	154

8.5	Defect prediction results: Spearman's correlation coefficient	155
9.1	Measures of the studied software systems	163
10.1	Software systems used for the experiments	173
10.2	The data set used for the experiments	174
A.1	A subset of the annotations made on Jmol. NOA stands for number of attributes and NOM for number of methods.	204
A.2	A subset of the results from the questionnaire, using a Likert Scale [Lik32] (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree)	204
A.3	A subset of the annotations made on the Smalltalk web application	205
A.4	A subset of the results from the questionnaire, using a Likert Scale (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree)	206

Part I
Prologue

Chapter 1

Introduction

In the light of the sheer size and complexity of today’s software systems, it is no wonder that software maintenance takes up the most part of a software system’s cost. In the last decades, while society has been increasingly relying on software, the cost of software maintenance, compared to the global cost of software, has grown. Researchers estimated it to be between 50% and 75% [Dav95; Som96] in 1995–96, while more recent estimates reached 85–90% [Erl00; SPL03].

The high cost of maintenance results from many factors. Among these, Corbi put in evidence and estimated how the system understanding, needed to perform maintenance tasks, takes up to 50–60% of maintenance time [Cor89]. Not updated, or not even existing documentation [KC98], and frequent turnover of developers concur to define this condition. Moreover, as a software system increases in size and complexity, maintenance becomes harder. For example, according to Purushothaman and Perry [PP05], 40% of bugs are introduced while fixing other bugs.

The first to observe that maintaining software becomes more complex over time, as software is continuously changed, were Lehman and Belady in 1985 [LB85]. In a set of laws they stated that a system must continuously change to remain useful in a mutating environment, and—if nothing is done to prevent it—the complexity of the system increases and its quality decreases, *i.e.*, the system undergoes a phenomenon known as “software aging” [Par94]. The only way to control the negative consequences of software aging is to place change and evolution in the center of the software development process [Nie04]. Without specific support for evolution, software systems become more complex, and thus harder to change and maintain.

In Lehman’s laws [LB85] the term “software evolution” was used for the first time, but it took until the nineties until the term received widespread acceptance. In the last twenty years, problems concerning software evolution continued to challenge researchers. Improvements of the software development process models [Roy70; Gil77; YCM78; Gil81; Boe88; Bec99; Coc01], and advances of development tools have been influencing this research domain. Software configuration management tools (SCM) were first introduced in 1975 [Roc75], and are now considered fundamental for software development [ELH⁺05]. The first bug tracking system was implemented in 1992, and nowadays such systems count dozens of implementations.

In the nineties, with the advent of the Internet and the improvement in network bandwidth, software development started to be distributed. In this scenario, SCMs obtained widespread attention and usage, and software repositories became available for analysis. The first approach

to analyze such repositories was proposed in 1997 [BAHS97]. Nowadays, as the usage of development tools—such as SCMs and bug tracking systems—has become an established practice, a research area is dedicated to mining software repositories (MSR) [HMHJ05].

Researchers, targeting different goals, proposed a number of approaches to mine software repositories. Some techniques study developers' behavior with the aim of enhancing development tools and environments. Other approaches analyze the data residing in various repositories with two main objectives, related to software evolution and maintenance:

1. *Infer the causes of current problems* in a software system. Such causes are the focus of software maintenance activities, to keep the system in an evolvable and maintainable state. Approaches in this area aim at detecting poorly designed components [VRD04; FG06], identifying critical bugs [DLP07a], spotting highly coupled modules [GHJ98; PGFL05; BH06; DL06b], *etc.*
2. *Predict the future* of a software system. This prediction allows practitioners to anticipate problems and to optimize available maintenance resources. Research in this area addresses questions like: Which component will change the most [GDL04]? Which software entity will generate more bugs [GKMS00; NB05a; NBZ06; NZHZ07; ZPZ07; BEP07]? *etc.*

In this dissertation, we survey the state of the art in analyzing software evolution and mining software repositories (cf. Chapter 2). Based on the survey, we extract four requirements for an approach that models software evolution and supports various maintenance activities:

1. *Integration*. There is a problematic gap between the information produced during software development and the information needed for mining software repositories. One of the issues is the lack of integration among the distinct pieces of information produced during software development, such as SCM data, source code, problem reports, *etc.* With few exceptions [FPG03; GHJ04; uMSB05], all the approaches proposed in the literature focus on a single data source. We argue that the integration of different data sources, which describe different evolutionary aspects, better supports maintenance activities than the analysis of a single data source.
2. *Flexibility*. Most software evolution and MSR approaches tackle a specific maintenance problem and cannot be adapted to address a different one. We envision a more general approach, on top of which one can build several analysis techniques, supporting different maintenance tasks.
3. *Modeling defects*. All the surveyed approaches, which consider software defects, do not model their evolution. We argue that defects are first class entities that—as source code—evolve over time.
4. *Replicability*. Replicability is a significant issue in the MSR research community [Rob10]. Robles observed that most of the experiments in MSR are difficult—when not impossible—to replicate, as the data is not available. We argue that publicly available data makes it possible not only to replicate the experiments, but also to compare and improve analysis techniques.

1.1 Our Thesis

The goal of our thesis work is to create an approach that, by modeling software evolution, supports an extensible set of software maintenance tasks.

Various software repositories, containing different types of data, revolve around the evolution of software systems. Among them, we decide to focus on source code and software defects. Source code plays a key role in software maintenance activities, and it is often the only reliable information about a software system, since the documentation is outdated or not existing [KC98]. Moreover, source code is the place where a considerable fraction of maintenance effort is spent, as for example 50% of maintenance time is used just to understand the code [FH83; Sta84; Cor89]. Software defects provide valuable information about software quality and might point to ailing system components, which represents candidates for bug fixing activities.

We formulate our thesis as:

An integrated view of software evolution, combining historical information regarding source code and software defects, supports an extensible set of software maintenance tasks, targeted at inferring the causes of problems and predicting the future of software systems.

We analyze several aspects of source code, defects and their evolution, as for example change coupling (the evolutionary dependency of software artifacts that frequently change together) or design flaws. Table 1.1 lists all of them and shows the types of data we use and the software maintenance tasks we support. Examples of such tasks are defect prediction, system re-documentation, and restructuring.

Table 1.1. The different aspects of the evolution of source code, software defects and their co-evolution considered in our thesis, and the corresponding maintenance activities supported

Evolutionary aspect analyzed	Data	Software maintenance task supported
Co-evolution of source code artifacts (change coupling)	SCM meta-data	System re-documentation and restructuring
		Identification of candidates for re-engineering
	SCM meta-data and bug repository	Defect prediction
Evolution of defects	Bug repository	Characterization and identification of critical software components and critical bugs
Co-evolution of code and defects	SCM meta-data and bug repository	Characterization and identification of critical software components
Evolution of source code metrics	Source code snapshots, SCM meta-data and bug repository	Defect prediction
Popularity metrics extracted from e-mails	Source code snapshots, SCM meta-data, mailing lists and bug repository	Defect prediction
Evolution of design flaws	Source code snapshots, SCM meta-data and bug repository	Software quality analysis

To validate our thesis, we devised an approach that fulfills the four requirements introduced in the previous section. In particular, our approach

- *is integrated*. It provides an integrated meta-model that combines multiple versions (snapshots) of the source code, the detailed evolution of each source code artifact as recorded by an SCM (SCM meta-data), and the evolution of software defects (bug repository).
- *is flexible*. We implemented the approach as a framework that serves as a basis to create software evolution analysis techniques. The framework is flexible with respect to both the meta-model and the analysis techniques that can be created on top of it (we created seven of them). As a proof on concept, we extended the meta-model to describe e-mail information and we devised a technique—on top of the framework—that uses this information to support defect prediction (cf. Chapter 8).
- *models defects as first class entities*. In our meta-model we provide an extensive description of software defects, which includes their histories.
- *enables replicability*. Our framework features a publicly accessible web interface that allows one to download the models residing in the framework. Other researchers can use such models to replicate the experiments we present.

The validation of our thesis consists in devising and applying several analysis techniques on top of our framework. With these techniques, we show that, by analyzing the evolution of source code and defects, we support the software maintenance tasks listed in Table 1.1. To complete the validation of the thesis, we validate each proposed analysis technique on its own.

1.2 Contributions

The contributions of this dissertation can be classified in three categories: modeling, analysis and tools.

1.2.1 Modeling

1. We define *Mevo* [DL08b; DL10], an integrated meta-model of software evolution that combines source code information, software defect data and SCM meta-data. We present Mevo in Chapter 3.
2. We model software defects as first class entities, including their history in the Mevo meta-model.

1.2.2 Analysis

3. We developed the Evolution Radar (cf. Chapter 4), a visualization technique that integrates module- and file-level change coupling information [DLL06; DL06b; DGLP08; DLL09]. The technique supports system restructuring, re-documentation and detection of reengineering candidates. We evaluate the Evolution Radar by analyzing two open-source systems and by performing an experiment with a practitioner.

4. We created the System Radiography and Bug Watch views [DLP07a], two visualizations to analyze the evolution of software defects at different levels of granularity (cf. Chapter 5). The views are useful to detect critical software components, to characterize bugs based on their histories, and to identify critical defects. We apply them on the bug database of Mozilla.
5. We devised a visualization technique, called *Discrete Time Figure* [DL06c; DL09], that supports the understanding of source code and defects co-evolution (cf. Chapter 6). On top of the visualization, we define a catalog of co-evolutionary patterns that are useful to characterize software entities. We use Discrete Time Figures and the catalog of patterns to characterize three large open-source systems.
6. We present two novel defect prediction approaches based on the evolution of source code metrics [DLR10] (e.g., fanIn/Out, coupling) and their comparison with existing defect prediction techniques (cf. Chapter 7).
7. We produced a benchmark for defect prediction [DLR10], in the form of a publicly available dataset consisting of hundreds of versions of several software systems (cf. Chapter 7).
8. We extended Mevo to model e-mail information (cf. Chapter 8). Based on metrics extracted from the extended meta-model, we devised a defect prediction technique and we evaluated its performance on four open-source systems [BDL10].
9. We investigate the correlation between change coupling and software defects on three open-source software systems [DLR09]. We also present a novel defect prediction approach, based on change coupling information, and its evaluation and comparison with existing prediction techniques (cf. Chapter 9).
10. We empirically analyze the relationship between a catalog of design flaws and software defects, on six open-source software systems (cf. Chapter 10). We also study the evolution of the flaws over multiple versions of the systems, to empirically assess whether adding design flaws is likely to generate defects [DBL10].

1.2.3 Tools

11. We built Churrasco [DL08a], an extensible framework that implements the Mevo meta-model and serves as a basis for all our analysis techniques.
12. We created three interactive and scalable visualization tools: Evolution Radar [DL06a], Bug's Life and BugCrawler [DL07]. These tools implement respectively the Evolution Radar visualization, the System Radiography and Bug Watch views and the Discrete Time Figure visualization.
13. We implemented Pendolino, a scriptable data analysis tool that computes and exports a variety of metrics extracted from Mevo models.

1.2.4 A Side Track: Collaborative Software Evolution Analysis

During the implementation of our software evolution framework, the need of collaboration in software development has gained increasing attention. Tools that support collaboration, such as

Jazz for Eclipse [Fro07], were only recently introduced, but hint at a larger current trend. Just as the software development teams are geographically distributed, consultants and analysts are too. Specialists in different domains of expertise should be allowed to collaborate without the need of being physically present together. For these reasons, we implemented support for collaborative software evolution analysis in our framework and we evaluated it with two experiments [DLLR09; DLLR10]. As this topic lies outside of the main target of our thesis, but is still relevant to software evolution analysis, we present it in an appendix (Appendix A).

1.3 Roadmap

Figure 1.1 shows a roadmap of our thesis work. Research topics are placed, according to their similarity, in two main spaces: Modeling software evolution and supporting software maintenance. The latter is further divided in two sub-spaces, according to the goal we want to achieve: inferring causes of problems or predicting the future. Research topics that target both the goals are placed on top of both the spaces. In Figure 1.1, research topics are connected by a path representing the reading flow of this dissertation, where we also indicate the chapters. When applicable, we show the publications and the tools that we implemented next to the topic. The research about collaborative software evolution analysis is connected to the rest of the thesis with a dashed path, since it is a side track not part of the main topic of the thesis.

We structured the remainder of this dissertation as follows:

- In **Chapter 2** (p.11) we provide a historical perspective on our work, by presenting the history of software evolution and mining software repositories. Subsequently, we explore approaches and techniques in the domain related to our thesis: modeling software evolution, software visualization, change coupling analysis, bug prediction and analysis, and design flaws detection and analysis.
- In **Chapter 3** (p.39) we describe how we tackle the problem of modeling software evolution with Mevo, an integrated meta-model of code and defect evolution. We also present an extensible framework that implements the meta-model, and serves as a basis to create analysis techniques and tools. The framework provides a web interface that can be used to retrieve Mevo models, thus enabling the replication of our experiments.

Part II: Inferring Causes of Problems. In this part of our dissertation, we present different techniques, built on top of our software evolution approach, aimed at inferring causes of problems in software systems, by analyzing their evolution.

- **Chapter 4** (p.67) presents the Evolution Radar, a visualization technique that supports system restructuring, re-documentation and identification of reengineering candidates. The Evolution Radar visualizes co-change information, a particular aspect of source code evolution that characterizes software artifacts frequently changed together. We apply the Evolution Radar on two open-source software systems and we let a developer use the tool for a re-documentation task, reporting on his experience.
- In **Chapter 5** (p.89) we analyze the evolution of software defects with a visualization technique called Bug's Life. The technique provides two views to study software defects at different granularity levels, enabling the characterization and the detection of critical bugs. We use Bug's Life to analyze the bug repository of Mozilla.

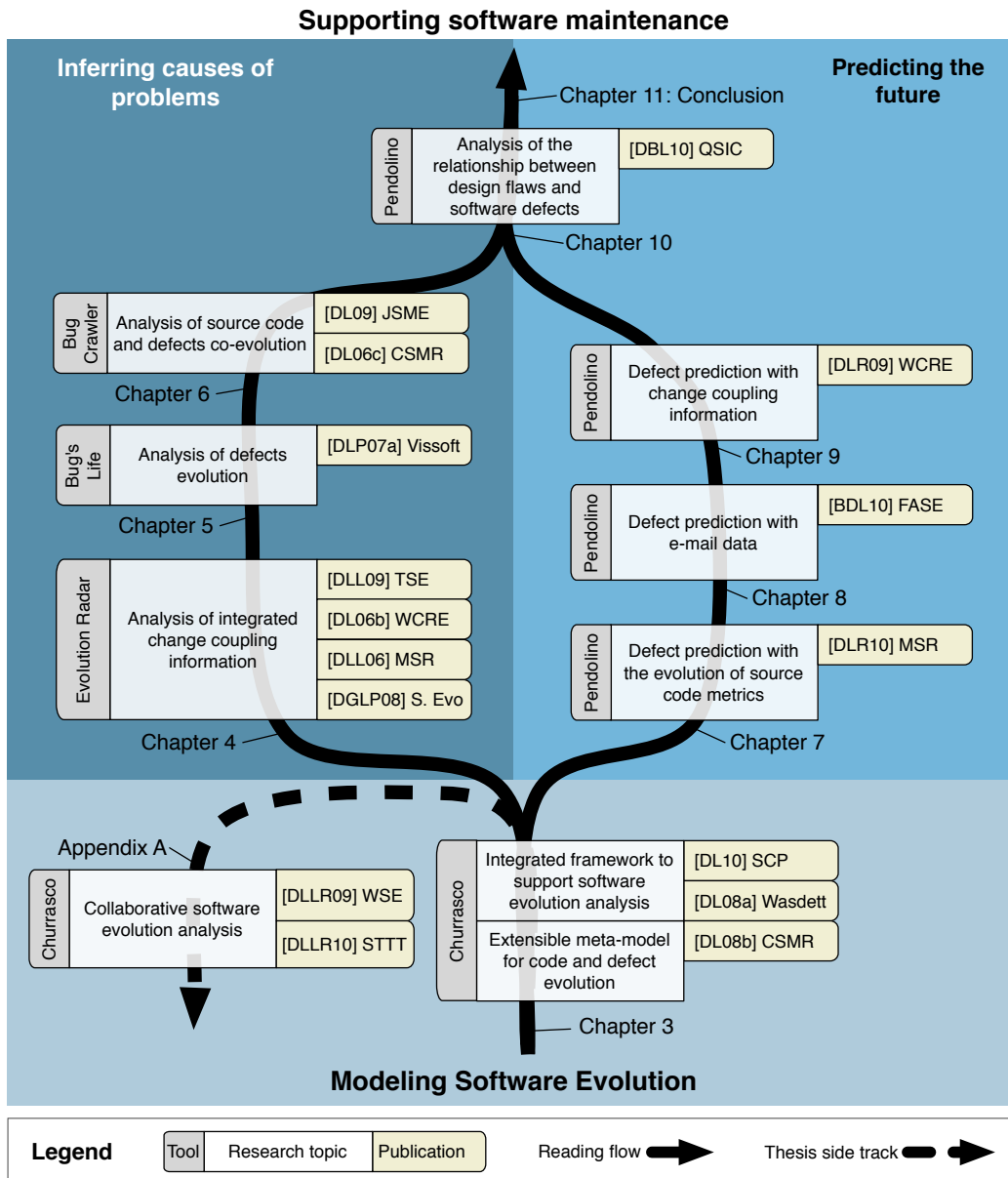


Figure 1.1. The roadmap of our work

- **Chapter 6** (p.103) discusses the co-evolution of source code and software defects. We introduce a visualization technique, called Discrete Time Figure, to study such co-evolution at different granularity levels. Based on the visualization, we define a catalog of co-evolutionary patterns that characterize software artifacts. We apply Discrete Time Figure to three open-source systems, we extract the patterns they contain and we characterize them based on the frequency of the patterns.

Part III: Predicting the Future. In the third part of our dissertation, we present different approaches to predict the future of a software system, and in particular to predict the locations of future software defects.

- **Chapter 7** (p.129) describes two novel defect prediction approaches based on the evolution of source code metrics. We apply the prediction techniques on five open-source systems and compare their performance with several existing defect prediction models. To let other researchers repeat the experiments or compare other approaches with ours, we made the entire processed data set publicly available.
- We claimed that our meta-model and framework are extensible. In **Chapter 8** (p.147), as a proof of concept, we extend them to include e-mail information and we devise a defect prediction technique that uses this new information. We apply the prediction technique on four software systems, showing that it provides an improvement over prediction approaches based only on source code information.
- In **Chapter 9** (p.159) we observe that many studies considered change coupling harmful. However, since there was no research about the correlation between change coupling and software defects, we conduct such a study, providing empirical evidence—on three software systems—that such a correlation exists. Moreover, we show that change coupling data can be used to improve existing defect prediction models based on historical information and source code metrics.
- In **Chapter 10** (p.171) we analyze the relationship between a catalog of design flaws and software defects, to investigate whether certain types of flaws are more harmful than others. We also study the evolution of the flaws over multiple versions of a system, to detect if adding design flaws is likely to generate bugs. By examining six open-source systems, we find out that, while there is no empirical evidence of one flaw being more harmful than others, introducing design flaws is likely to generate defects, with patterns that vary from system to system.

Part IV: Epilogue

- In **Chapter 11** (p.185) we take a step back from the individual analysis techniques by considering our work as a whole. We conclude this dissertation by discussing our approach, summarizing the contributions of this work and outline future research directions.

Part V: Appendices

- **Appendix A** (p.195) discusses how we can support collaboration in software evolution analysis. We present a visualization technique that allows different users to collaboratively analyze a software system and we report on two collaboration experiments performed with students. We build the visualization technique on top of our Churrasco framework. As collaboration is not the main track of the dissertation, we present it in an appendix.
- **Appendix B** (p.209) provides technical details about the Meta-base component of Churrasco [DLP07b]. The Meta-base supports interoperability by providing flexibility and persistence to any meta-model described according to the EMOF specification.

Note. This dissertation makes intensive use of color pictures. We recommend reading it on screen or as a color-printed paper version.

Chapter 2

State of the Art

In our dissertation we analyze different aspects of the evolution of source code and software defects. These analyses are challenging because developers do not write code to support software evolution analysis. For example, source code repositories contain the history of the code, but extracting the evolution of software artifacts is not trivial and in some cases heuristics have to be used. Another example concerns data integration: Many software projects have both a code and a bug repository, but they are disconnected. To know the relationship between the two data sources, *i.e.*, which software artifact is affected by which defects, researchers have to devise linking techniques.

2.1 Genesis of our Approach

Before presenting the research areas most related to our work, it is important to understand why there is a gap between software development and mining software repositories, *i.e.*, why code is not developed to support software evolution analysis. To understand the reasons behind this gap, we look at the history of software evolution.

2.1.1 The Seventies

At the first conference on software engineering in 1968 [NR69] software maintenance was considered a post production activity. The same idea was shared by Royce, who in 1970 introduced the Waterfall life-cycle model for software development [Roy70]. In this model, maintenance is the last phase of the process, after the deployment of the software product.

The seventies were also the decade in which Software Configuration Management (SCM) emerged as a discipline. In 1975, the same year when the first International Conference on Software Engineering (ICSE) was held, Rochkind introduced the first SCM, called Source Code Control System (SCCS) [Roc75]. To be precise, the discipline of configuration management started back in the fifties, when production in the aerospace industry experienced difficulties caused by inadequately documented engineering changes. However, SCCS was the first case in which configuration management was applied to software.

In the second half of the seventies, the waterfall model received the first criticisms. In 1976, Mills argued that software development should be incremental with continuous user participation [Mil76]. One year later, in his book “Software Metrics”, Gilb proposed evolutionary

project management, and introduced the terms evolution and evolutionary to the process lexicon [Gil77]. In 1978, Yau proposed the change mini-cycle model [YCM78], which introduced evolutionary elements in software development process models.

In 1979, Feldman developed the program *Make*, making an important contribution to SCMs.

2.1.2 The Eighties

During the eighties there were three important events: The creation of software evolution as a discipline, the criticisms about the waterfall model and the introduction of evolutionary process models.

In 1980, Manny Lehman in his seminal work [Leh80a; Leh80b] introduced the laws of software evolution. The formulated empirical laws were based on a study to understand the change process being applied to IBM's OS 360 operating system. Lehman confirmed the software evolution laws on other software systems in 1985 [LB85]. Lehman was the first to use the term *Software Evolution* to emphasize the difference with the post-deployment activity of software maintenance. Moreover, he was the first to observe that software must change to adapt to a changing world. Lehman coined the term *E-type software* to stress the difference between software that changes to adapt to the real world—maintaining its usefulness—and software that does not change, and therefore dies because it is not used in the real world. However, it took until the end of the eighties for the term software evolution to be widely accepted [Art88; OL90].

Concerning process models, in this decade researchers proposed two important evolutionary models: Gilb's evolutionary development in 1981 [Gil81] and Boehm's spiral model in 1988 [Boe88]. These models were introduced in the context of a growing criticism to the waterfall model. In 1982, McCracken and Jackson argued against the waterfall model [MJ82] and in 1986 Parnas and Clements stated that, while they believe in the ideal of the waterfall model, they found it impractical [PC86]. Nine years later, in 1995, Brooks marked the end of the waterfall model with his keynote at ICSE titled "The waterfall model is wrong!".

In the meantime, SCM systems continued their growth. In 1982, Tichy introduced Revision Control System (RCS) [Tic82], while four years later Concurrent Version System (CVS), a very popular versioning system (still used in a large number of open source projects and industrial settings) emerged.

2.1.3 From the Nineties to Mining Software Repositories

In the nineties, especially the second half, public awareness of evolutionary development was significantly accelerated. Moreover, it was in the nineties that SCM received widespread attention and usage. With the advent of the Internet and the improvement in network bandwidth, software development started to be distributed, source code repositories started to be remote and CVS played a key role in this transition, as it supports concurrent development. In the nineties, the first bug tracking systems were created: GNATS being the first in 1992, followed by Debbugs in 1994, Bugzilla in 1998, a service offered by SourceForge in 1999 and many others after 2000 (including the Google Code issue tracker in 2007).

In 1995, many contributors from the Rational Corporation created the Rational Unified Process, in which daily build and smoke tests were promoted, a influential practice institutionalized by Microsoft [McC95]. In 2000, Bennet and Rajlich proposed the software development staged model [BR00].

The origin of agile methods and processes took place around the year 2000. In 1996, Kent Beck joined the Chrysler C3 payroll project and in this context extreme programming practices (emphasis on communication, simplicity, testing) matured. It was after this experience that Kent Beck wrote the famous book “Extreme Programming Explained: Embrace Change” [Bec99]. In 1999, another two agile methods were proposed: Stapleton introduced the Dynamic Systems Development Method (DSDM) [Sta99], and De Luca created the Feature Driven Development method (FDD) which was first presented by Coed *et al.* in the book “Java Modeling in Color with UML” [CLL99]. One year later, Beedle introduced Scrum [BDS⁺00], an agile method destined to become famous. In February 2001, 17 process experts from DSDM, XP, Scrum, FDD and others created the Agile Alliance and coined the term “Agile methods”. Later in 2001, one of the participant, Cockburn, published the book “Agile Software Development” [Coc01], while in 2002, Martin wrote about agile software development in a dedicated book [Mar02].

Concerning SCM, two important steps in their growth were the creation of Subversion (SVN) in 2000 (the successor of CVS) and the release of Git in 2006 (an open source distributed version control system). In the second half of the nineties, SCMs became so used and popular that researchers started to mine source code repositories. The first approaches were proposed by Ball *et al.* in 1997 to find clusters of files frequently changed together [BAHS97], by Graves *et al.* in 1998 to compute the effort necessary for developers to make changes [GM98] and by Atkins *et al.* in 1999 to evaluate the impact of tools on software quality [ABGM99]. These are among the seminal research works where the field of mining software repositories has its roots.

2.1.4 The Advent of Mining Software Repositories

In the first half of the current decade, software repositories received more and more attention by researchers and practitioners. The usage of SCM systems became fundamental for software development. Estublier *et al.* stated that “[...] modern SCM systems are now unanimously considered to be essential to the success of any software development project [...] Furthermore, there is a lively international research community working on SCM, and a billion dollar commercial industry has developed” [ELH⁺05]. A number of new bug tracking systems were created (MantisBT, FogBugz, Jira, Savannah, FlySpray, Assembla, *etc.*) and their usage started to become established practice in software development.

In the meantime, software evolution started to be an active and well-respected research field in software engineering, and mining software repositories matured and started to be a research area on its own. In 2004, the first International Workshop on Mining Software Repositories (MSR) was held [HHM04]. In the following years, the topic gained increasing attention and the field continued to mature: Many software engineering and software maintenance conferences had sessions about mining and the international workshop on MSR became a working conference in 2008 [HLG08].

Mining software repositories is not only about source code, SCM and defect tracking systems. In this broad field, researchers mine and analyze a variety of different repositories. To have an impression of the approaches under the umbrella of MSR, we show in Figure 2.1 a word cloud obtained with all the titles and abstracts of the articles presented at the Working Conference on Mining Software Repositories 2008, 2009 and 2010.¹ Figure 2.1 shows that among the most mentioned concepts in MSR are the following: source code, changes, bugs/defects and developers. We model these concepts in our meta-model presented in Chapter 3.

¹We choose the editions 2008, 2009 and 2010 of MSR, because since then it became a conference.

Table 2.1. A summary of the approaches presented at MSR 2008, 2009 and 2010, classified according to the data they analyzed and the goals they achieved

Analyzed data (it can be more than one per approach)	
Versioning system meta data	27
Source code	26
Bug data	17
Email archives	6
Documentation	5
Data recorded in integrated development environments	4
System log files and stack traces	2
Build configuration data	1
Bytecode	1
Effort data	1
Unit tests	1
Work descriptions	1
Achieved goal (one per approach)	
Improving development tools:	13
- Code search tools	4
- Integrated development environments	4
- Recommender system	2
- SCM	2
- Bug tracking system	1
Analyzing developers practices to improve them or to detect erroneous behavior	8
Studying and discovering general principles	8
Creating a common dataset / infrastructure to analyze software repositories	6
Analyzing the impact of source code and SCM properties on bugs	5
Defect prediction	5
Analyzing change properties from SCM	3
Challenges of mining new repositories	3
Extracting and analyzing developer expertises	3
Analyzing log files and stack traces to support debugging	2
Automating bug report classification	2
Analyzing the evolution of the build system	1
Analyzing the replicability of approaches	1
Bug detection	1
Creating social networks of developers	1
Extract structural information from email archives	1
Generating documentation	1
License identification	1
Presenting large amounts of MSR data effectively	1

Apache projects such as Derby, Lucene, Jackrabbit, Maven, Mina, *etc.*) there is a convention to write the id of the bug that was fixed as a comment when committing the changes (*i.e.*, the fix). However, with such a convention, the links between bugs and code are not enforced by tools and their quality and reliability depends on the diligence of developers. To ameliorate this situation, researchers proposed two types of approaches: “during development” and “after development”.

“During development” approaches enhance integrated development environments (IDE) by linking data from multiple repositories and showing relevant artifacts only to the user. One of these approaches is Jazz for Eclipse [Fro07]: It provides an integrated environment with a strong emphasis on collaboration between developers. Jazz integrates source code with defects, defects with failed unit tests, source code with builds and builds with defects. Another integrated solution for Eclipse is Mylyn, a plugin created by Kersten and Murphy [KM05; KM06]. Mylyn models tasks² as first class entities and link them with the source code. It provides plugins to integrate Bugzilla, Trac and Jira repositories, allowing the user to retrieve and store tasks (e.g., bugs to fix) in these repositories.

“After development” approaches integrate evolutionary repositories (such as SCM repositories and bug databases) by analyzing the data they contain. The first approach to link software artifacts with bug reports was the Release History Database (RHDB) [FPG03], followed by others such as Kenyon [BEJWKG05], Hipikat [uMSB05] and softChange [GHJ04]. Since these approaches are closely related to ours, we discuss them in detail in Section 2.2.

2.1.6 Summing Up and Contextualizing our Approach

Figure 2.2 shows an overview of how software evolution, MSR, SCMs and bug tracking systems evolved over the last 40 years. The concept of software evolution was first introduced in 1980, and from then on problems concerning software evolution continued to challenge researchers. The progresses of software evolution were influenced not only by development process models, but also by tools used by developers on a daily basis, such as SCMs and IDEs. In recent times, software evolution became an active research field, and researchers started to mine software repositories to support software evolution analysis.

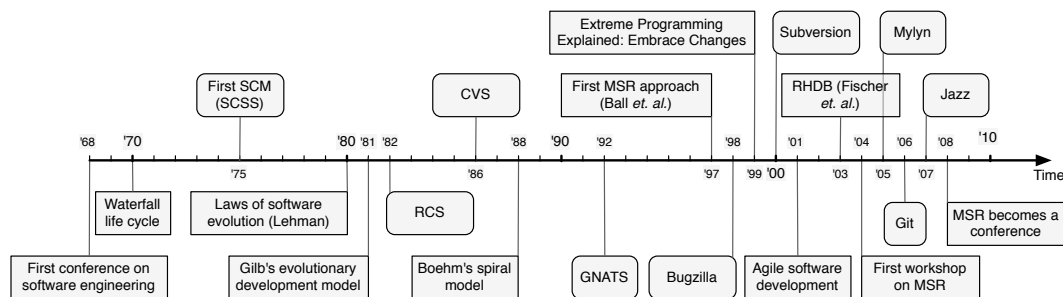


Figure 2.2. An overview of the history of software evolution and MSR

However, there is still a gap between the information produced during software development and the information needed for mining software repositories. One of the problems is the lack of integration between the various pieces of information produced during software development, such as SCM meta-data, source code, problem reports, etc. To tackle this problem, researchers proposed two types of approaches: “During development”, i.e., IDE enhancements/plugins, such as Jazz or Mylyn, and “After development”, i.e., integration for retrospective analysis. Even if we believe that the better and long term solution consists in IDE enhancements, the de facto standard is that in many software projects these solutions are not adopted and the only possibility is to apply “after development” approaches.

²A task can be a variety of activities, among which a bug to fix or a feature to implement.

Our thesis work is located in this context, and can be classified as an “after development” integration approach. Our goal is to show that integrating and analyzing different aspects of source code and defects evolution supports software maintenance activities, aimed at inferring causes of problems and predicting the future of software systems. Figure 2.3 shows the roadmap of our dissertation, where we superimpose the areas of related work, representing them as semi-transparent shapes. Some parts of our thesis are related to more than one research area: In these cases, we place them in the intersection of two shapes.

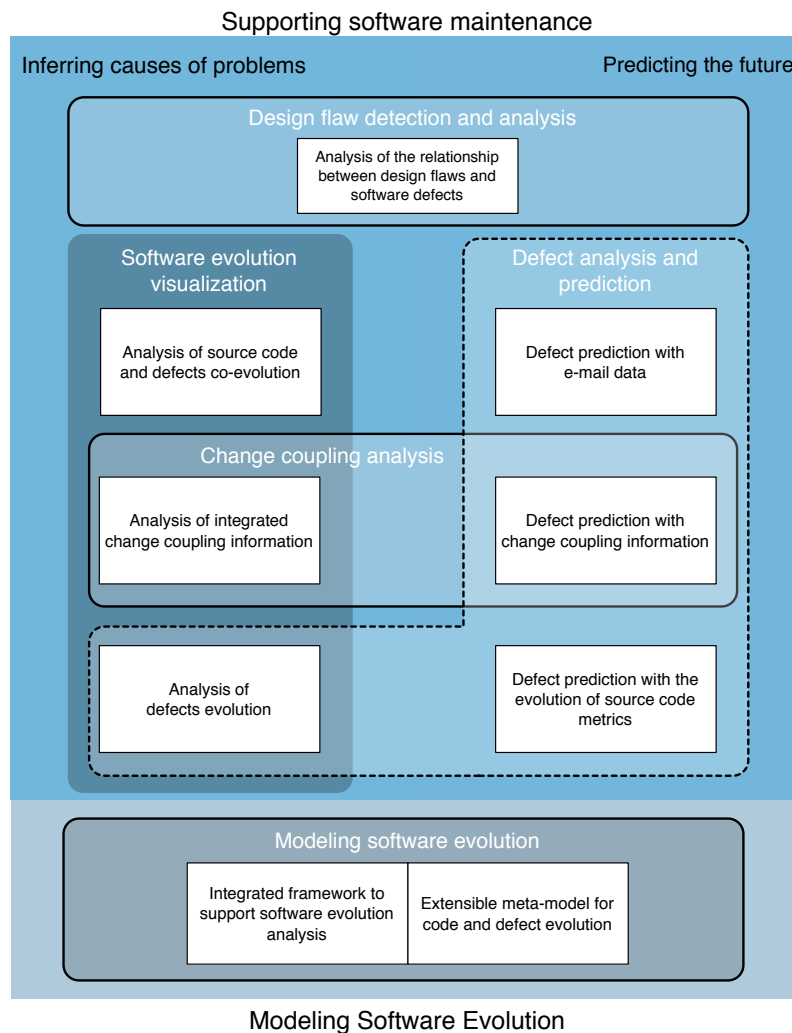


Figure 2.3. Related work in the context of our thesis roadmap

The research areas most related to our thesis work are:

- *Modeling software evolution.* In the first part of our thesis work we create a meta-model of software evolution and we implement it in the context of a software evolution analysis framework. This part of the thesis is related to other approaches to model software evolu-

tion and to create general frameworks for software evolution analysis and mining software repositories.

- *Software evolution visualization.* We devise different software visualization techniques to support the analysis of code, defects and their co-evolution. Related work in this case are visualization approaches that have been proposed as a means to analyze software evolution.
- *Change coupling analysis.* On top of our software evolution meta-model we built a number of analysis techniques. Two of them focus on change coupling information and are related to other approaches aimed at analyzing this type of data.
- *Defect prediction and analysis.* A number of techniques revolve around software defects with different goals: Understanding their properties and their relationship with source code, and predicting density and location of future bugs. We propose several techniques to predict defects and one visual approach to study their evolution.
- *Design flaw detection and analysis.* We exploit our evolutionary meta-model to study the relationship between the quality of the source code and the presence of software defects. We focus our investigation on a certain aspect of code quality, namely the presence of design flaws. Researchers largely studied software quality in general, and design flaws in particular, and proposed several approaches to detect, characterize and analyze design flaws.

In the following we detail the state of the art in the research areas listed above.

2.2 Modeling Software Evolution

Researchers proposed a number of approaches to model the evolution of software, summarized in Table 2.2. As shown in the table, these approaches vary on:

- *The data sources considered.* Some approaches model the source code, others consider also other pieces of data such as bug reports, e-mails, *etc.*
- *The granularity of the evolution.* An SCM repository contains two types of information that can be extracted: Versions (or snapshot) of the software system, *i.e.*, the source code of the system at a particular version, and the SCM meta-data, *i.e.*, the history of every versioned file including the time, author and comment of each commit. Several software evolution approaches model the SCM meta-data without considering snapshots; other approaches model the system's evolution as a collection of snapshots; others combine one or multiple snapshots with SCM meta-data.
- *The purpose.* While every software evolution analysis tool implements a model of software evolution, only few approaches provide a meta-model as a basis to devise analysis techniques on top. In other words, approaches for modeling software evolution vary on how many techniques were proposed on top of them.

Table 2.2. Approaches modeling software evolution

Approach	Modeled data			
	SCM	Snapshots	Bug	Others
CVSgrab & CVSscan	Yes	In [VTvW05]	In [VT06a]	None
Deep Intellisense	Yes	One	Yes	E-mail
EvoOnt	Yes	Yes	Yes	None
Hipikat	Yes	No	Yes	E-mail, documentation
Hismo	Yes	Yes	No	None
Kenyon	Yes	Yes	No	None
RHDB	Yes	In [PGFL05; FG06]	Yes	None
SoftChange	Yes	No	Yes	E-mail
Tesseract	Yes	No	Yes	E-mail

Approach	Techniques on top
CVSgrab & CVSscan	None
Deep Intellisense	None
EvoOnt	iSPARQL [KBT07]
Hipikat	None
Hismo	Yesterday's Weather [GDL04], Ownership Map [GKSD05], Class hierarchies evolution visualization [GLD05]
Kenyon	YARN [HJK ⁺ 07]
RHDB	Features visualization [FG04], RelVis [PGFL05], EvoGraph [FG06]
SoftChange	None
Tesseract	None

Hismo

In his PhD thesis [G05] Gırba defined *Hismo*, a meta-model of software evolution in which history is an explicit entity [GD06]. The main idea of Hismo is to add a time layer on top of every software entity, such as classes, methods and attributes. Like this, software entities have histories composed of all their versions.

Hismo is a generic meta-model, which can describe the history of any entity. In his PhD work [G05] Gırba used it to model the evolution of source code and CVS log files. To model source code evolution he added the Hismo layer on top of the FAMIX meta-model [DTD01]; to model CVS logs he created a Hismo compliant meta-model of versioning system meta-data.

Gırba validated the Hismo meta-model—and its implementation in the context of the Moose reengineering environment [DGN05]—by implementing a number of software evolution analyses on top of it. Gırba *et al.* presented “Yesterday’s Weather” [GDL04], an approach that uses history measurements to detect candidates for reverse engineering activities. The approach is based on the following idea: Entities that suffered important changes in the recent past are likely to change in the near future. Another approach built on top of Hismo is the Ownership Map [GKSD05], a visualization that depicts the code ownerships of all files in a CVS repository over time. The visualization is implemented in a tool called Chronia built on top of Moose. Gırba *et al.* enriched Lanza’s polymetric views [LD03] by adding a time layer on top of them [GLD05], as Hismo is adding a time layer on top of FAMIX. In particular, the authors devised a visualization of the evolution of class hierarchies and, based on the analysis of two large open

source systems, classified class hierarchies based on their history.

Hismo is flexible and can be adapted to model different types of data, as for example source code and CVS log files. However, Hismo does not integrate source code with versioning system meta-data in a unique meta-model, but to describe them Hismo is instantiated in two distinct meta-models. Moreover, the approach does not model software defects.

Release History Database

The Release History Database (RHDB) [FPG03] was the first approach to link versioning system files (from a CVS repository) with bug reports (from a Bugzilla database). The Release History meta-model describes SCM artifacts, problem reports and program features. Fischer and Gall proposed a technique on top of RHDB: The authors used multidimensional scaling to visualize the evolution of features, with the aim of uncovering hidden dependencies between software features [FG04].

Although the original Release History meta-model did not include information about the source code, a number of techniques applied on top of RHDB added this information. Antoniol *et al.* conducted a study on how to integrate the meta-model with a source code meta-model (FAMIX) [APGP05]. Pinzger *et al.* integrated source code data with change coupling information computed from the RDHB [PGF05]. They proposed RelVis, a visualization technique based on Kiviat diagrams, to provide integrated views on source code metrics across different releases together with change coupling information [PGFL05]. The EvoGraph visualization approach [FG06] combines release history data and source code changes to assess structural stability and recurring modifications.

EvoOnt

EvoOnt [KBT07] is a set of software ontologies and data exchange format based on the Web Ontology Language OWL³. EvoOnt is composed of three distinct but interconnected ontology models: (1) the software ontology, describing source code snapshots based on the FAMIX meta-model; (2) the version ontology, modeling SCM meta-data; (3) the bug ontology, describing problem reports according to the Bugzilla meta-model. Since OWL describes the semantics of the data—rather than its structure—EvoOnt can be extended while maintaining the functionalities of existing tools built on top of it.

To complement EvoOnt, Kiefer *et al.* developed iSPARQL [KBT07], a query engine that extends the Semantic Web query language SPARQL⁴ with facilities to query for similar software entities (*e.g.*, classes, methods, attributes) in OWL software repositories. The authors showed that iSPARQL, together with EvoOnt, can be used to perform a number of tasks sought in software repository mining projects, such as code evolution visualization and code smells detection.

Kenyon

Bevan *et al.* proposed the Kenyon framework [BEJWKG05], which provides an extensible infrastructure to retrieve the history of a software project from an SCM repository or a set of releases, and to process the retrieved information. It also provides a common interface based on object-relational persistence to access the processed data and to perform software evolution analysis.

³<http://www.w3.org/TR/owl-ref/>

⁴<http://www.w3.org/TR/rdf-sparql-query/>

Kenyon retrieves information from a number of SCM systems—CVS, SVN and ClearCase—and it models multiple snapshots of a system. The authors designed the framework to be extensible, so that other types of information can be added. In the implementation presented by Bevan *et al.* [BEJWKG05], no other type of data (such as problem reports) was included in Kenyon.

One approach built on top of the framework is YARN [HJK⁺07]. YARN animates the evolution of a software system, based on source code changes, to support the understanding of the system architecture.

CVSgrab & CVSscan

CVSgrab [VT06c; VT06b], proposed by Telea and Voinea, is a visualization tool to analyze CVS based software repositories. The tool allows the user to produce views, to interact with them, to do querying and filtering and to customize the visualizations through a set of metrics computed from the CVS data. While the first version of CVSgrab included only SCM related information, in a subsequent work the authors extended it to model also bug related information [VT06a].

The same authors presented another tool, called CVSscan [VTvW05], that samples multiple versions of a software system from a CVS repository, and visualizes the evolution of lines of code. The authors describe the entire toolset they developed [VT07], which models and visualizes information from SCM meta-data, source code and problem reports. To our knowledge, no analysis technique was built on top of this toolset.

Hipikat and SoftChange

Čubranić *et al.* introduced Hipikat [uMSB05], an approach for linking information from several data sources to form a so called “implicit group memory” (group of developers). The authors showed that such a group memory facilitates the insertion of newcomers in the group, by recommending relevant artifacts for specific tasks. The Hipikat meta-model combines information from versioning systems, bug tracking systems, e-mail archives and documentation.

SoftChange [GHJ04] integrates data from SCM repositories, bug tracking systems and e-mail archives. The data retrieved and processed by softChange is used for two types of software evolution analysis: (1) Statistics of the overall evolution of the project, and (2) analysis of the relationships among files and authors.

The meta-model of both Hipikat and SoftChange does not consider source code snapshots. We are not aware of any approach proposed on top of Hipikat or SoftChange.

Tesseract and Deep Intellisense

Sarma *et al.* developed Tesseract [SMWH09], an interactive visualization tool that displays relationships between versioned files, developers, bugs and communications. The tool provides four cross-linked displays which show: (1) the project activity over time with respect to the number of commits and number of communications; (2) a graph of change coupling dependencies among files, (3) a graph of communication dependencies among developers and (4) the defects affecting the considered files. The meta-model implemented in the tool describes the following pieces of information: SCM files and meta-data, problem reports including communications extracted from their comments, and e-mail data. Developers information is extracted from these data sources (for example from SCM accounts, people sending e-mails or assigned to fix bugs, *etc.*). Tesseract’s meta-model does not describe source code snapshots.

Deep Intellisense [HB08] is a Microsoft Visual Studio plugin that displays information extracted from various software related repositories to help developers understanding the rationale behind the code, *i.e.*, why the code look like it does. Deep Intellisense integrates the following data source: (1) Source code, (2) SCM repositories, (3) bug report, feature request and work item (stored in the same database at Microsoft) and (4) mailing list archives. Both Tesseract and Deep Intellisense are stand-alone tools, and no approach was built on top of them.

2.2.1 Summing Up

In recent years, researchers modeled software evolution in different ways. As discussed in Section 2.1.5, in our thesis we focus on a specific problem of software development tools for MSR, *i.e.*, the lack of integration of software repositories. For this reason, we surveyed software evolution models that integrate different data sources and serve as a basis for subsequent analyses.

RHDB and EvoOnt are the only approaches that model source code snapshots, SCM metadata and software defects. The models, however, were not extended beyond this, for example to model e-mail data or documentation. Only Hismo, Kenyon, RHDB, and EvoOnt were used to build analysis techniques on top, and still the number of techniques is limited. Although some of the surveyed software evolution approaches are flexible, only Hismo and RHDB were adapted/extended to model new pieces of information. In none of the presented approaches, software defects are modeled as first class entities that evolve over time.

2.3 Software Evolution Visualization

Software evolution visualization is software visualization applied to evolutionary information. Software visualization in turn is a specialization of information visualization focusing on software [Lan03].

2.3.1 Software Visualization

According to Stasko *et al.* software visualization is “*the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*” [SDBE98].

The goal of software visualization is to support the understanding of large amounts of data, when the question that one wants to answer about the data cannot be expressed as queries. According to Butler *et al.* [BAB⁺93] there are three categories of visualization:

1. *Descriptive visualization.* The visualized subject is known to a master user and the visualization is used to present the data to other people. Descriptive visualization is widely used for educational purposes.
2. *Analytical visualization.* Defined as the process of looking for something known in the available data. Visualization is particularly useful because it provides the context: For example when visualizing a class hierarchy, the hierarchy is the context for a given class.
3. *Explorative visualization.* Defined as the process of interpreting the nature of the available data. The user does not know (yet) what he is looking for, so he tries to discover recurring patterns or to spot outliers.

Software visualization approaches vary with respect to two dimensions: The type of visualized data and the level of abstraction. According to the type of data, visualizations can be classified as:⁵

- *Static visualizations* present information extracted by static analysis of the software. Their goal is to support the understanding of the structure of the system.
- *Dynamic visualizations* render information that is derived from instrumenting the execution of a program. Their goal is to support the understanding of the behavior of the system.
- *Evolutionary visualizations* present information extracted from the evolution of a software system such as SCM meta-data, differences between system snapshots, *etc.* Their goal is to support the analysis of the causes of problems in the software and the prediction of future changes or problems. All the visualization techniques built on top of our software evolution approach lie in this category.

Since our focus is on evolutionary visualizations, we present them in details, while for static and dynamic visualizations we provide a non comprehensive overview.

Static and dynamic visualizations

Based on their abstraction level, we distinguish three main classes of software visualization approaches: Code-level, design-level and architectural-level.

Code-level visualization. Line-based software visualization was addressed in a number of approaches. In 1992, Eick *et al.* proposed SeeSoft [ESS92], the first tool that uses a direct code line to pixel line visual mapping to represent files in a software system. On top of this mapping, SeeSoft superimposes other types of information such as which developer worked on a given line of code or which code fragments correspond to a given modification request.

Marcus *et al.* extended the visualization techniques of SeeSoft by exploiting the third dimension in a tool called sv3D [MFM03]. In particular, sv3D uses the third dimension to pack the line-based visualization more compactly.

Ducasse *et al.* worked at a finer granularity level, using a character to pixel representation of methods in object-oriented systems. The authors enriched this mapping with semantic information to provide overviews of the methods in a system [DLR05].

The Tarantula tool [JHS02] maps colors on program statements to show their participation to a test suite and the corresponding outcome (passed or failed). Based on this visual mapping, a user can identify statements involved in failures, and locate potentially faulty statements.

Design-level visualization. The next level of abstraction, after code, is the design level, where visualizations focus on self contained pieces of code, such as classes in object-oriented systems.

UML diagrams are the industry standard for representing object-oriented design. A number of tools, such as Rational Rose, ArgoUML, Enterprise Architect,⁶ provide the generation of UML diagrams from code.

⁵This classification is inspired by the taxonomy of software visualization tools presented by Price *et al.* [PBS93]. This taxonomy classifies software visualization tools in three main classes: (1) Algorithm animation, (2) dynamic visualization and (3) static visualization. We adapted this taxonomy by removing algorithm animation and by adding evolutionary visualization, since Price *et al.* proposed the taxonomy in 1993, before software evolution visualization approaches were proposed.

⁶Available respectively at:

Researchers investigated techniques to enrich and extend standard UML diagrams. Termeer *et al.* developed the MetricView tool which augments UML class diagrams with visual representation of class metrics extracted from the source code [TLTC05]. Knodel *et al.* proposed a tool called SAVE [KMNL06] that uses UML-like figures to represent architectural components, allowing the user to navigate from the components to the code. Through a user study, the authors showed that the visualization and the navigation capabilities offered by SAVE support the comprehension of relationships between source code and architectural models [KMN08].

Researchers also investigated different visualization techniques to represent source code at the design level. In his PhD thesis work Lanza introduced *polymetric views* [Lan03], a lightweight software visualization technique. A polymetric view is a representation of software entities and software relationships enriched with software metrics: For example, in the *System Complexity* view [LD03] rectangles represent classes and edges connecting rectangles represent inheritance relationships. The height, width and fill color intensity of class rectangles are proportional respectively to number of methods, attributes and lines of code that the corresponding class has.

Polymetric views can be enriched with dynamic or semantical information. Greevy *et al.* exploited a 3D visualization to add execution trace information to polymetric views in a tool called TraceCrawler [GLW06]. The tool is a 3D extension of CodeCrawler [LDGP05], the tool where Lanza originally implemented polymetric views. Ducasse and Lanza enriched polymetric views with information extracted from control flow analysis in a visualization called *class blueprint* [DL05]. The class blueprint renders classes and, inside them, attributes, methods and call flow information. The catalog of polymetric views presented in Lanza's PhD thesis is not limited to design level visualizations, but includes also architectural level visualizations.

Polymetric views are not the only visualization technique to combine static and dynamic information about a software system. Cornelissen *et al.* proposed a trace visualization method [CZH⁺08] based on a massive sequence and circular bundle view [Hol06], implemented in a tool called ExtraVis. ExtraVis shows the system's structural decomposition (*e.g.*, in terms of package structure) and renders traces on top of it as bundled splines, enabling the user to interactively explore and analyze program execution traces. Cornelissen *et al.* showed that ExtraVis supports three program comprehension tasks: trace exploration, feature location, and feature comprehension. Later, Cornelissen *et al.* conducted a controlled experiment on the usage of ExtraVis to perform eight program comprehension tasks, showing that the group using ExtraVis was faster and produced better results than the group using Eclipse [CZVRvD09].

Ducasse *et al.* proposed a generic visualization technique, called Distribution map [DGK06], to analyze how properties are distributed in a software system. The Distribution Map is based on the notion of focus (whether a property is well-encapsulated or cross-cutting) and spread (whether the property exists in several parts of the system). Given its generality, the Distribution Map technique can be applied in several types of analysis: Ducasse *et al.* applied it to study the distribution of linguistic concepts in software systems and to analyze the distribution of ownership among files.

Another direction of research is the use of metaphors to represent software. Wetzel and Lanza argue that a city is an appropriate metaphor for the visual representation of software systems [WL07a] and implement it in their CodeCity tool [WL08a], where buildings represent classes

-
- Rational Rose <http://www-01.ibm.com/software/awdtools/developer/rose/>
 - ArgoUML <http://argouml.tigris.org>
 - Enterprise Architect <http://www.sparxsystems.com/>

and districts represent packages. Kuhn *et al.* used a cartography metaphor to represent software systems [KLN08]. In their Software Cartographer tool, the authors use a consistent layout for software maps in which the position of a software artifact reflects its vocabulary, and distance corresponds to similarity of vocabulary.

Architectural-level visualization. The highest level of abstraction is the architecture level, consisting of system's modules and relationships among them.

In 1988, Müller and Klashinsky introduced *Rigi* [MK88], the first architectural visualization tool. *Rigi* is a programmable reverse engineering environment that provides interactive visualizations of hierarchical typed graphs and a Tcl interpreter for manipulating the graph data. The tool offers various capabilities for collapsing, expanding and filtering the nodes, navigating the hierarchical models and making layouts. *Rigi* has been a very successful tool [KM10]: Other architecture visualization tools were built on top of it [KC98; OS01] and it inspired other architectural visualization projects. Two of them were *Shrimp* [SM95] and its Eclipse-based continuation *Creole* [LMSW03]. These tools display architectural diagrams using nested graphs where graph nodes embed source code fragments. *Creole* and *shrimp* provide animated panning, zooming, and fisheye-view actions to let the user interact with the visualizations.

Lungu *et al.* introduced *Softwarent* [LL06], an architectural visualization and exploration platform on top of which they experimented with automatic exploration mechanisms [LLG06]. In a subsequent work, the authors extended *Softwarent* to analyze the evolution of the relationships between a system's modules [LL07].

Ducasse *et al.* proposed the Package surface blueprint [DPS⁺07], a visualization approach to study the relationships among the packages of a system. The visualization shows how a package under analysis references other packages, by rendering with different colors class references and inheritance relationships.

2.3.2 Evolutionary Visualization

Researchers employed visualization in software evolution analysis to break down the data quantity and complexity. Table 2.3 presents a summary of evolutionary visualizations classified according not only to the abstraction level—as all software visualization approaches—but also to the visualized evolutionary data. With respect to this second dimension, we distinguish three main classes of evolutionary visualizations: (1) Approaches that extract and visualize information from multiple evolutionary snapshots of a software system, (2) techniques that visualize historical information about software systems as extracted from SCM log files, and (3) visualizations of data retrieved from various repositories such as SCM repositories, problem report databases and e-mail archives.

Approaches based on multiple evolutionary snapshots

Many approaches consider different releases of a software system (evolutionary snapshots) and visualize their history and their differences at various abstraction levels.

Telea *et al.* proposed a code level visualization technique called Code Flows, which displays the evolution of source code over several versions [TA08]. The visualization, based on a code matching technique that detects correspondences in consecutive ASTs [CAT07], is useful to both follow unchanged code and detect important events such as code drift, splits, merges, insertions and deletions.

Table 2.3. Software evolution visualization approaches classified according to the abstraction level and the visualized data sources

Abstraction level	Visualized data		
	Multiple evo. snapshots	SCM log files	Integrated data sources
Code	[TA08]	[BE96]	[FD04; VTvW05]
Design	[GLD05; Lan01; TG02]	[TM02; VRD04; WHH04; GKSD05]	[FG06; CKN ⁺ 03; VT06c; SMWH09; HB08]
Architectural	[GJR99; Jaz02]	[GHJ98; GJK03]	[PGFL05; HJK ⁺ 07]

Several approaches were proposed at a higher level of abstraction: the design level. Girba *et al.* visualized the evolution of class hierarchies and classified them based on their history [GLD05]. Lanza's Evolution Matrix [Lan01] visualizes the system's history in a matrix where each row represents the history of a class, and each column an evolutionary snapshot of the system. A cell in the Evolution Matrix represents a version of a class, where its dimensions are mapped to evolutionary measurements computed on subsequent versions. Tu and Godfrey proposed an approach that integrates the use of metrics, software visualization and origin analysis for studying software evolution [TG02].

Jazayeri *et al.* used a three-dimensional visual representation at the architectural level for analyzing a software system's release history [GJR99]. Later, Jazayeri proposed a retrospective analysis technique to evaluate architectural stability, based on the use of colors to depict changes in different releases [Jaz02].

Approaches based on SCM logs

Another approach to software evolution visualization consists in retrieving the history of a software system from SCM log files.

Working at the level of code, Ball and Eick focused on the visualization of different source code evolution statistics such as code version history, difference between releases, static properties of code, code profiling and execution hot spots, and program slices [BE96].

A number of evolutionary visualizations were proposed at the design level, *i.e.*, rendering SCM files, and able to scale to visualize the entire system. Taylor and Munro used visualization together with animation to study the evolution of a CVS repository [TM02]. The technique, called revision towers, allows one to find out where the active areas of the project are and how work is shared out across the project. Rysselberghe and Demeyer used a scatterplot visualization of CVS data [VRD04] to recognize relevant changes in a software system such as: (1) unstable components, (2) coherent entities, (3) design and architectural evolution, and (4) fluctuations in team productivity. Wu *et al.* used a spectrograph metaphor to visualize how changes occur in software systems [WHH04]. The Ownership Map [GKSD05], introduced by Girba *et al.*, visualizes code ownership of files over time, based on information extracted from CVS logs.

Gall *et al.* used a graph based representation to visualize historical relationships among system's modules extracted from the release history of a system [GHJ98]. The authors applied the visualization to analyze historical relationships among modules of a large telecommunications system and showed that it supported the understanding of the system architecture. Later Gall *et al.* revisited the technique to work at the design level, visualizing classes and historical de-

dependencies among them [GJK03]. They validated the approach on the history of an industrial system, demonstrating that their visualization can be used to detect architectural weaknesses.

Approaches based on integrated data sources

A number of approaches use information from both different releases of a software system and SCM log files.

Augur [FD04] is a code level visualization tool that combines, within one visual frame, information about both software artifacts and the activities of a software project at a given moment (extracted from SCM logs). Although Augur works at the code level, *i.e.*, it visualizes individual code lines, it scales to the design level showing information about multiple files in a single view. Another tool working at the code level is CVSScan [VTvW05] which samples multiple versions of a software system from a CVS repository, and visualizes the evolution of lines of code.

Other approaches target the design level: The EvoGraph visualization [FG06], presented by Fischer and Gall, combines release history data and source code changes to assess structural stability and recurring modifications. Collberg *et al.* proposed a graph drawing technique for visualization of large graphs with temporal component, with the aim of understanding the evolution of legacy software [CKN⁺03].

At a higher level of abstraction, the architectural level, Pinzger *et al.* [PGFL05] proposed a visualization technique based on Kiviat diagrams. The visualization provides integrated views on source code metrics in different releases together with coupling information computed from CVS log files. Hindle *et al.* developed the YARN tool [HJK⁺07], which animates the evolution of the relationships among system's components over time to support the understanding of the system architecture. To generate the animations YARN extracts source code changes from an SCM repository and maps them on software components.

Several evolutionary visualization approaches integrate source code data with information extracted from bug repositories. CVSgrab supports querying, analysis and visualization of CVS based software repositories, integrating also Bugzilla information [VT06c]. Voinea and Telea applied CVSgrab to assess change propagation of buggy files in Mozilla [VT06a]. Two tools that integrate information extracted not only from source code and bug repositories but also from other data sources are Tesseract and Deep Intellisense. Tesseract provides interactive visualizations of relationships between files, developers, bugs and e-mails [SMWH09], while Deep Intellisense integrates source code, SCM repositories, bug reports, feature requests, work items and mailing list archives [HB08]. All the mentioned tools, *i.e.*, CVSgrab, Tesseract and Deep Intellisense, provide visualizations at the design level.

2.3.3 Summing Up

We presented a survey of software visualization approaches pertinent to the research presented in this dissertation. We classified the approaches according to the type of visualized data (static, dynamic and evolutionary) and the abstraction level (code-, design- and architectural-level).

With this survey, we showed that visualization is an effective means to analyze and understand large amounts of data, typical of software evolution studies, where researchers deal with large and long-lived systems. For this reason, in our thesis work we use visualization to analyze the evolution of source code (cf. Chapter 4), software defects (cf. Chapter 5) and their co-evolution (cf. Chapter 6).

2.4 Change Coupling Analysis

Change (or logical) coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system [GHJ98]. This co-change information can either be present in the versioning system, or must be inferred by analysis. For example, Subversion marks co-changing files at commit time as belonging to the same *change set* while in CVS the logically coupled files must be inferred from the modification time of each file.

The concept of co-change in versioning system was first introduced by Ball and Eick [BAHS97]. They used this information to visualize a graph of co-changed classes and detect clusters of classes that often changed together during the evolution of a system. The authors discovered that classes belonging to the same cluster were semantically related.

Gall *et al.* revised the concept of co-change to detect implicit relationships between modules [GHJ98], and named it change (or logical) coupling. They analyzed the dependencies between modules of a large telecommunications system and showed that the concept helps to derive useful insights on the system architecture. Later the same authors revisited the technique to work at a lower abstraction level. They detected change couplings at the class level [GJK03] and validated it on 28 releases of an industrial software system. The authors showed through a case study that architectural weaknesses, such as poorly designed interfaces and inheritance hierarchies, could be detected based on change coupling information.

Other work was performed at finer granularity levels. Zimmermann *et al.* [ZWDZ05] used the co-change information to predict entities (classes, methods, fields, *etc.*) that are likely to be modified when one is being modified. Ying *et al.* proposed an approach, based on data mining techniques, to recommend potentially relevant source code artifacts to a developer performing a modification task [YMNCC04]. The authors showed that the approach can reveal valuable dependencies by applying it to the Eclipse and Mozilla open source projects. Breu and Zimmermann [BZ06] applied data mining techniques on co-changed entities to identify and rank crosscutting concerns in software systems. Bouktif *et al.* [BGA06] improved precision and recall of co-changing files detection with respect to previous approaches. They introduced the concept of change-patterns, in particular the “Synchrony” pattern for co-changing files, and proposed an approach to detect such change-patterns in CVS repositories using dynamic time warping.

The analysis of change coupling has two major benefits:

1. It is more lightweight than structural analysis, as only the data provided by the SCM log files is needed, *i.e.*, it is not necessary to parse and model the whole system. Moreover, as it works at the text level, it can be used to analyze systems written in multiple languages.
2. It can reveal hidden dependencies that are not present in the code or in the documentation.

Since in our dissertation we propose a visual approach to analyze change coupling information, we review other visualization techniques.

2.4.1 Change Coupling Visualization

Ball *et al.* were the first to visualize information about change coupling [BAHS97]. They proposed a static graph visualization technique where nodes represent classes and two classes are connected with an edge if there is a modification report (a commit) in which both the classes changed. The nodes are positioned according to the number of times the corresponding classes

changed together, in such a way that coupled classes are close to each other. The color and shape of a node represents the module the class belongs to.

Ratzinger *et al.* proposed a visualization technique to render the change coupling between java classes, visualizing also module (package) information [RFG05]. Classes are rendered as small ellipse and grouped in larger ellipses representing the packages they belong to. The visualization shows the change coupling among classes through edges connecting the ellipses, whereas the thickness of the edges describes the “strength” of the visualized couplings. The technique does not scale on large systems, since visualizing coupling as edges suffers from over-plotting.⁷

Another change coupling visualization approach, already mentioned as architectural level evolutionary visualization, was presented by Pinzger *et al.* [PGFL05] with Kiviat Diagrams. They represent coupling relationships as edges between modules and use surfaces to depict complete releases. Their work is not the first to represent the coupling as graphs. In fact, the nodes and edges representation was used since the first publications related to change coupling [GHJ98; GJK03]. The drawback of this representation is that it either represents only modules, thus being very coarse-grained, or it represents modules and files, but then incurs scalability and over-plotting problems.

Beyer and Hassan introduced the Evolution Storyboards [BH06]: A storyboard is a sequence of animated panels that shows the files composing a CVS repository, with an energy based layout, where the distance of two files is computed according to their change coupling, similarly to the approach of Ball *et al.* [BAHS97]. The visualization allows the user to easily spot clusters of related files and to compare this cluster with the system decomposition in module, by rendering the module information on the color of the files (files belonging to the same module are rendered with the same color). Each panel is computed according to a particular time period, and the animation in the panel shows how the files move according to how their change coupling changed over the considered time. The visualization is scalable and the authors were able to apply it on large software systems. The Evolution Storyboard does not show the dependencies between modules, but only among files.

2.4.2 Summing Up

From our survey on co-change analysis approaches we draw the following conclusions:

- *Lack of integration.* The main problem with existing techniques is that they work either at the architecture level or at the file (or even lower) level. Working at the architecture level provides high-level insights about the system’s structure, but low-level information about finer-grained entities is lost, and it is difficult to say which specific artifacts are causing the coupling. Working at the file level makes one lose the global view of the system, and it becomes difficult to establish which higher-level consequences the coupling of a specific file has.

To overcome this problem, in Chapter 4 we propose a visualization that integrates change coupling information at a module-level (which modules are coupled with each other) and at a file-level (which files are responsible for the change couplings).

- *Impact on software defects.* Change coupling was considered a bad symptom in a software system: At a fine grained level because a developer who changes an entity might forget

⁷Over-plotting is the problem of multiple visual objects sharing the same space, and thus being positioned on top of one another. This makes it difficult, or even impossible, to distinguish individual objects, threatening the analysis.

to change related entities or, at the system level, because high coupling among modules points to design issues such as architecture decay. Researchers studied change coupling in order to address these two issues, using change recommendation systems and software evolution analysis approaches.

However, no research so far was conducted to empirically assess whether change coupling correlates with the presence of software defects, a tangible effect of design issues. We perform such a study in Chapter 9.

2.5 Defect Prediction and Analysis

Predicting the location and density of software defects in a system was actively researched for more than a decade, with more than a hundred published research papers [MK10]. In the last years, top software engineering conferences (*e.g.*, the International Conference on Software Engineering) and journals (*e.g.*, IEEE Transactions on Software Engineering) featured articles and research tracks on defect prediction.

Defect analysis has recently received increased attention and researchers are especially active developing bug triaging systems and improving bug data and bug tracking systems.

2.5.1 Defect Analysis

With the term defect analysis we indicate approaches where the focus is on defects, and not approaches where defects are “only” a property of another entity being the focus of the analysis. All the techniques that link problem reports with other software related artifacts—to study where problems are in a system [FG04; PGFL05; FG06; VT06a] or to create a comprehensive model of software evolution [uMSB05; GHJ04; SMWH09; HB08]—lie in the latter category, and we already presented them when surveying approaches for modeling (*cf.* Section 2.2) or visualizing (*cf.* Section 2.3) software evolution.

Bug triaging

Researchers exploited the knowledge stored in bug tracking systems to aid bug triaging. Anvik *et al.* presented an approach to semi-automatically assign a developer to a newly received bug report [AHM06]. They apply machine learning algorithms to recommend to a triager a set of developers who may be appropriate for resolving a given bug. A similar approach was presented by Čubranić and Murphy [uM04].

Jeong *et al.* observed that a relevant percentage of problem reports (between 37% and 44% for Mozilla and Eclipse) are reassigned to other developers to get fixed, thus increasing the time needed to fix the bug [JKZ09]. To improve the performances of bug triaging approaches the authors proposed a graph model that describes the “reassignment” history of a problem report. Jeong *et al.* empirically demonstrated—on a dataset of 445,000 problem reports—that their graph model can reduce reassignment events by up to 72%, and improve prediction accuracy of bug triaging approaches by up to 23%.

Weiss *et al.* presented an approach that allows early effort estimation, supporting the task of assigning bugs to schedule stable releases of a software project [WPZZ07]. Given a new bug, the technique predicts the person-hours needed to fix it by first looking for similar (using text similarities techniques), earlier bugs in the bug database, and then by using their average fixing

time as a prediction. The authors showed that, with a sufficient number of bugs, their prediction is close (one hour of deviation only) to the actual fixing time.

Wang *et al.* focused on a specific task that the bug triager has to perform [WZX⁺08]: Given a new problem report, deciding whether it is a duplicate of an existing one. The authors improved existing techniques based on natural language, with execution information. They showed that their improved approach can detect 67–93% of duplicate bugs, compared to 43–72% of previous techniques based only on natural language information.

Bug quality

Researchers investigated the quality of bug reports. Bettenburg *et al.* analyzed the impact of duplicate bugs for the developers in charge to fix them [BPZK08]. The authors found out that, in contradiction to popular wisdom, bug duplicates are not considered as a serious problem for open source projects, and in many cases the additional information provided by duplicates helps to resolve bugs quicker. The authors also showed that this additional information can improve automatic triaging. In a follow-up work [BJS⁺08], Bettenburg *et al.* conducted a survey among developers and users of three large open source projects (*e.g.*, Apache, Eclipse and Mozilla) to find out what makes a good bug report. The analysis of the responses revealed an information mismatch between what developers need and what users supply. To fill this gap, the authors developed a prototype that measures the quality of new bug reports and recommends which pieces of information should be added to improve the quality of the report.

Hooimeijer *et al.* introduced a model of bug report quality, based on a statistical analysis of problem reports extracted from the Firefox bugzilla database [HW07]. Based on their model, the authors showed that some problem report features are more important than others, and they should be highlighted when creating new problem reports.

Antoniol *et al.* showed that a considerable fraction of problem reports marked as bugs in Bugzilla (according to their severity) are indeed “non bugs”, *i.e.*, problems not related to corrective maintenance [AADP⁺08]. The authors presented a text-based classification technique that distinguishes bugs from non bugs, and empirically validated it on Mozilla, Eclipse and JBoss, obtaining a correct classification in approximately 80% of the cases.

Bug history

Few researchers so far analyzed the history of software bugs. Halverson *et al.* presented a number of visualizations [HEDK06] to support the coordination of work in software development, namely the Work Item History and the Social Health Overview. The Work Item History shows status changes of bug reports and makes problematic patterns such as resolve-reopened visible. The Social Health Overview provides an interactive overview of bug reports with drill down capabilities. Bug reports are represented as circles, whereas different bug measures such as the life-cycle-patterns or the bug’s heat can be mapped to the size and color circles.

Aranda and Venolia performed a study of coordination activities around bug fixing from three major product divisions at Microsoft [AV09]. They first queried the bug database—containing rich bug histories—to extract the people involved in the bug fixes, and then contacted and interviewed them. The authors showed that social, organizational, and technical knowledge highly influences the history of bugs, whereas the information stored in bug repository only is not sufficient, and at times even misleading, to support developers coordination for bug fixing.

Summing up

While researchers proposed a number of approaches for bug triaging and to improve bug quality, only little effort was spent so far to model the evolution of bugs and to exploit the information residing in their histories. Moreover, all the mentioned approaches, but the one of Halverson *et al.* [HEDK06], tackle a specific task—such as detecting duplicated bugs—while no approach was devised to support the understanding of a bug repository as a whole and the analysis of bug evolution. We propose such an approach in Chapter 5.

2.5.2 Defect Prediction

Defect prediction approaches vary with respect to three dimensions: The data used for the prediction, the kind of prediction and its granularity. According to the used data we distinguish techniques based on SCM log files data, metrics extracted from the source code and other types of data. The granularity of the prediction can be at either the file/class level, or module/package level. For the kind of prediction, we differentiate between classification and raking techniques.

Classification techniques

The goal of classification techniques is to predict, for each software artifact, whether it will have at least one defect reported. The outcome of a classification approach is a classification table such as Table 2.4, called *confusion matrix*.

Table 2.4. Confusion matrix: The outcome of classification approaches

		Observation	
		Buggy	Non buggy
Prediction	Buggy	True positive	False positive
	Non buggy	False negative	True negative

To assess the performance of a classification model, typically used measures are:

- *Precision*. It is a measure of exactness defined as the ratio between the number of true positives and the number of artifacts predicted as buggy (true positives plus false positives), where a value close to one means that every artifact predicted as buggy actually had defects.
- *Recall*. It is a measure of completeness defined as the ratio between the number of true positives and the number of artifacts that actually had defects (true positives plus false negatives), where a value close to one means that every software artifact that had defects was predicted as buggy.
- *F-measure*. It is a measure that combines precision and recall, defined as their harmonic mean.

- *Accuracy*: It is a measure of classification goodness defined as the ratio between the number of correct classifications (true positives plus true negatives) and the total number of software artifacts, where a value close to one means that the model classified perfectly, without doing miss-classification errors.

Ranking techniques

The goal of ranking techniques is to predict the order of software artifacts according to the number of defects they have (a list where the artifacts with most defects come first). To assess the quality of such a ranking, the typical measures are:

- *The Pearson's correlation coefficient*, which measures the correlation between two variables, assuming a linear relationship.
- *The Spearman's rank correlation coefficient*, which measures the correlation between a predicted and observed ranking, and it is suitable for general associations [Tri06]. In a defect prediction context, the Spearman's correlation is preferable to the Pearson's one, as it is recommended with skewed data [Tri06] (with respect to a normal distribution), a typical characteristic of bug prediction dataset, where most of the software artifacts are non-buggy and only few ones are buggy. A Spearman's correlation close to 1 or -1 indicates an identical or opposite ranking, whereas a value close to 0 indicates no correlation.

SCM approaches

SCM approaches use information extracted from SCM log files, assuming that recently or frequently changed files are the most probable source of future bugs.

Khoshgoftaar *et al.* classified modules as defect-prone based on the number of past modifications to the source files composing the module [KAG⁺96]. They showed that the number of lines added or removed in the past is a good predictor for future defects at the module level. Graves *et al.* devised an approach based on statistical models to find the best predictors for modules' future faults [GKMS00]. The authors found out that the best predictor is the sum of contributions to a module in its history. Nagappan and Ball performed a study on the influence of code churn (*i.e.*, the amount of change to the system) on the defect density in Windows Server 2003. They found that relative code churn was a better predictor than absolute churn [NB05b]. Hassan introduced the entropy of changes, a measure of the complexity of code changes [Has09]. The entropy was compared to amount of changes and the amount of previous bugs, and it was found to be often better. The entropy metric was evaluated on six open-source systems: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. Moser *et al.* proposed a classification technique based on metrics (including code churn, past bugs and refactorings, number of authors, file size and age, *etc.*), to predict the presence/absence of bugs in Eclipse's files [MPS08].

The previous techniques do not make use of the defect archives to predict bugs, while the following ones do. Hassan and Holt's top ten list approach validates heuristics about the defect-proneness of the most and most recently changed and bug-fixed files, using the defect repository data [HH05]. The approach was validated on six open-source case studies: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. They found that recently modified and fixed entities were the most defect-prone. Ostrand *et al.* predict faults on two industrial systems, using change and defect data [OWB05]. The bug cache approach by Kim *et al.* uses the same properties of recent changes and defects as the top ten list approach, but further assumes that faults occur in

bursts [KZWZ07]. The bug-introducing changes are identified from the SCM logs. Seven open-source systems were used to validate the findings (Apache, PostgreSQL, Subversion, Mozilla, JEdit, Columba, and Eclipse). Bernstein *et al.* use bug and change information in non-linear prediction models [BEP07]. Six eclipse plugins were used to validate the approach.

Source code metrics approaches

Defect prediction approaches based on code metrics assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics. One standard set of metrics used is the Chidamber and Kemerer (CK) metrics suite [CK94].

Basili *et al.* used CK metrics on eight medium-sized information management systems based on the same requirements [BBM96]. Ohlsson and Alberg used several graph metrics including McCabe's cyclomatic complexity on an Ericsson telecom system [OA96]. El Emam *et al.* used the CK metrics in conjunction with Briand's coupling metrics [BDW99] to predict faults on a commercial Java system [EMM01]. Subramanyam and Krishnan used CK metrics on a commercial C++/Java system [SK03]; Gyimothy *et al.* performed a similar analysis on Mozilla [GFS05]. Nagappan and Ball estimated the pre-release defect density of Windows Server 2003 with a static analysis tool [NB05a]. Nagappan *et al.* used a catalog of source code metrics to predict post-release defects at the module level on five Microsoft systems, and found that it was possible to build predictors for one individual project, but that no predictor would perform well on all the projects [NBZ06]. Zimmermann *et al.* applied a number of code metrics on Eclipse [ZPZ07].

Other approaches

Ostrand *et al.* conducted a series of studies on the whole history of different systems to analyze how the characteristics of source code files can predict defects [OW02; OWB04; OWB07]. On this basis, they proposed an effective and automatable predictive model based on such characteristics (*e.g.*, age, lines of code) [OWB07].

Zimmermann and Nagappan used dependencies between binaries in Windows server 2003 to predict defect [ZN08]. Marcus *et al.* used a cohesion measurement based on LSI for defect prediction on several C++ systems, including Mozilla [MPF08]. Binkley and Schach devised a coupling dependency metric and showed that it outperforms several other metrics in predicting run-time failures [BS98]. Neuhaus *et al.* used a variety of features of Mozilla (past bugs, package imports, call structure) to detect vulnerabilities [NZHZ07]. In an empirical study of 52 Eclipse plug-ins, Schröter *et al.* showed that design data, such as import relationships, can predict post-release failures [SZZ06].

Pinzger *et al.* empirically investigated the relationship between the fragmentation of developer contributions and the number of post-release defects [PNM08]. To do so, they measured the fragmentation of contributions with network centrality metrics computed on a developer-artifact network. Wolf *et al.* analyzed the network of communications between developers to understand how they are related to issues in integration of modules of a system [WSDN09]. They conceptualized communication as based on developer's comments on work items.

Zimmermann *et al.* tackled the problem of cross-project defect prediction [ZNG⁺09], *i.e.*, computing prediction models from a project and applying it on a different one. Their experiments showed that using models from projects in the same domain or with the same process does

not lead to accurate predictions. Therefore, the authors identified important factors influencing the performance of cross-project predictors.

Bird *et al.* studied a problem common to many bug databases and affecting a number of defect prediction approaches [BBA⁺09]: They observed that the set of bugs that are linked to commit comments (and thus to software artifacts) is not a fair representation of the full population of bugs. Their analysis of several software projects showed that there is a systematic bias that threatens the effectiveness of bug prediction models. However, the technique used to link software artifacts with problem reports [FPG03; ZPZ07] still represents the state of the art.

Summing up

We observe that both case studies and the granularity of approaches vary. Varying case studies makes a comparative evaluation of the results difficult. Validations performed on industrial systems are not reproducible, because it is not possible to obtain the data that was employed. There is also some variation among open-source case studies, as some approaches have more restrictive requirements than others. Concerning the granularity of the approaches, some of them predict defects at the class level, others consider files, while others consider modules or directories (subsystems), or even binaries. While the goal of some approaches is classification, *i.e.*, predicting the presence or absence of bugs for each component, the goal of others is ranking, *i.e.*, predicting the amount of bugs affecting each component in the future, producing a ranked list of components.

These observations lead to the lack of comparison between approaches and the occasional diverging results when comparisons are performed. As a consequence, we identify the need of a benchmark to establish a common ground for comparison. We propose such a benchmark in Chapter 7.

2.6 Design Flaw Detection and Analysis

Researchers devised a number of approaches to address the problem of detecting and correcting design flaws in object-oriented software systems. Marinescu transformed informal design rules, guidelines, and heuristics [GHJV95; Rie96; FBB⁺99] into *detection strategies* [Mar04], metrics-based logical conditions that detect violations against design guidelines. Studying various large-scale software systems, Marinescu provided evidence that these strategies accurately spot design issues in object-oriented programs [Mar04]. Rațiu and Gîrba [RDGM04] applied such detection strategy concept to find design problems revealed by a software system's history. Trifu *et al.* proposed correction strategies to refactor design problems detected using the suite of detection strategies defined by Marinescu [TSG04].

Later, Lanza and Marinescu expanded the detection strategies previously proposed by Marinescu, and—analyzing statistical information from many industrial projects and generally accepted semantics—they deduced many single and combined threshold values for the detection [LM06]. They show in detail how to identify design flaws in code, which solution strategies can be used, and how to devise possible remedies.

Salehie *et al.* proposed a metric-based heuristic framework to detect and locate object-oriented design flaws similar to those illustrated by Marinescu [SLT06].

Wettel and Lanza enriched the CodeCity visualization [WL07b] with design flaws information [WL08b]. Their approach, called *disharmony map*, locates software artifacts that are flawed

according to the detection strategies mentioned above. They display software systems using a 3D visualization technique based on a city metaphor [WL07b], and they enrich such a visualization with the results returned by a number of detection strategies.

Mäntylä and Lassenius conducted an empirical study in a Finnish software product company on the relationship between code smells and software evolvability [ML06]. They measured code smells both automatically, based on program analysis and source code metrics, and subjectively, based on developers' evaluations. The authors concluded that decisions concerning software evolvability improvement should be based on a combination of subjective evaluations and code metrics, as the empirical study showed that they do not fully correlate. Further, Mäntylä in his PhD work [M09; M10] proposed a group of code smells based on a study of 563 evolvability issues found in industrial and student code reviews.

Khomh *et al.* investigated the relationship between code smells and change-proneness [KPG09]. They analyzed multiple releases of Azureus and Eclipse to answer the following question: Are classes with code smells more change-prone than other classes? The results of their empirical study demonstrated that classes with code smells are indeed more change-prone than others, and that specific smells are more correlated than others to change-proneness.

2.6.1 Summing Up

The presence of design flaws in a software system has a negative impact on the quality of the software, as they indicate violations of design practices and principles, which make a software system harder to understand, maintain, and evolve. Design flaws were thoroughly analyzed in literature: To find good metrics and thresholds for their classification [Mar04; LM06; SLT06], to propose correction strategies and refactorings [TSG04; LM06], to visualize them [WL08b], and to put them in relation to code evolvability [ML06] or change-proneness [KPG09]. However, nobody so far analyzed the relationship between design flaws and software defects, a tangible effect of poor software quality. We investigate such a relationship in Chapter 10, conducting an empirical study on six open-source systems.

2.7 Summary

We started this chapter by looking at the history of software evolution, to understand why software development tools are not adequate for software evolution analysis. One of the problems is the lack of integration between the various pieces of information produced during software development, such as SCM meta-data, source code, problem reports, *etc.* To tackle this problem, researchers proposed improvements of development tools (*e.g.*, IDEs, SCMs) and approaches that integrate and model different software repositories to support software evolution analysis. We surveyed these approaches, concluding that:

- *Integration.* Approaches that model and combine different evolutionary aspects are a minority, as most software evolution analysis techniques focus on a single type of data. Moreover, the survey revealed that only one approach integrates source code snapshots, SCM meta-data and software defects [FPG03].
- *Flexibility.* According to our survey, only three approaches were used to build (few) analysis techniques on top, and only two approaches were extended to model new pieces of evolutionary information.

- *Modeling defects.* None of the surveyed approaches models software defects as first class entities that evolve over time.

To overcome these limitations, we propose an approach—presented in the next chapter—that integrates different aspects of source code and defects evolution. On top of this approach, we devise a number of analysis techniques, related to different research areas. We surveyed each of these research areas, identifying potentials for improvement:

- *Change coupling analysis.* We observed that existing change coupling analysis techniques work either at the architecture level, leading to a loss of detailed information, or at the file level, leading to an explosion of the data to be analyzed. None of the existing approaches integrates both levels of granularity.

We also noticed that, while researchers considered the presence of change coupling a bad symptom, nobody empirically assessed whether it correlates with the presence of software defects.

- *Bug Analysis.* Our survey revealed that little effort was spent to model the evolution of bugs and to exploit the information residing in their histories. In particular, no approach was proposed to analyze the evolution of defects.
- *Bug prediction.* Bug prediction techniques vary with respect to the granularity of the prediction (file/class level or module/package level), the task they perform (classification or ranking) and the case studies they were validated on. These differences lead to the lack of comparison between approaches and the need of a benchmark to establish a baseline.
- *Design flaw detection and analysis.* The presence of design flaws in a software system has a negative impact on software quality attributes such as maintainability and evolvability. However, no study was carried out to investigate the relationship between design flaws and the presence of software defects.

As we employ *software visualization* in three of our analysis techniques, we also presented a survey of visualization approaches, showing that visualization is an effective means to analyze large amounts of data, typical of large and long-lived systems.

One last observation is orthogonal to the surveyed research areas and concerns all MSR approaches. In a recent work [Rob10], Robles reviewed all articles (171) published in the proceedings of the international workshop on MSR (2004-2007) and working conference on MSR (2008-2009). He investigated whether the experiments presented in these articles can be replicated. The study revealed that in most of the cases MSR approaches cannot be replicated entirely, leading the author to conclude that *replicability* is a significant issue for MSR in particular, and software evolution analysis in general. In our approach, presented in the next chapter, we also address the issue of replicability.

Chapter 3

Modeling and Supporting Software Evolution

In our thesis work we created an approach that, by modeling software evolution, supports an extensible set of software maintenance tasks. Our approach consists of two parts: In the first one we created a meta-model of evolving software systems, and we devised a framework that implements the meta-model and serves as a basis for analysis. The meta-model and the framework overcome the limitations that we identified when surveying previous approaches. We convert such limitations in requirements, and list them in Table 3.1. In the second part of the dissertation, we propose a number of analysis techniques—on top of our framework—to support several maintenance tasks. In this chapter, we present the first part of our research.

Table 3.1. Requirements for a software evolution meta-model and framework

<i>R1</i>	Integration of different evolutionary aspects.
<i>R2</i>	Flexibility with respect to creating new techniques on top of the approach and extending the meta-model.
<i>R3</i>	Modeling of software defects as first class entities.
<i>R4</i>	Replicability of the analyses performed and availability of the data.

Structure of the chapter. In Section 3.1 we describe how we model an evolving software system, including source code and software defects. In Section 3.2 we present the software evolution framework that we implemented to validate our approach, and we outline the analysis techniques that we built on top of it in Section 3.3. In Section 3.4 we discuss the role of tools in our approach, and we conclude in Section 3.5.

3.1 Modeling an Evolving Software System

There is a number of aspects about a software system that can be considered when modeling its evolution, residing on various repositories, such as defect databases, email archives, versioning system repositories. In our approach we focus on source code and software defects.

3.1.1 Modeling Source Code

Among the various evolutionary aspects we selected source code for two main reasons:

1. While there are many repositories revolving around software projects, many of them contain incomplete or not updated pieces of information. For example, requirements and design documents, use cases and, in general, documentation are often outdated [KC98]; Information that can be extracted from e-mail archives might refer to old system components that were removed or completely restructured. On the other hand, source code is the most reliable source of information: It generates the binaries that run the actual software system, and it is the final product of the development process.
2. Developers spend a considerable fraction of maintenance effort for reading, restructuring and refactoring source code. For example, researchers estimated that software maintainers spend, at least, 50% of their time to understand the source code [FH83; Sta84].

For the mentioned motivations we claim that *any* software evolution meta-model should take the source code into consideration.

In our approach we model source code as an evolving entity, thus modeling its history. To do so, we consider two types of information that can both be extracted from an SCM (or versioning system) repository: Source code snapshots and SCM meta-data recorded by the SCM at commit time. Figure 3.1 shows an example of these two types of information.

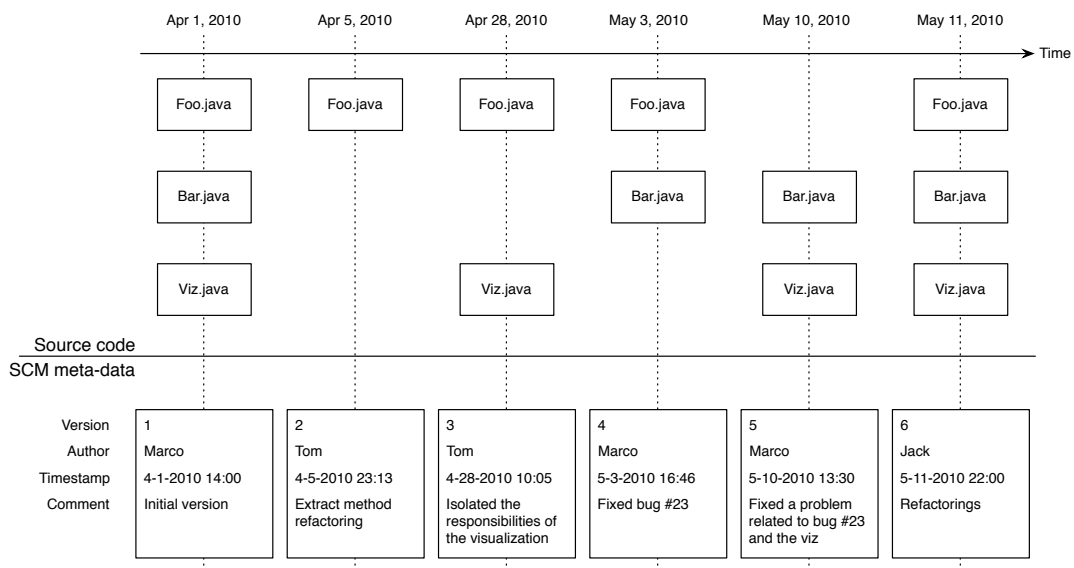


Figure 3.1. An example of source code snapshots and SCM meta-data

Source code snapshots are complete versions of a software system that can be retrieved—for example—from an SCM by performing a “check out” operation. Such an operation can be performed with respect to a particular version number or timestamp. For example, in the scenario depicted in Figure 3.1, one can check out a snapshot at version 3 or at any timestamp between April 28, 2010 - 10:05 and May 3, 2010 - 16:45, obtaining the same system version.

The obtained source code snapshot would include `Foo.java` and `Viz.java` as modified in version 3, and `Bar.java` as committed in version 1.

SCM meta-data consists in information describing commit operations: The version number, who committed the code (and hence performed the changes), the timestamp (when the code was committed) and a comment that the developers can write to describe the performed changes. Figure 3.1 shows some examples of SCM meta-data.

This type of information has the benefit—over source code snapshots—of being lightweight, as only the commits are described. Moreover, it contains information that cannot be found in the source code, such as who modified the code and when. It is also possible to know the size of the change for each file, in terms of number of lines added and removed.¹

On the other hand, SCM meta-data does not provide any information about source code and, to retrieve this data, one has to check out the code from the repository. Checking out a version of a software system every—for example—three days might be very time consuming (especially for large systems) and it might be that the system did not change much in such a short time period. For these reasons we opted for a mixed approach, in which we consider SCM meta-data and bi-weekly system snapshots. In this way, we have all the pieces of information about source code every two weeks, and—in between—the evolution is captured by the SCM meta-data, which also provides information orthogonal to the source code (e.g., authors, timestamps, comments).

Versioning System Meta-Model

The versioning system meta-model describes the evolution of software artifacts as recorded by a versioning system (or SCM). Figure 3.2 shows a class diagram of this meta-model.

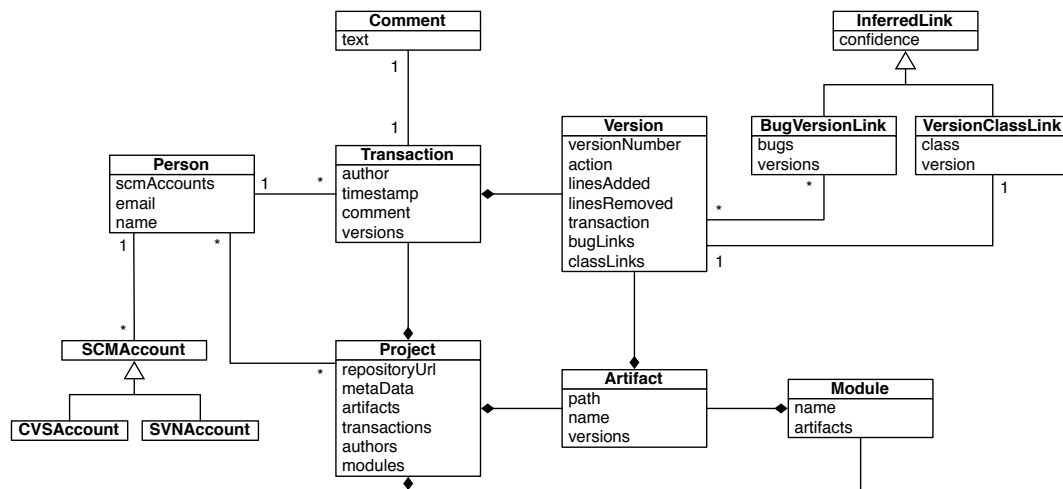


Figure 3.2. A class diagram of the versioning system meta-model

An artifact (typically a file) has a name, a path (the location in the repository) and several versions—one per transaction—where each version can be linked to a source code entity (i.e., a class) and one or more software defects. Since these links are not formally defined and we have to infer them with analysis techniques (detailed later in Section 3.2.2), we decided to model

¹In CVS and SVN a modified line is considered as a line added and one removed.

them with two ad-hoc classes, where we also describe the confidence, *i.e.*, an indication of the quality of the inferred links. Given an artifact, each of its versions has a unique version number, the number of lines added and removed (with respect to the previous version) and an “action” identifier to describe the type of change: Addition or deletion if the artifact is added or removed from the repository, modification otherwise.

A transaction involves a set of artifacts and defines their versions; The transaction is performed at a certain timestamp by an author (modeled with Person), who also writes a comment to document the committed changes. An artifact, in the versioning system, belongs to a project and, if the project is modularized, it also belongs to a module. In some software projects the system modularization (list of modules and artifacts they contain) can be retrieved directly from the versioning system; In other projects we have to find the module decomposition somewhere else (*e.g.*, in the documentation) and manually import it in the versioning system model. In the project class, we also keep information about the url of the repository and the raw meta-data that we use to extract and populate the versioning system model.

The class Person has a particular role in our meta-model, as it is shared by the versioning system and the bug meta-models. A person has a name, can have one or more SCM accounts (account names retrieved from SCM meta-data), and can have an e-mail (retrieved from a bug database). Since we retrieve data from both versioning system repositories and bug databases, we might find the same person represented by two different pieces of information, *e.g.*, an e-mail of a developer assigned to fix a bug and an SCM account. We tackle this aliasing problem, *i.e.*, the same person represented by different pieces of information, by manually inspecting the data and figuring out when it refers to the same person. More sophisticated approaches can be used, as the one proposed by Bird *et al.* based on fuzzy string similarity, domain name matching, clustering, heuristics, and manual post-processing [BGD⁺06].

Source Code Meta-Model

To model source code snapshots, we use the FAMIX meta-model, a language independent representation of object-oriented code, including the concepts of classes, attributes, methods, packages, inheritances, accesses, invocations, *etc.* More details about FAMIX can be found in [DTD01].

3.1.2 Modeling Software Defects

Software defects, often considered as an unwanted “side dish” of the evolution phenomenon, in fact represent a valuable source of information that can lead to interesting insights about a system, that would be hard or impossible to obtain relying exclusively on the source code. In several software projects, developers keep track of defects by means of bug tracking systems such as Bugzilla, Jira, Scarab or Trac.²

Figure 3.3 shows the distribution of ca. a quarter of a million Mozilla’s bugs,³ as recorded in Bugzilla from September 1998 to April 2003, according to their lifetime. We define lifetime as the time elapsed between the moment in which the bug was reported and its last activity (modification of one of its properties). With this example distribution, we want to show that bugs indeed live long, as more than 50% of the considered Mozilla’s bugs lived more than six months. For this reason, we argue that software defects should be modeled as *first class entities*:

²Available respectively at: <http://www.bugzilla.org>, <http://www.atlassian.com/software/jira/>, <http://scarab.tigris.org> and <http://www.mantisbt.org>

³<http://www.mozilla.org>

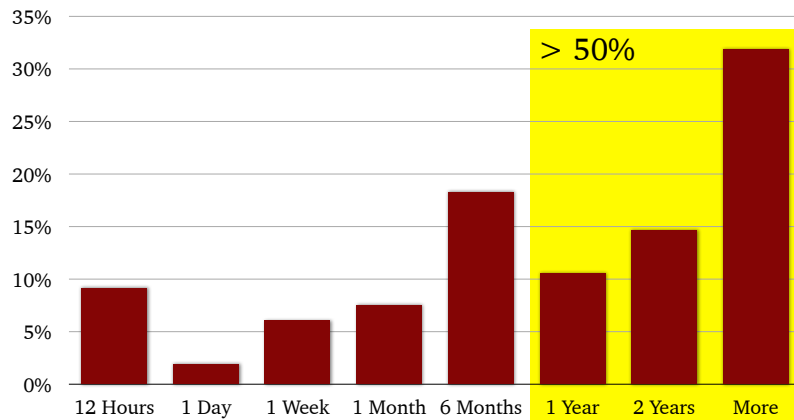


Figure 3.3. Distribution of Mozilla's bugs according to their lifetime. More than 50% of bugs lived more than six months.

We consider not only several properties of bugs (such as the problem description, its severity, the condition in which it was detected, the people involved for fixing / testing it, *etc.*), but also their *histories*, *i.e.*, the sequences of states that they traverse.

Bug Meta-Model

The bug meta-model describes the structure of a software defect. It is an abstraction of the Bugzilla implementation: We chose Bugzilla because it is among the most used bug tracking systems in the open source community, and because the models of other systems, such as Trac, Scarab and Jira, are simplifications of the Bugzilla one.

Figure 3.4 provides an example of a bug report from the Eclipse project. A bug report has a number of properties that we describe in our bug meta-model depicted in Figure 3.5. These properties are: the problem, the criticality, the state in which the bug is, the condition in which the bug was detected, the people involved, the dependencies with other bugs and with the source code and the history of the bug.

The problem. It includes the unique identifier, the (short) description of the problem, a list of keywords describing the problem, when the problem was reported (*creationTimestamp*) and its location in the system. The location is identified by the pair *product-component*, where a product contains several components. Each bug has a list of comments (*BugComment*), describing possible solutions, and a list of attachments such as screenshots or patches.

The criticality. It is indicated by the fixing *priority* (from 1 to 5) and by its *severity*. The possible severities are, in order: *Blocker* (application unusable), *critical*, *major*, *normal*, *minor*, *trivial* (minor cosmetic issue), *enhancement* (request of enhancement).

The state. The state of a bug is composed of its *status* and its *resolution*. The status identifies at which stage of the life cycle the bug is. The possible values are: Unconfirmed, new, assigned, resolved, verified, closed and reopened. The resolution indicates whether and how the problem

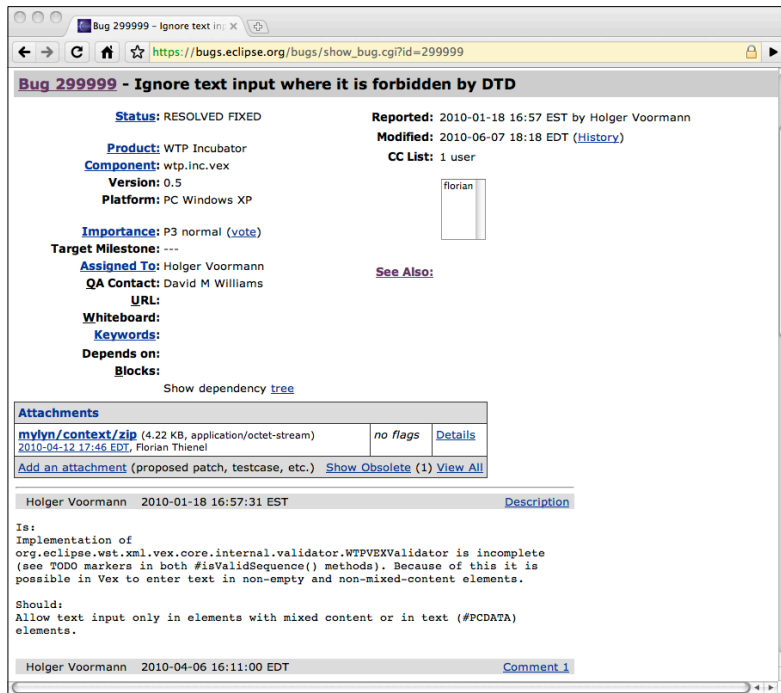


Figure 3.4. An example of a bug report

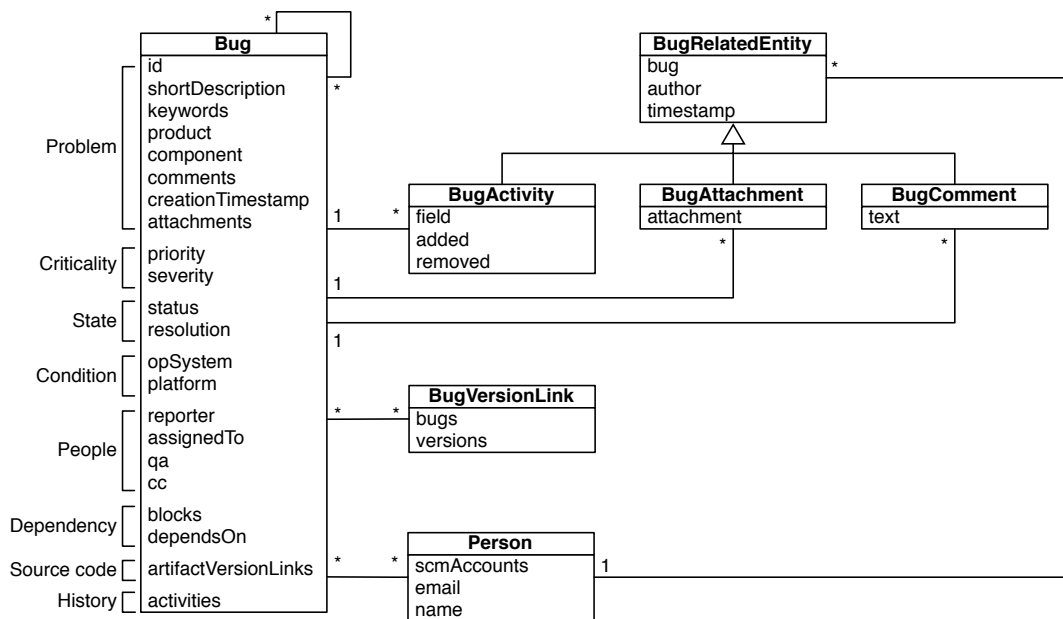


Figure 3.5. A class diagram of the bug meta-model

was solved, once the bug reaches the resolved status. Possible values here are: *Fixed*, *invalid*, *won't fix*, *not yet*, *remind*, *duplicate*, and *works for me*.

Figure 3.6 shows all possible bug statuses and transitions, based on the Bugzilla system.⁴ Each possible status is associated with a different color. Green colors represent statuses in which the bug is considered fixed (i.e., *resolved*, *verified* or *closed*), while red colors represent statuses in which the bug has to be fixed (i.e., *new*, *assigned* or *reopened*). We consider *reopened* as the most critical status, since a first attempt did not fix the bug. *Unconfirmed* is associated with cyan, since it is not known yet if the reported bug is real. In the remainder of this dissertation, and in particular in Chapter 5, we use the same color mapping for bug statuses.

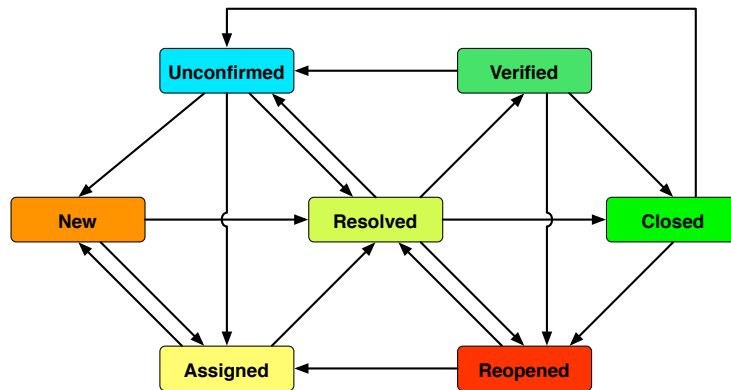


Figure 3.6. The Bug Status Transition Graph

The typical life cycle of a bug is the following: It is reported (either *new* or *unconfirmed* according to the privileges of the reporter), it is assigned to a developer for fixing (*assigned*) and then he/she proposes a solution (*resolved*) or decides that a solution is not needed (for example when the resolution is set to duplicate or invalid or won't fix). When the bug is *resolved*, the quality assurance tests the proposed solution and sets the bug status to one of the following: *Verified*, *closed*, *reopened*, *unconfirmed*. The bug status transition graph does not have a final state, because a bug can always be reopened.

The condition. The settings in which the bug was detected, e.g., *operating system* and *platform*.

The people. The set of people involved includes the *reporter* of the bug, the developer in charge to fix it (*assignedTo*), the quality assurance (*qa*) person who test the solution and a list of people who are interested in being notified of the bug fixing progress (*cc*).

The dependencies. Bugs compose graphs: Each bug *b* can have a list of other bugs that need to be fixed before *b* can get fixed (*dependsOn*). Symmetrically, each bug *b* can have a list of other bugs that can be fixed only after *b* get fixed (*blocks*).

The source code. Bugs might be linked with the software artifact they affect. These links are important because they act as bridges between the versioning system and the bug meta-models. As previously mentioned when discussing the versioning system meta-model, bug-version links are not formally defined and hence we use a confidence parameter to model this fact.

⁴See <http://www.bugzilla.org/docs/2.18/html/lifecycle.html>

The history. Our meta-model takes time into account. Every field of a bug can be modified over time thus generating a bug activity (*BugActivity*). The activity records which field is changed (*field*), when (*timestamp*), by whom (*author*) and the pair of old and new values (*added* and *removed*). Activities are important because they allow us to keep track of a bug's history and life cycle, *i.e.*, the sequence of statuses the bug went through. Through bug activities we fulfill our requirement R3: Modeling of software defects as first class entities that evolve over time and thus have histories.

3.1.3 Mevo: A Meta-Model of Evolving Software Systems

We discussed how to model versioning system data, source code and software defects in three distinct meta-models. Now we combine them in Mevo, a unique meta-model of evolving software systems, depicted in Figure 3.7. Other approaches for modeling software evolution exist (*e.g.*, Hismo [GD06], RHDB [FPG03; APGP05], Kenyon [BEJWKG05], Hipikat [uMSB05], Tesseract [SMWH09]): We extensively discussed them when surveying related work in Section 2.2.

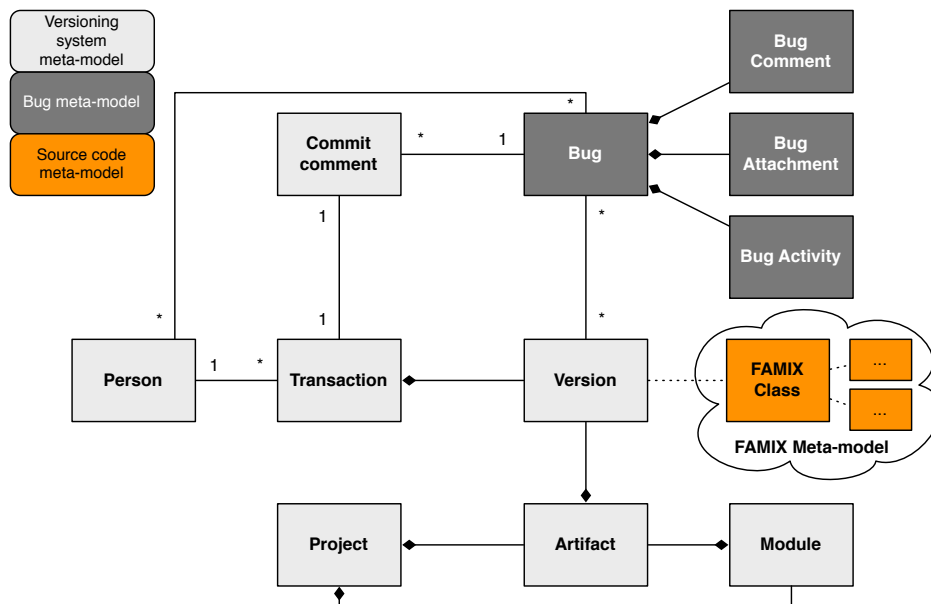


Figure 3.7. Mevo: A meta-model of an evolving software system. It is composed of three linked parts: The versioning system, the source code and the bug meta-models.

With the Mevo meta-model we fulfill our requirements for modeling an evolving software system. We wanted our meta-model to integrate different evolutionary aspects (requirement R1, cf. Table 3.1): Mevo combines the history of a software system (as recorded by a versioning system), multiple snapshots of the source code and information about software defects affecting the source code. The meta-model is extensible (requirement R2): We show in Chapter 8 how Mevo can be augmented to model e-mail data. Finally, in Mevo we model software defects as first class entities that evolve over time (requirement R3), and we model their histories by means of bug activities.

3.2 Our Approach in Action

We presented our meta-model to describe evolving software systems. Now we discuss how we use such a meta-model in practice, *i.e.*, how we populate its instances and which analysis techniques we build on top of it. Figure 3.8 provides an overview of our approach “in action”, showing the two phases that compose it: data retrieval and pre-processing, and applications.

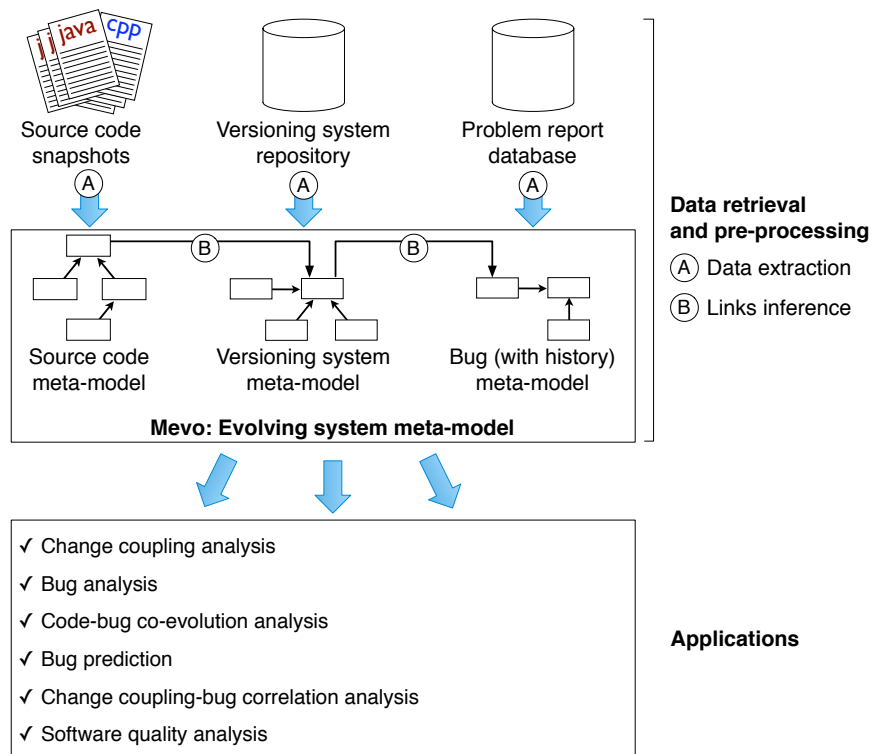


Figure 3.8. The overall view of our approach in action, divided in data retrieval and pre-processing (links inference) and applications

Before being able to do any analysis, we need to retrieve and pre-process the data, *i.e.*, to populate the Mevo meta-model. First we extract and process the data from object-oriented source code, versioning system repositories and bug databases. Table 3.2 shows which languages, which versioning systems and which bug tracking systems we support in our implementation. The data is then stored, according to the Mevo meta-model specification, into a database. Then we link the different parts together, *i.e.*, we integrate the three parts of the meta-model.

Table 3.2. Languages and technologies supported in our approach implementation

Supported languages	C++, Java, Smalltalk
Supported versioning systems	CVS, SVN, Store [Sto00]
Supported bug tracking systems	Bugzilla, Jira

3.2.1 Retrieving the Data and Populating the Models

We obtain multiple source code snapshots of a software system by either downloading them from the system web site or by performing “check out” operations (one per snapshot) from the versioning system repository. Once we obtained the source code snapshots, we parse them with inFusion (for Java and C++) or Moose⁵ [DGN05] (for Smalltalk), obtaining FAMIX models that we use to populate the corresponding part of Mevo.

To populate the versioning system part of Mevo, we use different approaches: For CVS and SVN we parse the versioning system log files, which provide all the pieces of information about the history of all versioned artifacts. Since SVN logs—as opposed to CVS logs—do not contain information about lines added and removed, we have to do some additional data extraction for SVN: We perform a SVN diff for each version of each file to compute the lines added and removed. In the case of Store, historical information about versioned software artifacts is available as Smalltalk objects: We process these objects and populate the versioning system model.

Finally, to instantiate the bug meta-model, we query bug databases with a web interface (available for both Bugzilla and Jira), which provides bug data in XML format. However, as the web interface does not provide bug activity information in XML format, to import the data concerning bug histories we have to parse HTML pages. When importing bug data, we filter out bug reports whose resolution is marked as “duplicate” or “invalid”. We also disregard bugs whose severity is “enhancement” because they are not related to corrective maintenance.

3.2.2 Pre-processing and Linking the Data

Before linking the three parts of the Mevo meta-model, if the versioning system from which we imported the data is CVS, we need to pre-process the data. In Mevo, we model the concept of *transaction* and each transaction has a list of artifact versions. While SVN and Store mark co-changing artifacts at commit time as belonging to the same transaction, in CVS the artifacts committed in the same transaction must be inferred from the modification time of each of them.

Reconstructing transactions

To reconstruct the transactions, we take into account the following commit data: CVS account name, comment and timestamp. Given two or more commits, a necessary (but not sufficient) condition for them to be considered in the same transaction is that the user names and the comments coincide. We consider them to be in the same transaction if also a time condition holds. For that, two possible techniques are the “fixed time window” and the “sliding time window” approach, proposed by Zimmermann and Weißgerber [ZW04] (depicted in Figure 3.9):

1. In the fixed time window approach, the beginning of the time window is fixed to the first commit (artifact₁, version 1.1). All other commits with a timestamp included in the window are considered to be in the same transaction (only artifact₂ version 1.4).
2. In Zimmermann’s sliding window approach, the beginning of the time window is moved to the most recent commit recognized to be in the transaction. By doing this, artifact₃ version 1.2 is also included in the transaction. The transactions reconstructed using this approach include commits taking longer than the size of the time window. We use a time window of 200 seconds as proposed by Zimmermann and Weißgerber [ZW04].

⁵inFusion and Moose are available at <http://www.intooitus.com> and <http://www.moosetechnology.org>

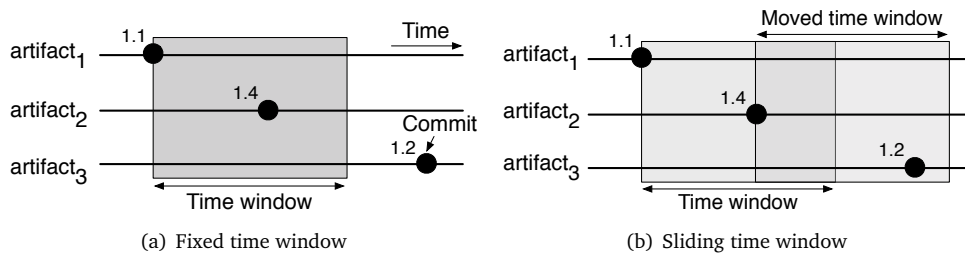


Figure 3.9. Reconstructing CVS transactions: Fixed and sliding time window

Linking source code and versioning system data

After pre-processing the versioning system data, *i.e.*, reconstructing CVS transactions, we link the versioning system part of Mevo with the source code and the bug parts, respectively with the version-class and bug-version relationships. Through these two relationships, software bugs and source code classes are also (indirectly) linked. However, Figure 3.10 shows that, in our implementation, the linking is possible only for certain combination of languages, versioning systems and bug tracking systems. On the other hand, it is always possible to populate individual parts of the meta-model, as for example the source code or the versioning system part only.

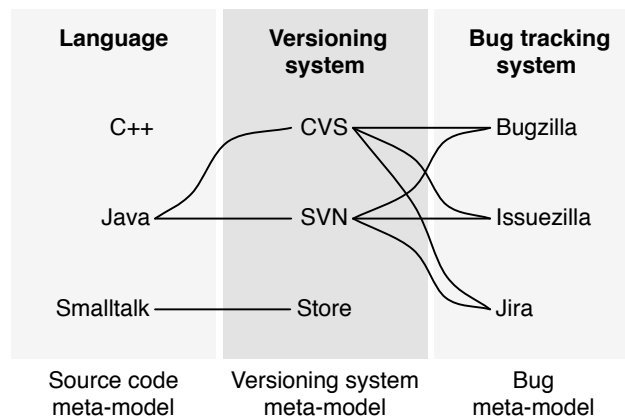


Figure 3.10. Possible linkings among the different parts of the Mevo meta-model in our implementation

As Figure 3.10 shows, in our implementation it is not possible to link C++ classes with any versioning system artifact. Concerning Smalltalk and Store, the linking is straightforward as classes and versions, both available as Smalltalk objects, are already linked to each other.

To link Java classes with CVS or SVN artifacts, we use an approach based on the correspondence between the directory structure in the versioning system and the package nesting structure (or fully qualified name) in Java. With this approach, the class Layout shown in Figure 3.11 is translated in the following path to be found in the versioning system repository: `../org/uml/ui/Layout.java`.

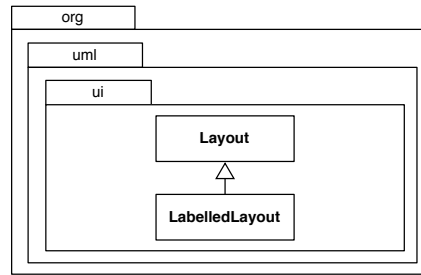
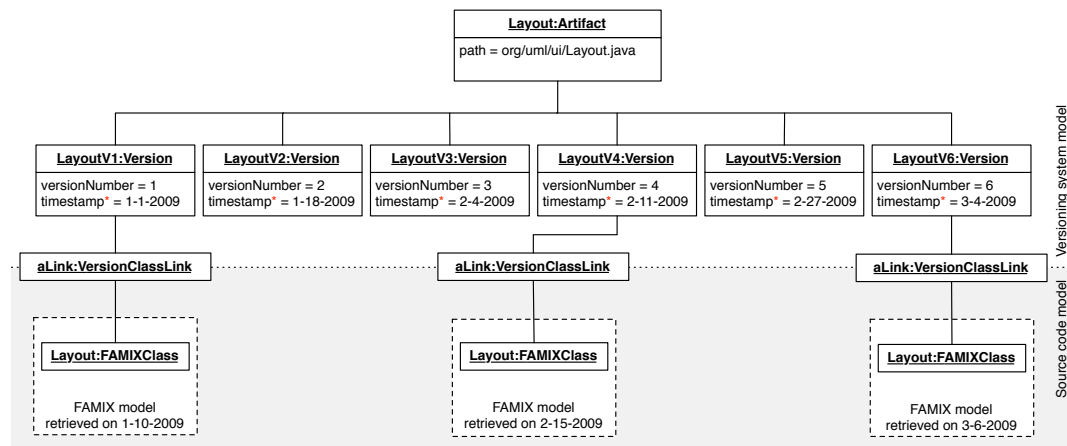


Figure 3.11. An example of package nesting. With our linking approach the Java class `Layout` is translated in the path `./org/uml/ui/Layout.java` to be found in the versioning system repository.

In the Mevo meta-model we consider multiple source code snapshots: For this reason, in a model we can have multiple FAMIX classes representing multiple versions of a class over time (one for each snapshot in which the class existed). We link each FAMIX class with an artifact version of the versioning system model. The selection of the artifact version is based on the timestamp of the transaction in which the version was committed. Let `Foo` be a FAMIX class and t_{FAMIX} the timestamp of the source code snapshot containing `Foo`: The artifact version `FooVer` that we link to `Foo` is the most recent one⁶ for which the following condition holds

$$t_{\text{FooVer}} \leq t_{\text{FAMIX}} \quad (3.1)$$

where t_{FooVer} is the timestamp of the transaction in which `FooVer` was committed. Figure 3.12 shows an example of linking between multiple versions of a FAMIX class and artifact versions.



* timestamp is not an attribute of the class `Version`, but of the class `Transaction`. To make the diagram simpler and more readable we include timestamp directly in `Version`

Figure 3.12. Linking multiple versions of the `Layout` FAMIX class with artifact versions of the versioning system model

⁶The most recent artifact version is the one committed in the most recent transaction.

Linking versioning system and bug data

The last part of the linking process deals with bugs and artifact versions, *i.e.*, the connection between the versioning system and the bug meta-models, represented by the class BugVersionLink (cf. Figure 3.2).

An artifact version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (*e.g.*, “fixed bug 12345”). Such references allow us to link problem reports with versioning system artifacts, and therefore with source code entities, since versioning system artifacts and FAMIX classes are already linked. However, the link between a CVS / SVN artifact and a Bugzilla / Jira problem report has not yet been formally defined. To find a reference to the problem report id, we use pattern matching techniques on the developer comments, an approach widely used in practice [FPG03; ZPZ07]. To choose which regular expression to use to detect the links, we apply it on sample comments taken from the software project under study. Since some bug references are just plain numbers, without keywords such as “fix” or “bug”, it is possible to obtain false positives. Thus, in our algorithm, each time we find a candidate reference to a bug report, we check that a bug with such an id exists, and that the date in which this bug was reported is before the timestamp of the commit comment in which the reference was found (a simple sanity check that the bug is fixed *after* being reported, cf. Figure 3.13).

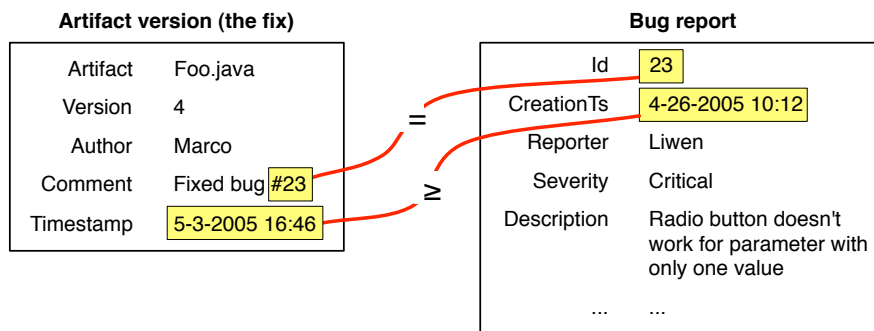


Figure 3.13. Linking versioning system artifacts with bug reports: We first detect bug ids in commit comments and then check that the fix was committed (commit timestamp) after the bug was reported (bug creation timestamp).

As depicted in Figure 3.2, the class BugVersionLink has three attributes: Confidence (inherited from InferredLink), bugs and versions. We use the confidence attribute to model the uncertainty of the links, *i.e.*, the fact that they have to be inferred with heuristics. Possible values for the confidence range from 1 to 10, with a default value of 6. Bugs and versions model the many-to-many relationship between artifact versions and bug reports: In fact, a developer can fix multiple bugs in the same commit, thus writing a commit comment such as “fixed bugs #123 and #130”. On the other hand, bugs can be fixed more than once, *i.e.*, in different commits: It is the case of re-opened bugs. For such bugs it is not clear, without additional knowledge, whether the first attempt to fix the bug was wrong—and thus “overwritten” by subsequent fixes—or was right but incomplete—thus “completed” by subsequent fixes. In our approach, when we find multiple commits fixing the same bug, we perform the following operations:

1. We check that the fixes, *i.e.*, the commits having the bug reference in the comment, were

the only ones committed between two re-opened events or between the creation and a re-opened events (see Figure 3.14). In case there are multiple fixes between two re-opened events, we filter them out, as the data is not consistent (a bug must be re-opened before being fixed again).

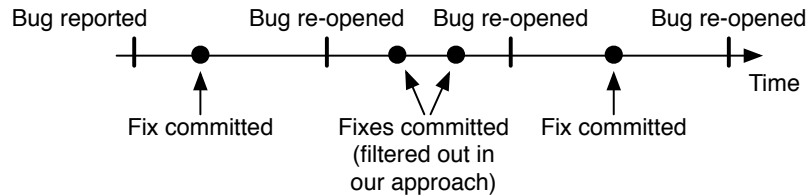


Figure 3.14. Linking fixes (commits fixing a bug) with a bug re-opened several times: If there are multiple fixes between two re-opened events, we filter them out, as the data is not consistent (a bug must be re-opened before being fixed again).

2. We link all the fixes (satisfying the previous condition) to the bug report, by means of the `BugVersionLink` class.
3. We decrease the confidence value of `BugVersionLink` proportionally to the number of fixes, *i.e.*, the greater the number of commits fixing a bug, the lower the confidence of each link between the fix and the bug. The rationale behind this choice is that the more attempts were required to fix a bug, the lower the impact of each attempt.

3.2.3 Limitations

Since the data sources from which we are populating our meta-model were not designed for this purpose, there are a number of technical issues that have to be considered.

Generality vs Implementation. Although the Mevo meta-model is general and independent from the language (as long as it is object-oriented), the versioning system and the bug tracking system, our approach supports only certain languages (Java, C++ and Smalltalk), versioning systems (CVS, SVN and Store) and bug tracking systems (Bugzilla and Jira). Moreover, the complete meta-model—including the versioning system, the bug and the source code parts—can be exploited only with the Java language and the CVS or SVN versioning system, as illustrated in Figure 3.10. In fact, in our implementation, the linking between source code entities and versioning system artifacts works for Java and Smalltalk (not for C++), but for Smalltalk / Store there is no link with a bug tracking system. For this reason, in our thesis work, we mainly focus on software projects developed in Java using CVS or SVN, with few case studies in Smalltalk / Store and none in C++.

Java Inner Classes. A second issue comes from the file-based nature of versioning systems such as CVS and SVN, and the way inner classes are defined in Java. In fact, they are defined in the same file as the container class, and so the same versioning system artifact might point to several classes (the container class and the inner classes). For this reason, in our approach we filter out Java inner classes.

Renaming. In the Mevo meta-model the identity of a versioning system artifact is based on its name. In the versioning system, renaming an artifact (or moving it to a different directory) is seen as an addition of a new artifact (with the new name) and a removal of an old artifact (with the old name). In our approach, we do not handle renaming, *i.e.*, we model it as the versioning system, with an addition and a removal. Researchers proposed techniques to manage the identity of software artifacts, robust to renaming events [APM04; KN06]. However, we did not implement these techniques in our approach.

Linking Bugs. Another technical issue concerns linking versioning system artifacts with bugs: The pattern matching technique we use to detect bug references in commit comments does not guarantee that all links are found. In fact, all the links that do not have a bug reference in a commit comment cannot be found with our approach. Bird *et al.* recently studied this problem in bug databases [BBA⁺09]: They observed that the set of bugs which are linked to commit comments is not a fair representation of the full population of bugs. Their analysis of several software projects showed that there is a systematic bias that might threaten the effectiveness of techniques validated on bug datasets, such as bug prediction models. However, the approach based on pattern matching represents the state of the art in linking bugs to versioning system artifacts [FPG03; ZPZ07].

Non Bugs. Antoniol *et al.* showed that a considerable fraction of problem reports marked as bugs in Bugzilla (according to their severity) are indeed “non bugs”, *i.e.*, problems not related to corrective maintenance [AADP⁺08]. To tackle this issue, we manually inspected statistically significant samples of bugs linked with versioning system artifacts, and found that—in all the analyzed software projects—more than 95% of them were real bugs. This is not in contradiction with the findings of Antoniol *et al.* [AADP⁺08]: Bugs mentioned as fixes in commit comments are intuitively more likely to be real bugs, as they got fixed. As a consequence, the impact of this issue on our experiments is limited.

Large Commits. The way developers use versioning systems impacts the quality of our models. In particular, one problem that might pollute our data consists in large commits, *i.e.*, transactions involving a large number of files. We distinguish two types of large commits: The first one—involving a relatively small amount of files—groups two or more conceptual changes, while the second one concerns a single conceptual change but involves hundreds of files.

We illustrate the first type with the following example: A developer modified the files `Foo.java` and `Bar.java` to fix the bug 745, and the files `Boo.java` and `Baa.java` to add a new feature. He then committed all the changes together (in the same transaction), thus grouping two conceptual changes: a bug fix with an addition of features. The problem with these commits concerns the linking between software artifacts and bugs. In the previous example—supposing that the developers wrote “Fixed bug 745” as a commit comment—our approach would link the bug 745 not only with `Foo.java` and `Bar.java` (correct) but also with `Boo.java` and `Baa.java` (wrong). As a consequence, we rely on developers’ meticulousness for the quality of our data.

Regarding the second type of large commits, a representative example is the license update, which can involve all source code files. The problem with these commits deals with co-change analysis, when we count how many times software artifacts change together. One possible solution is to filter out the transactions that involve more than a certain number of artifacts (*e.g.*, 50 or 100 depending on the size of the system). However, since the choice of the best threshold

depends on the system and on the type of analysis that one wants to conduct (and might also require some manual inspection), we do not perform the filtering during the pre-processing of the data, but leave it as part of the subsequent analyses.

Table 3.3. Commit size in several software projects

Project	# commits	% 1 file	% 2-4 files	% 5-9 files	% 10+ files
Ant	14,078	96.25	3.35	0.27	0.13
Django	4,812	87.43	12.57	0	0
Gcc	87,900	47.26	40.64	6.64	5.46
Gimp	23,215	91.68	7.85	0.33	0.13
Glib	5,684	88.79	10.66	0.44	0.11
Gnome-desktop	4,195	89.92	9.58	0.38	0.12
Gnome-utils	6,611	80.34	19.32	0.33	0.02
Httpd	39,801	56.17	40.76	2.89	0.17
Inkscape	14,519	90.92	8.83	0.22	0.03
Jakarta	70,654	77.43	20.74	1.64	0.18
Jboss	5,962	95.67	4.29	0.03	0
KDE	817,795	78.48	20.59	0.83	0.10
Lucene	14,078	80.52	18.45	0.93	0.10
Ruby on Rails	9,251	96.25	3.35	0.27	0.13
Spamassassin	10,270	91.17	8.26	0.50	0.08
Subversion	21,729	50.26	47.83	1.70	0.21
Total	1,158,824	75.69	22.41	1.38	0.52

Table 3.3 presents statistics gathered from 16 open-source software projects. The table shows that (1) most of the commits ($\sim 75\%$) involve one file only, (2) about 22% of the commits are changes of two, three or four files, (3) while only less than 2% of the commits involve more than four files. From this data we conclude that the impact of large commits (especially the second type) is limited.

3.3 Applications

Once we populated our Mevo meta-model, we can proceed with the subsequent analyses on top of it. In the following, we briefly introduce the analysis techniques, whereas we detail each of them in a dedicated chapter in the remainder of the dissertation.

Change coupling analysis (cf. Chapter 4)

Meta-model: The versioning system part of Mevo

Tool: Evolution Radar

We devise a visualization-based approach that integrates co-change information (change coupling) at different levels of abstraction: The module level, to understand the relationships among system's modules, and the file level to uncover the causes of the couplings. The analysis technique supports the retrospective analysis of a software system and maintenance activities such as restructuring and re-documentation.

Bug analysis (cf. Chapter 5)

Meta-model: The bug part of Mevo

Tool: Bug's Life

We propose a visualization approach to support the analysis of software defects and their evolution. The technique is composed of two visualizations: The first one—aimed at studying the bug database in the large—is helpful to understand how bugs are distributed in system components and over time. The second visualization supports bug analysis in the small, *i.e.*, the inspection of bugs affecting one or few software components. The visualization facilitates the characterization of bugs and the identification of the most critical ones.

Code-bug co-evolution analysis (cf. Chapter 6)

Meta-model: The versioning system and bug parts of Mevo

Tool: Bug Crawler

We create a visual approach to uncover the relationship between evolving software and the way it is affected by software defects. By visually putting the two aspects close to each other, we characterize the evolution of software artifacts at different granularity levels. On top of the visualization we define a catalog of co-evolution patterns that, due to their formal definition, can be automatically detected in a software system.

Bug prediction (cf. Chapter 7)

Meta-model: Mevo

Tool: Pendolino

We devise two defect prediction techniques based on the evolution of source code metrics, and we compare their performance with well-known bug prediction approaches. For the comparison, we propose a benchmark in the form of a publicly available data set consisting of several software systems.

Bug prediction with e-mail data (cf. Chapter 8)

Meta-model: An extension of Mevo

Tool: Pendolino

We investigate whether information extracted from development mailing lists can be used to predict post-release defects or to improve existing defect prediction models. To do so, we extend Mevo to model e-mail data, thus showing its flexibility.

Change coupling-bug correlation analysis (cf. Chapter 9)

Meta-model: The versioning system and bug parts of Mevo

Tool: Pendolino

Researchers studied change coupling and observed that it points to design issues such as architectural decay. We analyze the relationship between change coupling and a tangible effect of design issues, *i.e.*, software defects. We investigate whether change coupling correlates with defects, and if the performance of bug prediction models based on software metrics can be improved with change coupling information.

Software quality analysis (cf. Chapter 10)

Meta-model: Mevo

Tool: Pendolino

The presence of design flaws in a software system has a negative impact on the quality of the software, as they indicate violations of design practices and principles. We study the relationship between software defects—tangible effects of poor software quality—and a number of design flaws, finding that no design flaw can be considered more harmful than the others. We also analyze the correlation between the introduction of new flaws in a software component and the generation of defects affecting that component.

3.4 Tool Support

Analyzing the evolution of large and long-lived software systems requires extensive tool support due to the amount and complexity of the data that needs to be processed. Our approach is unavoidably tied to the tools that we developed to implement our techniques in practice.

We created a framework called *Churrasco* [DL08b], which implements the Mevo meta-model, with the following goals:

- *Meta-model population.* Churrasco supports the population of Mevo in batch mode, hiding the data retrieval and pre-processing tasks from the users. To create a model, *i.e.*, import a software project to be analyzed, Churrasco provides a web interface.
- *Base for analysis.* The framework serves as a basis for the analysis, as it stores all the models in a centralized database and it provides an interface that allows us to build analysis tools on top of it.

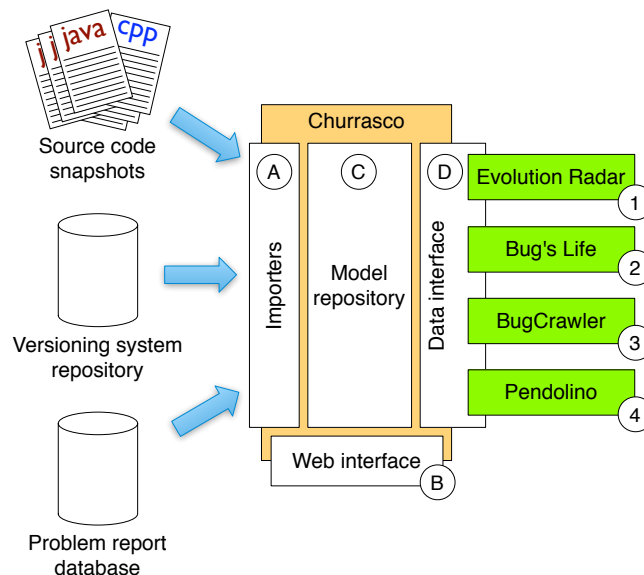


Figure 3.15. The main components of the Churrasco framework and the analysis tools built on top of it

Figure 3.15 shows the main components of the Churrasco framework: (A) The importers, which retrieve and pre-process the data from the various data sources, (B) the web interface, which allows the users to create new models, (C) the model repository, where the Mevo meta-model is implemented and all the models are stored, and (D) the data interface, which allows us to build analysis tools on top of Churrasco. Figure 3.15 shows also the analysis tools that we implemented on top of Churrasco, and which implement the applications of our approach that we present in the following chapters:

1. *The Evolution Radar* visualizes integrated change coupling information, supporting the analysis of the coupling at different levels of abstraction, *i.e.*, the module and the file level.

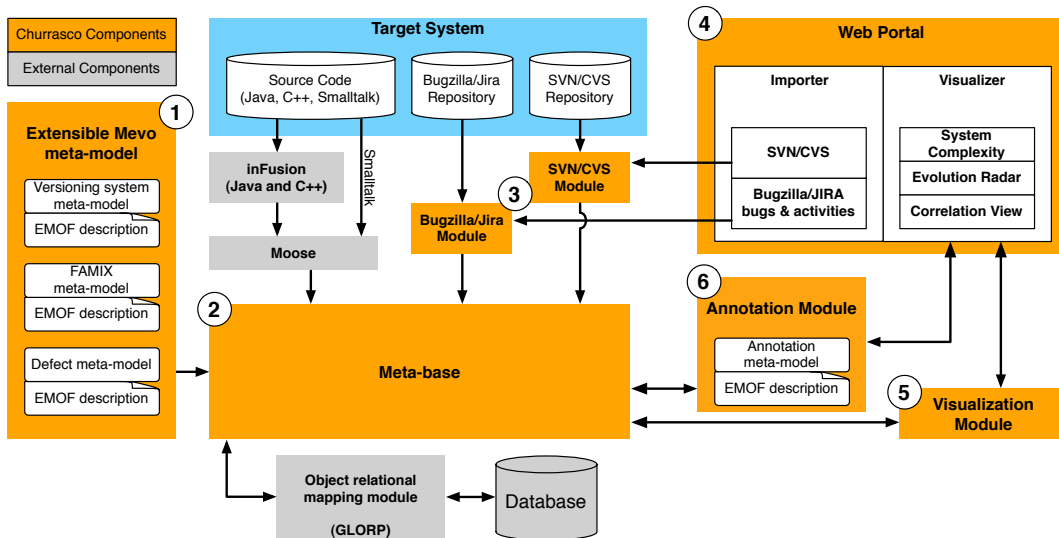


Figure 3.16. The architecture of Churrasco

2. *Bug's Life* is a visualization tool that supports the analysis of bugs and bug histories in the large (at the system level to identify critical components) and in the small (at the component level to identify the most critical bugs).
3. *BugCrawler* provides interactive visualizations of code and bugs co-evolution, and it supports the automated detection of co-evolutionary patterns.
4. *Pendolino* is a scriptable data analysis tool that serves as a bridge to Matlab or other statistical tools. To act as a bridge, *Pendolino* computes and exports a variety of metrics and properties (e.g., source code metrics, historical metrics, bug metrics, number of design flaws, change coupling measures, etc.) about Mevo models in a format that can be imported in Matlab. We subsequently use the metrics in Matlab to create and evaluate regression models for different tasks, such as defect prediction.

3.4.1 Churrasco's Architecture

Figure 3.16 depicts Churrasco's architecture, consisting of:

1. *The extensible Mevo meta-model*. It can be extended using the facilities provided by the Meta-base module.
2. *The Meta-base* supports flexible and dynamic object-relational persistence. It uses the external component GLORP [Kni00] (Generic Lightweight Object-Relational Persistence), providing object-relational persistence, to read from/write to a database. The Meta-base also uses the inFusion tool and the Moose reengineering environment to create a representation of the source code (C++, Java or Smalltalk) based on the FAMIX meta-model. For software systems written in Java or C++, we first parse them with inFusion and then we import them in Moose and in the Meta-base. For systems developed in Smalltalk, we directly parse them in Moose and import them in the Meta-base.

3. *The Bugzilla/Jira and SVN/CVS modules* retrieve and process the data from Bugzilla/Jira and SVN/CVS repositories.
4. *The Web portal* represents the front-end of the framework accessible through a web browser.
5. *The Visualization module* supports software evolution analysis by creating and exporting interactive visualizations.
6. *The Annotation module* supports collaborative analysis by enriching any entity in the system with annotations. It communicates with the web visualization module to depict the annotations within the visualizations.

The Meta-base

Churrasco's *Meta-base* [DLP07b] provides flexibility and persistence to *any* meta-model, and in particular to Mevo. It takes as input a meta-model described in EMOF and outputs a descriptor that defines the mapping between the object instances of the meta-model, *i.e.*, the model, and tables in the database. EMOF (Essential Meta Object Facilities) is a subset of MOF,⁷ a meta-meta-model used to describe meta-models.

The Meta-base ensures persistence through the object-relational module GLORP. By generating descriptors of the mapping between the database and the meta-model, the Meta-base can be adapted dynamically and automatically to any meta-model. This allows us to modify and extend dynamically any meta-model. For more details, we refer the reader to Appendix B.

The SVN/CVS and Bugzilla/Jira modules

The SVN/CVS and Bugzilla/Jira modules retrieve and process data from, respectively, SVN/CVS and Bugzilla/Jira repositories. They take as input the URL of the repositories and then populate the models by means of the Meta-base component. They are initially launched from the web importer (discussed later) to create the models, and then they automatically update all the models in the database every night, with the new information (new commits or bug reports).

The SVN/CVS module populates the versioning system model, by checking out the project with the given repository, creating and parsing SVN/CVS log files. Checked out snapshots of the system are then used to create FAMIX models using the inFusion and Moose external components.

The Bugzilla/Jira module retrieves and parses all the bug reports (in XML format) from the given repository. Then, it populates the corresponding part of the defect model. Finally, the module retrieves all bug activities from the given repository. Since Bugzilla and Jira do not provide this information in XML format, Churrasco parses HTML pages and populates the corresponding part of the model.

The web portal

The web portal is the front-end of Churrasco, developed using the Seaside framework [DLR07]. It allows users both to create the models, and to analyze them by means of different web-based visualizations. To create new models and access the visualizations, the user has to log in the web portal.

⁷MOF and EMOF are standards defined by the OMG (Object Management Group) for Model Driven Engineering. The specifications are available at: <http://www.omg.org/mof/>

(a) The importer page

Selected project details	
Name	argouml
Time period	1/26/98 - 4/7/09
Svn files	13920
Svn commits	15257
Svn repository url	http://argouml.tigris.org/svn/argouml/trunk
Bugs number	5800
Bugzilla url	
FAMIX models	argouml-0.10 - argouml-0.12 - argouml-0.14 - argouml-0.16 - argouml-0.18.1 - argouml-0.20 - argouml-0.22 - argouml-0.24

(b) The project list page

Figure 3.17. The Churrascope web portal

Figure 3.17(a) shows the importer web page of Churrasco, ready to import the ArgoUML software project. The information needed to create a model is the URL of the SVN (or CVS) repository and the URLs of the Bugzilla (or Jira) repository (one URL for bug reports, one for bug activities). Since, depending on the size of the software system to be imported, retrieving the data can take a long time, the user can also indicate an e-mail address to be notified when the importing is finished.

Figure 3.17(b) shows the project list page of Churrasco, which contains a list of projects available in the database, and—for a selected project—information such as the number of files and commits, the time period (time between the first and last commit), the number of bugs, a collection of FAMIX models corresponding to different versions of the system, *etc.* The page also provides a set of *actions* to the user, *i.e.*, links to the web visualizations provided by Churrasco.

The visualization module

This module offers a set of interactive web-based visualizations. Their goal is different from the goals of the tools built on top of Churrasco: The visualizations are aimed at supporting collaboration in software evolution analysis. For this reason, we present them in details in Appendix A, where we discuss collaborative software evolution analysis.

Figure 3.18 shows an example visualization rendered in the Churrasco web portal. The main panel is the view where all the figures are rendered as SVG graphics.⁸ The figures are interactive: Selecting one of them will highlight the figure (red boundary), generate a context menu and show the figure details in the information panel on the left. Churrasco provides other panels useful to configure and interact with the visualization that we present in Appendix A.

The annotation module

The idea behind Churrasco's annotation module is that each visualized model entity can be enriched with annotations to let different users collaborate in the analysis of a system. We discuss the use of annotations to support collaboration, together with two collaboration experiments, in Appendix A.

3.4.2 Discussion

With the Mevo meta-model, we already fulfilled some requirements that we selected for a software evolution analysis approach: The integration of different evolutionary aspects (*R1*) and the modeling of software defects as first class entities (*R3*).

By means of the Churrasco framework, we fulfilled the remaining requirements. Churrasco provides a flexible and extensible meta-model support (part of requirement *R2*), which allows us to change and extend the meta-model definition dynamically. This was helpful in the past, when we changed the Mevo meta-model to add new pieces of information (for example bug activities), and it allowed us to extend Mevo with e-mail data, as presented in Chapter 8. Churrasco is also flexible with respect to creating tools on top of it (the remaining part of requirement *R2*): The framework provides a data interface to access the information it stores, *i.e.*, Mevo models. We exploited this interface in creating a number of tools, *e.g.*, the Evolution Radar, Bug's Life, Bug Crawler and Pendolino.

⁸SVG (Scalable Vector Graphics) is a declarative XML-based language for vector graphics specification. It is available at: <http://www.w3.org/Graphics/SVG/>

The screenshot shows the Churrascode web portal interface. The browser address bar displays the URL: `http://evo.inf.unisi.ch:8018/seaside/Churrascode?_s=OlvkjacQrotmkDT&k=npkCPdmi`. The page is titled "Episode System complexity" and shows the user is logged in as "Marco D'Ambros".

On the left sidebar, there are several sections:

- Recent annotations:** Shows an annotation by Michele Lanza on November 14, 2008, at 2:34:04 am, regarding a hierarchy with TokenType, Lexer, and Parser subclasses.
- Participants:** Lists Michele Lanza and Marco D'Ambros.
- Create pdf report:** Includes a "Download the report" button.
- System Complexity:**
 - Target system: argoUML-0.24
 - Selected figure information:**

Name	JavaRecognizer
Type	Churrascode.SFAMIXClass
# attributes	146
# methods	91
WLOC	3406
 - Metrics mapping:**

Node width	# attributes
Node height	# methods
Node color	WLOC
 - Apply view on packages:** Lists application, application:api, application:configuration, application:events, application:helpers, and application:modules.
 - Regular expression matcher:** Includes "Clear selection" and "Spawn selection" buttons.

The main visualization area shows a complex UML diagram. A callout "Interactive Visualization" points to the diagram. A callout "User" points to the top right corner. A callout "Selected figure" points to a highlighted node in the diagram. A callout "Context menu" points to a menu with "Show annotations" and "Add annotation" options.

Figure 3.18. A screenshot of the Churrascode web portal showing a visualization of ArgoUML

The last requirement to fulfill is the replicability of the analyses performed and the availability of the data (R4). The Churrascode framework, together with all the models it contains, is publicly available on the web. The analyses and the experiments that we present in the following chapters can be replicated, as the needed data can be downloaded from the Churrascode web portal. In some cases (e.g., for bug prediction and for software quality analysis) we even provide dedicated benchmarks to make the replication of the experiments easier.

Other features of Churrascode include the annotation and collaboration support, discussed in Appendix A.

3.5 Summary

The approach presented in this dissertation is composed of two main parts: Modeling different aspects of software evolution and analyzing them to support an extensible set of software maintenance activities.

In this chapter, we presented the first part of the approach. We introduced Mevo, a meta-model that integrates three aspects of the evolution of a system: (1) multiple versions of the source code, (2) the history of software artifacts, as recorded by a versioning system, and (3) software defects with their histories. We discussed how to retrieve these pieces of information from distinct software repositories, and how to link them in a unique Mevo model.

To apply our approach in practice, we developed Churrasco, an extensible framework that implements the Mevo meta-model and serves as a basis for the subsequent analyses. We described Churrasco, its architecture and its main components.

At the beginning of this chapter, based on limitations identified in previous approaches, we defined four requirements for modeling and analyzing software evolution (cf. Table 3.1): (*R1*) integration, (*R2*) flexibility, (*R3*) modeling defects, and (*R4*) replicability. We discussed how the Mevo meta-model and the Churrasco framework fulfill these requirements. Mevo integrates various types of evolutionary information (*R1*) and models defects as first class entities (*R3*). Churrasco provides a flexible meta-model support, and a data interface to create tools on top of it (*R2*). The framework also offers a web portal from which all the models used in our analyses and experiments can be retrieved, thus facilitating replicability (*R4*).

We briefly introduced the analysis techniques we devised on top of our approach, mentioning also the tools that implement them. These techniques constitute the second part of our thesis work, and thus we discuss them in detail in the remainder of this dissertation, together with their validation.

Part II

Inferring Causes of Problems

To analyze the evolution of software, one first has to model it. In the previous part of this dissertation, we introduced Mevo, a meta-model of evolving software systems that integrates source code, SCM meta-data and software defects information. We implemented such a meta-model in Churrasco, an extensible framework that serves as a basis for building various analysis techniques.

In this part of the thesis, we present three such analysis techniques, aimed at detecting causes of problems in software systems. These techniques focus on different aspects of software evolution, i.e., the different parts of the Mevo meta-model, and support its analysis by means of interactive visualizations.

In Chapter 4 we focus on change coupling, evolutionary dependencies between source code artifacts. We present a visualization technique, called Evolution Radar, that supports the analysis of change coupling information at different levels of abstraction. Subsequently, in Chapter 5, we shift our attention on the evolution of software defects, introducing a visual approach to study a bug repository at different levels of granularity: In the large, visualizing the entire repository, and in the small, rendering individual bugs. Finally, in Chapter 6, after focusing on the evolution of code and defects in isolation, we analyze their co-evolution by means of a dedicated visualization called Discrete Time Figure. On top of the visualization, we define a catalog of co-evolutionary patterns that characterize the evolution of software entities.

Chapter 4

Analyzing Integrated Change Coupling Information

In the previous chapter, we introduced Mevo—our meta-model of evolving systems—and Churrasco, a framework that implements our approach to support various software evolution analysis techniques. In this chapter, we present one of the analysis techniques built on top of Churrasco. The technique focuses on the evolution of source code artifacts—as recorded by a versioning system and modeled in Mevo—and processes this evolutionary information to detect change coupling relationships.

Change coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system [GHJ98], although they are not necessarily structurally related (for example by means of inheritance, subsystem membership, usage, *etc.*). They are therefore linked to each other from a development process point of view: Coupled entities have changed together in the past and are likely to change together in the future. Change coupling information reveals potentially “misplaced” artifacts in a software system: To prevent a developer modifying a file in a system from forgetting to modify logically related files only because they are placed in other subsystems or packages, software artifacts that evolve together should be placed close (*e.g.*, in the same subsystem) to each other.

As we pointed out in Section 2.4, when surveying approaches dealing with change coupling analysis, such an analysis has two main benefits: (1) It is more lightweight than structural analysis and (2) it can reveal hidden dependencies that are not present in the code or in the documentation. However, our survey revealed also the main problem of previous techniques, *i.e.*, the lack of integration: Existing approaches work either at the architecture level or at the file (or even lower) level. Working at the architecture level provides high-level insights about the system’s structure, but low-level information about finer-grained entities is lost, and it is difficult to say which specific artifact is causing the coupling. Working at the file level makes one lose the global view of the system and it becomes difficult to establish which higher-level consequences the coupling of a specific file has.

We devised an approach that makes up for this dichotomy by integrating both levels of information employing an interactive visualization that we named the *Evolution Radar* [DLL06; DL06b; DLL09]. The Evolution Radar visualizes information both at a module-level (which modules are coupled with each other) and at a file-level (which files are responsible for the change couplings).

With the Evolution Radar visualization we tackle the following problems:

- Presenting large amounts of evolutionary information in a scalable way.
- Identifying outliers among change coupling relationships.
- Enabling developers and analysts to study and inspect these relationships and to guide them to the files that are responsible for the change couplings.

To validate our technique we applied it on ArgoUML and Azureus, two large and long-lived open source Java systems, and on CodeCity, a 3D visualization tool implemented in Smalltalk.

Structure of the chapter. In Section 4.1 we introduce our approach based on the *Evolution Radar*, a visualization technique that renders change coupling information and discuss its benefits and shortcomings. We illustrate the use of the radar for retrospective analysis in Section 4.2 and show how it can also be used for maintenance tasks in Section 4.3. We conclude by summarizing our contributions in Section 4.4.

4.1 The Evolution Radar

The Evolution Radar is a visualization technique to render file-level and module-level change coupling information in an integrated and interactive way. It is interactive, and allows the user to navigate and query the visualized information.

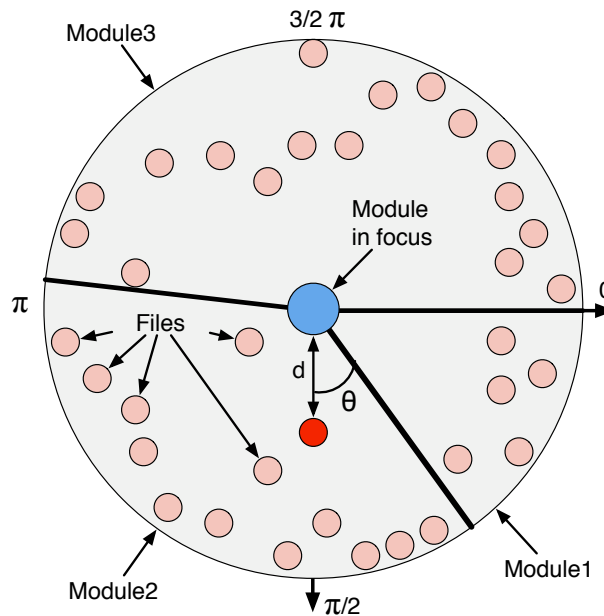


Figure 4.1. Principles of the Evolution Radar

The Evolution Radar shows the dependencies between a module in focus and all the other modules of a system. The module in focus is represented as a circle and placed in the center

of a circular surface (cf. Figure 4.1). All the other modules are visualized as sectors, whose size is proportional to the number of files contained in the corresponding module. The sectors are sorted according to this size metric, and placed in clockwise order. Within each module sector, files belonging to that module are represented as colored circles and positioned using polar coordinates where the angle and the distance to the center are computed according to the following rules:

- Distance d to the center is a linear function of the change coupling the file has with the module in focus, *i.e.*, the more they are coupled, the closer the circle representing the file is placed to the center of the circular surface. The exact definition of the distance is:

$$d = \frac{R}{cc_{max}} \times (cc_{max} - cc) \quad (4.1)$$

where R is the radius of the circular surface, cc_{max} the maximum value of the change coupling and cc the value of the change coupling.

- Angle θ . The files of each module are alphabetically sorted considering the entire directory path, and the circles representing them are uniformly distributed in the sectors with respect to the angle coordinates. Like this, files belonging to the same directory, or classes belonging to the same package in Java, are close to each other. Although this is not the only type of sorting possible, it was particularly useful in our experiments because it maintained the modules decomposition (in directories or packages) in the visualization. Other types of sorting might expose different insights into the system.

Algorithm 1 shows the pseudo code of the layout algorithm. The Evolution Radar can map arbitrary metrics on the color and the size of the circle figures representing files.

Algorithm 1: The Evolution Radar layout algorithm

```

; // Let M be the module in focus, R the radar's radius and CC(M,f) the
change coupling between a module M and a file f
S := {All files f | f ∉ M}
cc-max := maxf ∈ S CC(M, f)
theta := 0
angle-step :=  $\frac{2\pi}{|S|}$ 
forall m ∈ {All modules  $\tilde{m}$  |  $\tilde{m} \neq M$ }, sorted by size do
  draw module sector initial boundary at theta
  forall f ∈ m, sorted by path do
     $\theta(f) := \text{theta}$ 
     $r(f) := \frac{R}{cc\text{-max}} \times (cc\text{-max} - CC(M, f))$ 
    theta := theta + angle-step
    draw f in polar coordinates ( $\theta, r$ )
  end
  draw module sector final boundary at theta
end

```

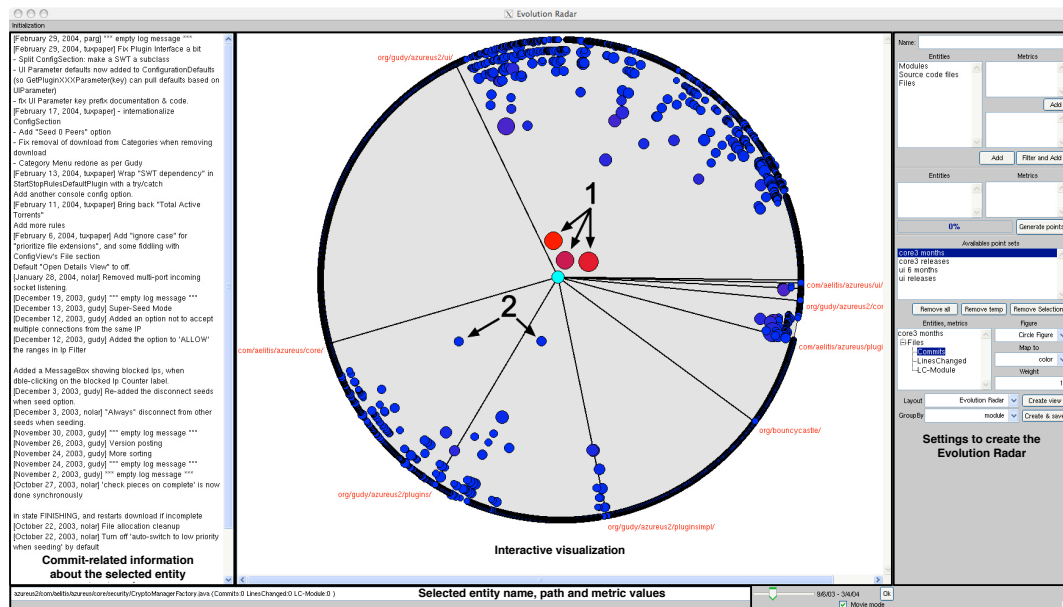


Figure 4.2. An example Evolution Radar applied on the core3 module of Azureus

4.1.1 Example

Figure 4.2 shows an example Evolution Radar visualizing the coupling between Azureus' core3 module (represented as the cyan circle in the center) and all the other modules (represented as the sectors). The size of the figures is proportional to the number of lines changed while their colors map the number of commits, using a heat-map from blue (lowest value) to red (highest value). We see that the ui module (on the top-right part of the radar) is the largest and most coupled module. The three files marked as 1 in the figure are the ones with the strongest coupling. They should be further analyzed to identify the most appropriate module to contain them: core3 or ui. Other modules do not have such a strong coupling, but we see the presence of some outliers, *i.e.*, files for which the coupling measure is much higher with respect to the context. The two files marked as 2, belonging to the plugins and pluginsimpl modules, are such outliers and should also be analyzed and moved in case they belong to the wrong module.

Figure 4.2 also shows the structure of the Evolution Radar tool. In the center the Evolution Radar has the interactive visualization which is set up using the panel on the right, selecting the entities (*e.g.*, source code files only or all files), the module in the center and the metrics. When an entity is selected in the visualization, it is possible to show the commit-related information about it (author, timestamp, comments, *etc.*) in the left panel. The tool allows the user to either consider all the commits or just the ones involved in the coupling. On the bottom part it displays information about the selected entity, such as its metric values.

4.1.2 Change Coupling Measure

In the Evolution Radar files are placed according to the change coupling they have with the module in focus. To compute the change coupling we use the following formula:

$$CC(M, f) = \max_{f_i \in M} CC(f_i, f) \quad (4.2)$$

where $CC(M, f)$ is the change coupling between the module in focus M and a given file f and $CC(f_i, f)$ is the coupling between the files f_i and f . It is also possible to use other group operators such as the average or the median. We use the maximum because it points us to the files with the strongest coupling, *i.e.*, the main causes for the module dependencies.

The value of the coupling between two files is equal to the number of transactions including both files. While SVN and Store mark co-changing files at commit time as belonging to the same transaction, in CVS transactions are not recorded and we need to reconstruct them, as described in Section 3.2.2.

4.1.3 Interaction

The Evolution Radar is implemented as an interactive visualization. It is possible to inspect all the visualized entities, *i.e.*, files and modules, to see commit-related information such as author, timestamp, comments, lines added and removed, *etc.* Moreover, it is possible to see the source code of selected files. Three important features to perform analyses with the Evolution Radar are (a) moving through time, (b) tracking and (c) spawning.

Moving through time

The change coupling measure is time-dependent. If we compute it considering the whole history of the system we can obtain misleading results.

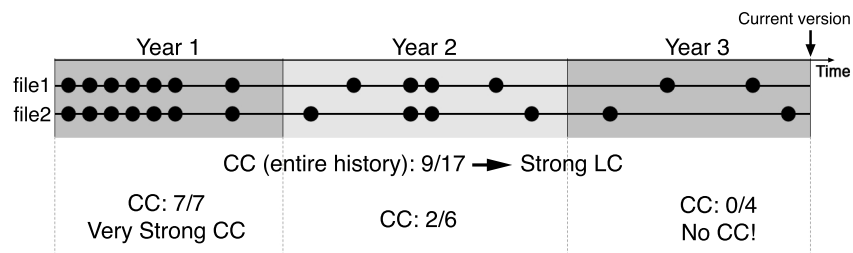


Figure 4.3. An example of misleading results when considering the entire history of artifacts to compute the change coupling value: file1 and file2 are not coupled during the last year.

Figure 4.3 shows an example of such a situation. It depicts the history, in terms of commits, of two files: file1 and file2. The time is on the horizontal axis from left to right and commits are represented as circles. If we compute the coupling measure according to the entire history we obtain 9 shared commits out of a total of 17, a high value because the files changed together more than fifty percent of the time. Although this result is correct, it is misleading since we could conclude that file1 and file2 are strongly coupled. Actually file1 and file2 were strongly coupled in the past but they are not coupled at all during the last year of the system.

Since we study change coupling information to detect architectural decay and design issues in the current version of a system, recent dependencies are more important than old ones. In other words, if two files were strongly coupled at the beginning of a system, but are not anymore

in recent times (perhaps due to a reengineering phase), we do not consider them as a potential problem.

For this reason the Evolution Radar is time-dependent, *i.e.*, it can be computed either considering the entire history of files or a given time window. When creating the radar, the user can divide the lifetime of the system into time intervals. For each interval a different radar is created, and the change coupling is computed with respect to the given time interval.

In each visualization all the files are displayed, even those inserted in the repository after the considered time interval or removed before. Like this, the theta coordinate of a file does not change in different radars and the position of the figures, with respect to the angle, is stable over time. This does not alter the semantic of the visualization, since these files are always at the boundary of the radar, their change coupling being zero.

The radius coordinate has the same scale in all the radars, *i.e.*, the same distance in different radars represents the same value of the coupling. This makes it possible to compare radars and to analyze the evolution of the coupling over time. In our tool implementation the user “moves through time” by using a slider, which causes the corresponding radar to be displayed. However, having several radars raises the issue of tracking the same entity across different visualizations, discussed next.

Tracking

Tracking allows the user to keep track of files over time. When a file is selected for tracking in a visualization related to a particular time interval, it is highlighted in all the radars (with respect to all the other time intervals) in which the file exists.

Figure 4.4 shows an example of tracking through four radars, related to four consecutive time intervals, from January 2004 to December 2005. The highlighting consists in using a yellow border for the tracked files and in showing a text label with the name of the file (indicated with arrows in Figure 4.4). It is thus possible to detect files with a strong change coupling in the last period of time and then analyze the coupling in the past, allowing us to distinguish between persistent and recent coupling.

Spawning

The spawning feature is aimed at inspecting the change coupling details. Outliers indicate that the corresponding files have a strong coupling with certain files of the module in focus, but we ignore which ones.

To uncover this dependency between files we spawn an auxiliary Evolution Radar as shown in Figure 4.5: The outliers are grouped to form a temporary module M_t represented by a circle figure. The module in focus (M) is expanded, *i.e.*, a circle figure is created for each file composing it. A new Evolution Radar is then created: The temporary module M_t is placed in the center of it. The files belonging to the module previously in focus (M) are placed around the center. The distance from the center is a linear function of the change coupling they have with the module in the center M_t . For the angle coordinate alphabetical sorting is used. Since all the files belong to the same module there is only one sector.

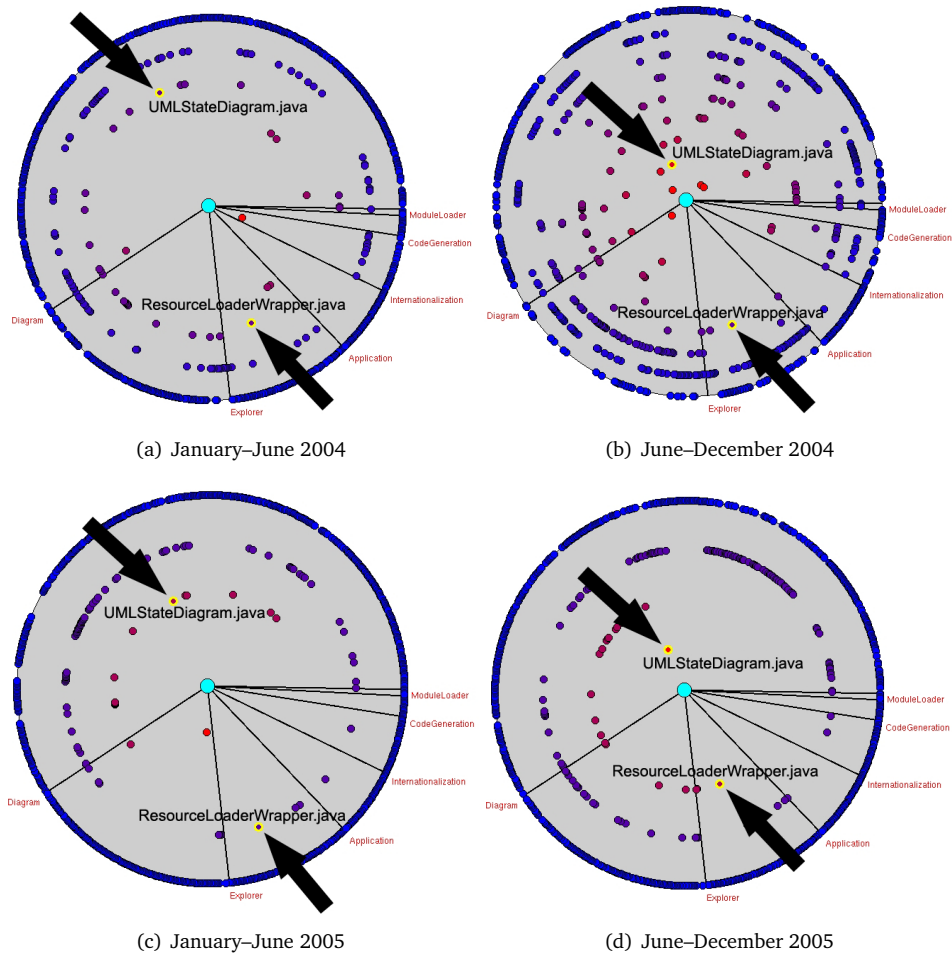


Figure 4.4. Evolution of the change coupling with the Model module of ArgoUML. The Evolution Radar keeps track of selected files along time (yellow border).

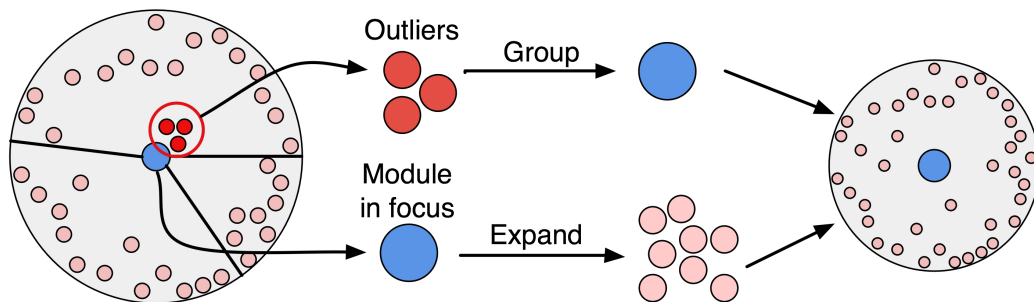


Figure 4.5. Spawning an auxiliary Evolution Radar

4.1.4 Discussion

One of the advantages of the Evolution Radar is that it does not visualize the coupling relationships as edges and therefore does not suffer from overplotting: The radar always remains intelligible, *i.e.*, it is easy to spot the heavily coupled modules (they are displayed as “spikes” pointing to the center). It is also easy to spot single files responsible for the coupling (they are placed close to the center).

The Evolution Radar is a general visualization technique, *i.e.*, it is applicable to any kind of entity. The only requirement is to define a grouping criterion and a distance metric. The radar can also be enriched by adding more structural information. A sector can be divided in sub-sectors to visualize sub-groups, *e.g.*, sub-modules, as proposed by Stasko and Zhang [SZ00].

The main drawback, common to many visualizations, is that it requires a trained eye to interpret the displayed information.

4.2 Using the Evolution Radar for Retrospective Analysis

We implemented two versions of the Evolution Radar: One is a stand alone tool to analyze systems developed using CVS or SVN, while the second version is integrated in an IDE environment to develop Smalltalk code. In this section, we apply the stand alone Evolution Radar tool to perform retrospective analysis on two open source software systems: ArgoUML and Azureus.¹ ArgoUML is a UML modeling tool written in Java, consisting of 1,565 classes and more than 200,000 lines of code. Azureus is a BitTorrent client written in Java, with 4,222 classes and more than 300,000 lines of code. In the following we present a summary of the analyses of the systems, giving several examples of our approach.

During the analyses, we use the term *strong coupling* (or strong dependency) between a file f and a group of files M when the following two conditions² hold:

1. $CC(M, f) \geq 10$ *i.e.*, there is at least one file f_M of M having a number of shared commits with f greater or equal than 10.
2. The number of commits shared by f and f_M divided by the total number of commits of f is greater than 0.35, *i.e.*, f is modified more than 35% of the times together with f_M .

4.2.1 Azureus

Since Azureus does not have any documentation about its architecture, *i.e.*, an explicit decomposition of the system in modules, we used the Java package structure to decompose the system. Throughout the analysis we use the following metric mappings: The size (area) of the figures representing files maps the number of lines changed and the color heat-map represents the number of commits. Both metrics are computed relative to the considered time interval.

Getting an overview

The first goal of our analysis is to obtain an initial understanding of the package histories, *i.e.*, when they were introduced or removed from the system.

¹Available respectively at <http://argouml.tigris.org> and <http://azureus.sourceforge.net>

²This definition is valid in the context of the analyzed software systems and its goal is to avoid mentioning the thresholds all the times.

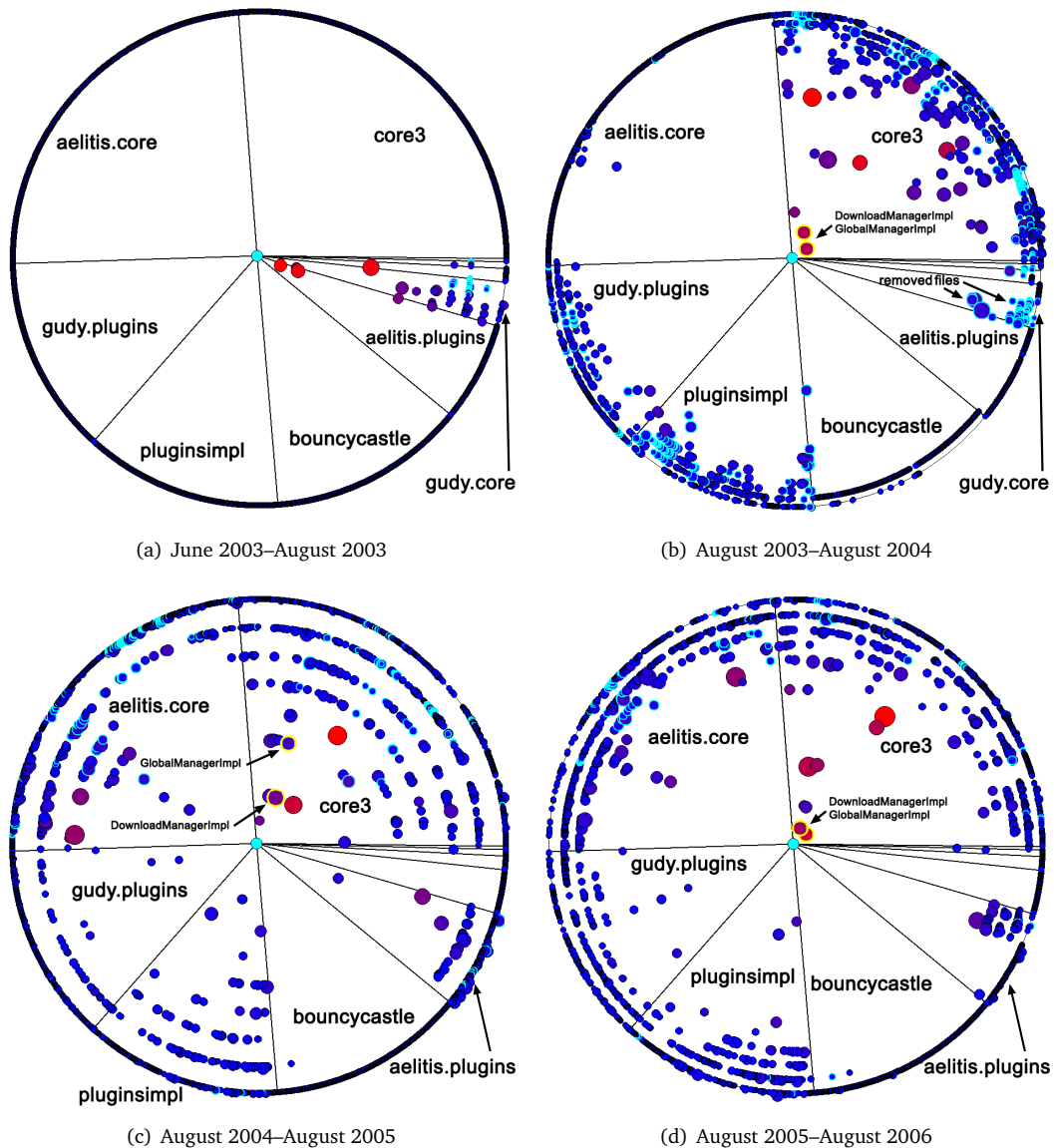


Figure 4.6. Evolution Radars of the `org.gudy.azureus2.ui` package of Azureus

We apply the Evolution Radar with the package `org.gudy.azureus2.ui` in the center, since it is one of the packages existing from the first to the last version of the system. Figure 4.6 shows the result with a time interval of one year.³ Figures with cyan borders represent files that were removed during the considered time interval. During the first three months (Figure 4.6(a)) `org.gudy.azureus2.ui` was coupled with `org.gudy.azureus2.core` (`gudy.core` from now on), while the other packages did not exist. In the following year (Figure 4.6(b)) `gudy.core` was removed,

³The first radar refers to a time interval of only three months because the time intervals are computed backward starting from the latest version of the system.

since all the figures have a cyan border and in the following radars there is no activity in this package. The core of the system became `org.gudy.azureus2.core3` coupled with `ui` (because of the figures close to the center) and `com.aelitis.azureus.core` (`aelitis.core` from now on) with very low activities (few figures). The plugins were also introduced with a clear separation from the `ui`, since the coupling was weak (no figures close to the center). From August 2004 to August 2005 (Figure 4.6(c)) the architecture decayed, as most of the packages were strongly coupled with the `ui`. Finally, during the last year (Figure 4.6(d)) the coupling decreased with all the packages, but in `core3` there were still files with a strong dependency with the `ui` (figures close to the center).

Detailing the dependency between `core3` and `ui`

In Figure 4.6 the two files indicated with the arrows and highlighted with the tracking feature (yellow border) are `GlobalManagerImpl.java` and `DownloadManagerImpl.java`. `GlobalManagerImpl` is a singleton responsible of keeping track of the `DownloadManager` objects that represent downloads in progress. They are the most coupled from 2003 to 2004 and from 2005 to 2006, and they have a strong dependency from 2004 to 2005. This strong and long-lived coupling indicates a design problem (a code smell), conforming to Martin's Common Closure Principle [Mar00]: "classes that change together belong together". The coupling indicates that the classes are misplaced or there are other design issues. We spawn two other radars having these files as the center to see which parts of the `ui` package they have dependencies with.

Figure 4.7 shows the radars corresponding to August 2004–2005 and August 2005–2006. The dependency between `GlobalManagerImpl` and `DownloadManagerImpl` and the `ui` package is mainly due to the class `MyTorrentsView`, a *God class* (defined by Riel as a class that tends to centralize the intelligence of the system [Rie96]).

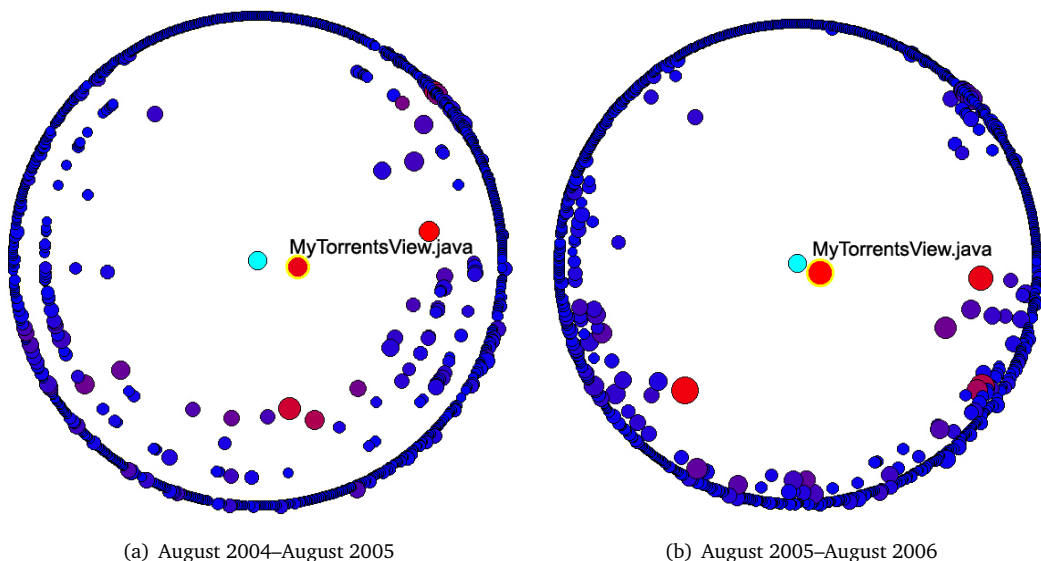


Figure 4.7. Evolution Radars for `GlobalManagerImpl` and `DownloadManagerImpl`

Such information is useful for (1) an analyst, because it points to design shortcomings and for (2) a developer, because when modifying `GlobalManagerImpl` or `DownloadManagerImpl` in `core3` she knows that `MyTorrentsView` is likely to need modifications as well.

Understanding the roles of modules

With this first series of radars we obtained an overall idea of the dependencies and evolution of the packages. We still do not have any idea of why there are two core packages and which roles they have. To obtain this information we select the files with the most intense activities (big and red) and we display their commit comments, as shown on the left of Figure 4.2. We find out that `core3` has more to do with managing files, download, upload and priorities while `aelitis.core` is more related to the network layer (distributed hash tables, TCP and UDP protocols).

In all the radars shown in Figure 4.6 the `org.bouncycastle` module has very low activity and coupling. A closer inspection reveals that the mentioned package was imported in the system at a given moment and never modified later. It implements an external cryptographic library⁴ used by the Azureus developers.

Analyzing the core package

As a second step, we want to understand the dependencies of `aelitis.core` with the rest of the system and we want to detect which are the files responsible for these dependencies. We create a radar for every six months of the system's history. We start the study from the most recent one, since we are interested in problems in the current version of the system. Using a relatively short time interval (six months) ensures that the coupling is due to recent changes and is not biased by commits far in the past.

Figure 4.8 shows the radar of `aelitis.core` from February to August 2006. Two packages are strongly coupled with `aelitis.core`: `core3` and `com.aelitis.azureus.plugins`. Comparing the radar with the static dependencies (e.g., invocations and inheritances) of `aelitis.core` extracted from the source code, we see that:

- The module with the strongest static dependency is `core3`. In fact, the strength of the dependency between packages, measured by the total number of dependencies between the contained classes, is one order of magnitude stronger for `core3` than for any other module. However, when we look at the radar we are surprised to see that the module with the strongest coupling is a different one: `com.aelitis.azureus.plugins`.
- Most of the static dependency with `core3` comes from the sub-package `util`, responsible for 45% of the dependencies between the two packages. However, the radar shows that `util` is less coupled than its sibling sub-packages `peer` and `download`. The same holds also in the past: In the radars referring to previous time intervals `util` is always less coupled than `download` and `peer`.

These observations demonstrate that change coupling is an implicit dependency. It complements static analysis, but it can only be inferred from the evolution of the system.

Figure 4.8 shows that the two classes with the strongest dependencies are `DHTPlugin` in the plugin package and `PEPeerControllImpl` in the `core3` package. Using the tracking feature of the Evolution Radar we found out that, in the previous year, these two classes were outliers, *i.e.*, they had a coupling much stronger than all the other classes in the package. To see the details of the dependency for `DHTPlugin` we spawn a new radar having the class as the center. The situation is different from the one shown in Figure 4.7 for `DownloadManagerImpl` and `GlobalManagerImpl`. For these two classes the coupling was mainly due to one single class (`MyTorrentsView`), while

⁴The Bouncy Castle library is available at <http://www.bouncycastle.org>

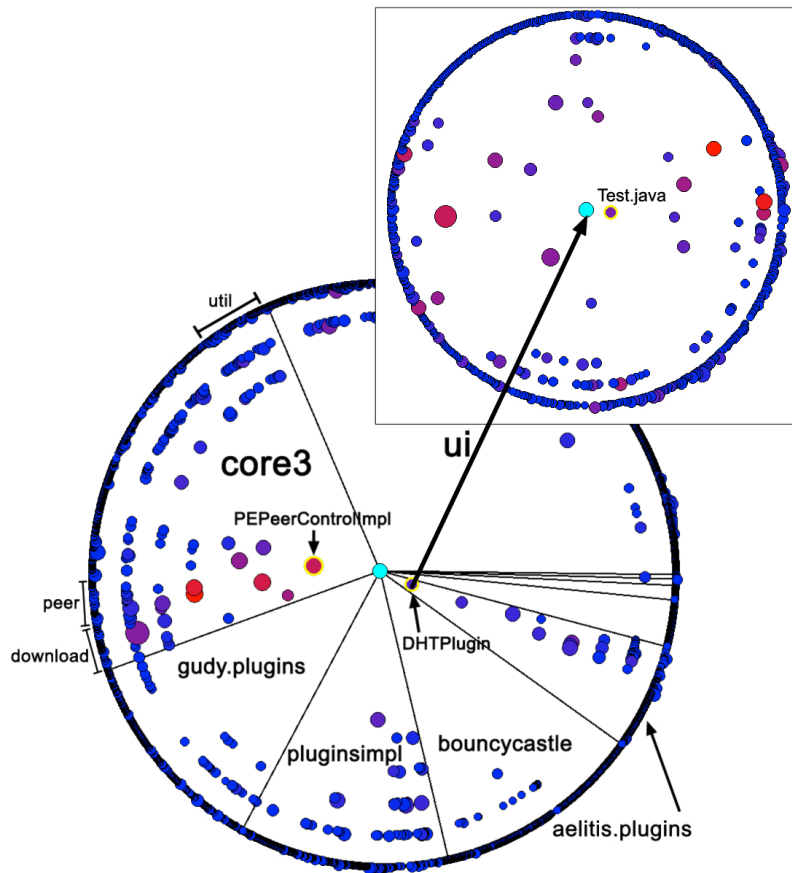


Figure 4.8. Evolution Radar for Azureus's aelitis.core package and details of the coupling between this package and the class DHTPlugin, February – August 2006

for DHTPlugin it is scattered among many files. This means that the file DHTPlugin.java was often changed together with many files in the aelitis.core package. This is a symptom of a misplaced file. Looking at the source code of the ten most coupled files, we discovered that all of them use DHTPlugin or its subclasses, meaning that core classes use plugin classes. Moreover, the class with the strongest coupling is a test case using DHTPlugin. Therefore, a modification in the plugin package can break a test in the core package.

This scenario repeats itself for the most coupled file in core3: The file PEPeerControlImpl.java is coupled with several files in aelitis.core. For the discussed reasons, DHTPlugin, PEPeerControlImpl, and their subclasses should be moved to aelitis.core.

Detecting renaming

The Evolution Radar can keep track of files, even if they are renamed or their code is moved to another place. This is possible because files are positioned according not only to their names, but also to their relationship with the center package. When a file f_{old} is renamed to f_{new} , the relationship f_{old} used to have with the package in the center, will hold for f_{new} (note that CVS

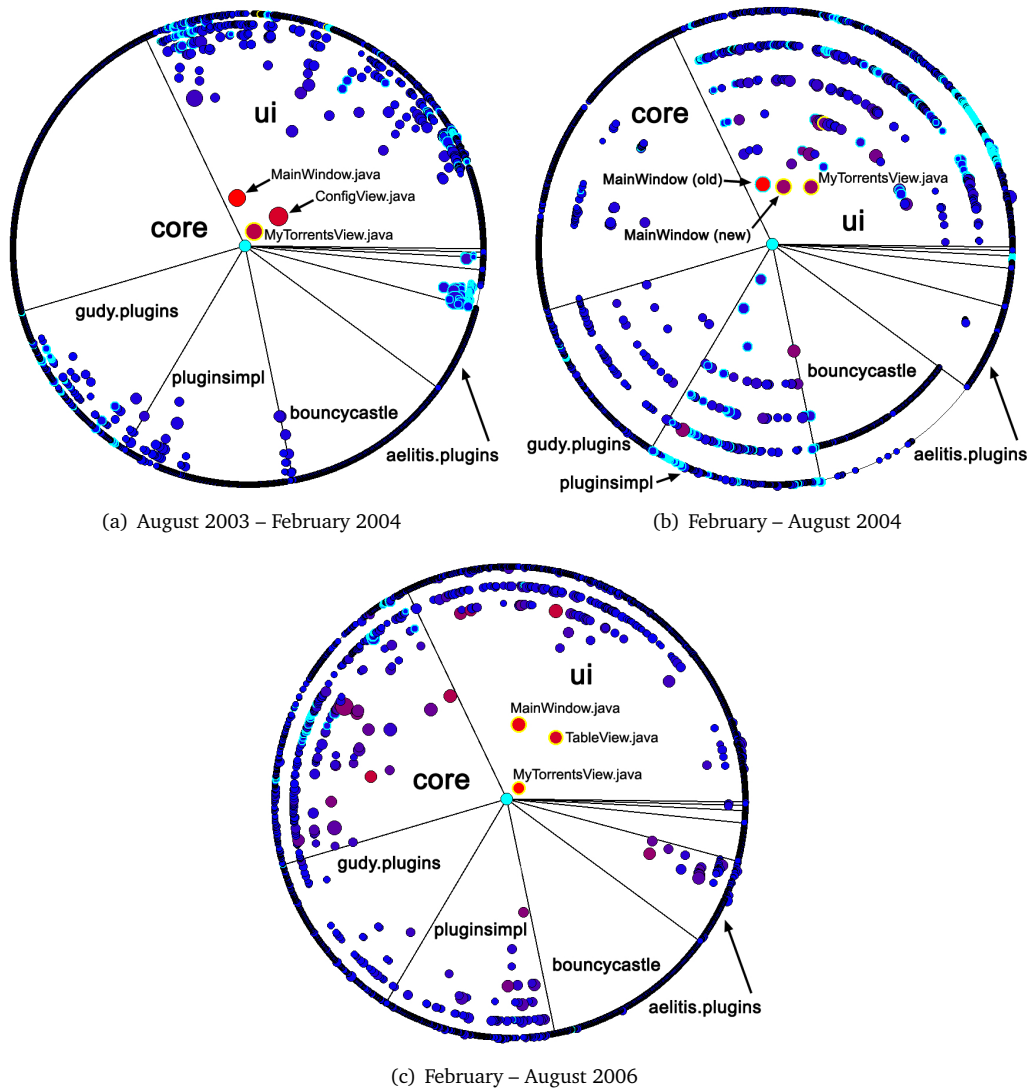


Figure 4.9. Evolution Radars for the core3 package of Azureus. They are helpful to detect the transition from the old `MainWindow.java` to the new one.

and SVN record it as a removal of f_{old} and an addition of f_{new}). Looking at similar couplings and removed files, we can detect such situations.

Figure 4.9 shows three Evolution Radars having the core3 package in the center. From February to August 2006 core3 had dependencies with the plugins packages and aelitis.core, and a strong coupling with the ui. In the ui there are three outliers: `MyTorrentsView.java` (already detected and discussed), `TableView.java` (the superclass of `MyTorrentView`, a God class as well) and `MainWindow.java`. Figure 4.9(a) shows the radar corresponding to August 2003 – February 2004. In the ui there are again three outliers: `MyTorrentsView.java`, `ConfigView.java` and `MainWindow.java`.

This `MainWindow.java` is a different file from the one in Figure 4.9(c) and, in fact, it is not highlighted by the tracking feature: The two `MainWindow` classes belong to different sub-packages. However, since they have the same type of coupling with `core3`, they can represent the same logical entity. Figure 4.9(b) shows the transition between the old and the new `MainWindow.java`. They both have a strong coupling with `core3` and the old one is removed, since it has a cyan border.

4.2.2 ArgoUML

We inferred ArgoUML's system decomposition into modules from its web site. We omitted the modules for which the documentation says "They are all insignificant enough not to be mentioned when listing dependencies" and focus our analysis on the three largest modules: `Model`, `Explorer` and `Diagram`. From the documentation we know that `Model` is the central module that all the others rely and depend on. `Explorer` and `Diagram` do not depend on each other.

We use the same analysis approach as for `Azureus`: We create a radar for every six months of the system's history. As metric we use the change coupling for both the position and the color of the figures. The size is proportional to the total number of lines modified in the considered time interval.

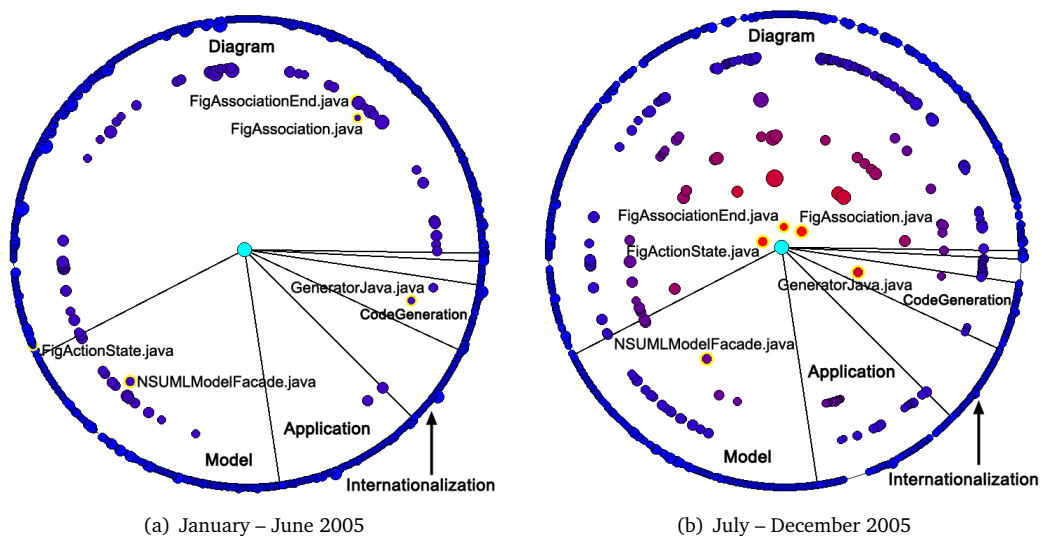


Figure 4.10. Evolution Radars applied to the Explorer module of ArgoUML

The Explorer module

Figure 4.10(b) shows the Evolution Radar for the last six months of history of the Explorer module. This module is much more coupled with `Diagram` than with `Model`, although the documentation states that the dependency is with `Model` and not with `Diagram`. The most coupled files in `Diagram` are `FigActionState.java`, `FigAssociationEnd.java` and `FigAssociation.java`. Using the tracking feature, we discover that the coupling with these files is recent: In the radar for the previous six

months (Figure 4.10(a)) they are not close to the center. This implies that the dependency is due to recent changes only. To inspect the change coupling details, we spawn an auxiliary radar: We group the three files and generate another radar centered on them, shown in Figure 4.11.

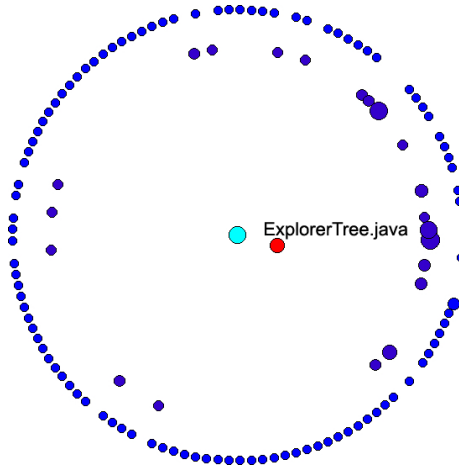


Figure 4.11. Details of the change coupling between ArgoUML's Explorer module and the classes FigActionState, FigAssociationEnd and FigAssociation

We now see that the dependency is mainly due to ExplorerTree.java. The high-level dependency between two modules is thus reduced to a dependency between four files. These four files represent a problem in the system, because modifying one of them may break the others, and since they belong to different modules, it is easy to forget this hidden dependency.

The visualization in Figure 4.10(b) shows that the file GeneratorJava.java is an outlier, since its coupling is much stronger with respect to all the other files in the same module (CodeGeneration). By spawning a group composed of GeneratorJava.java we obtain a visualization very similar to Figure 4.11, in which the main responsible for the dependency is again ExplorerTree.java. Reading the code reveals that the ExplorerTree class is responsible for managing mouse listeners and generating names for figures. This explains the dependencies with FigActionState, FigAssociationEnd and FigAssociation in the Diagram module, but does not explain the dependency with GeneratorJava. The past (see Figure 4.10(a) and Figure 4.12(a)) reveals that GeneratorJava.java is an outlier since January 2003. This long-lasting dependency indicates design problems. A further inspection is required for the ExplorerTree class in the Explorer module, since it is the main responsible for the coupling with the modules Diagram and CodeGeneration.

Detecting a move operation

The radars in Figure 4.10(b) and Figure 4.10(a) show that during 2005 the file NSUMLModelFacade.java had the strongest coupling—in the *Model* module—with Explorer (module in the center). Going six months back in time, from June to December 2004 (see Figure 4.12(a)), we see that the coupling with NSUMLModelFacade.java was weak, while there was a strong dependency with ModelFacade.java. This file was also heavily modified during that time interval, given its dimension with respect to the other figures (the area is proportional to the total number of lines modified). ModelFacade.java was also strongly coupled with the Diagram module (see

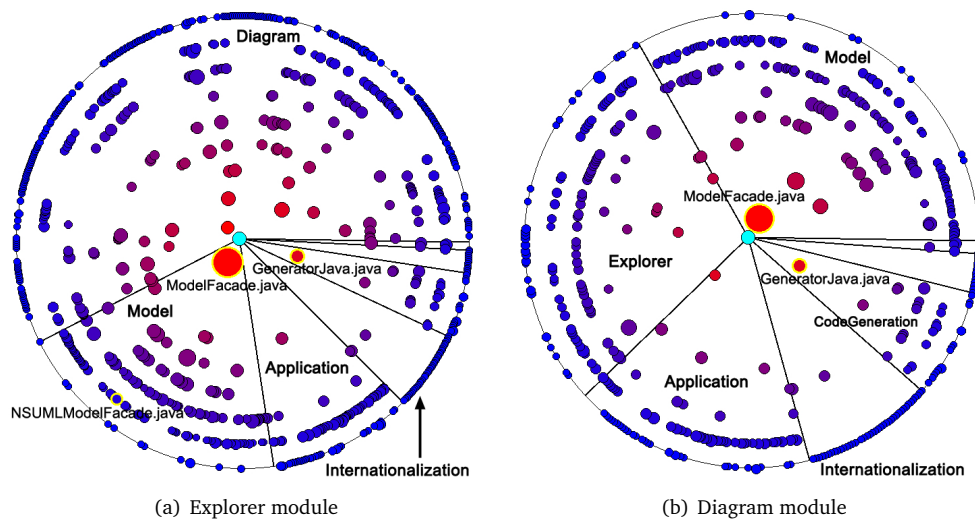


Figure 4.12. Evolution Radars of the Explorer and Diagram modules of ArgoUML from June to December 2004

Figure 4.12(b)). By looking at its source code we find that it was a God class [Rie96] with thousands of lines of codes, 444 public and 9 private methods, all static. The ModelFacade class is not present in the other radars (Figure 4.10(b) and Figure 4.10(a)) because it was removed from the system the 30th of January 2005. By looking at the source code of the most coupled file in these two radars, *i.e.*, NSUMLModelFacade.java, we discover that it is also a very large class with 317 public methods. Moreover, we find out that 292 of these methods have the same signature of methods in the ModelFacade class, where 75% of the code is duplicated. The only difference is that NSUMLModelFacade's methods are not static. Also, it contains only two attributes, while ModelFacade has 114 attributes. ModelFacade represented a problem in the system and thus was removed. Since most of the methods were copied to NSUMLModelFacade, the problem was just relocated.

This example shows how historical data reveals problems, that are difficult to detect looking at one version of the system only. Knowing the evolution of ModelFacade helped us to understand the role of NSUMLModelFacade in the current version of the system.

Identifying system phases

As a final scenario, we analyze the change coupling evolution of the Explorer module with all the others. From Figure 4.12(a), we see that from June to December 2004 the couplings were very strong. Then, from January 2005 to June 2005 (Figure 4.10(a)), they heavily decreased. This suggests that in the previous period the module was restructured and its quality was improved, since in the next time interval the coupling with the other modules was weak. The effort spent for the restructuring can be seen from the size of the figures, representing the total number of lines changed: In the radar relative to June – December 2004 (Figure 4.12(a)) the figures are bigger than in the radar relative to January – June 2005 (Figure 4.10(a)). At the end of the restructuring phase, the class ModelFacade was removed. From June to December 2005 (see Figure 4.10(b)) the coupling increased again. This can be related to a new restructuring phase.

4.2.3 Discussion

We applied the Evolution Radar on two open source software systems, showing that it helps in answering questions about the evolution of a system that are useful to developers, analysts, and project managers. The Evolution Radar offers a visual way to assess the files that might change in the future based on the prediction offered by change coupling. Due to the fine-grained level of the visualization, files can be inspected individually. For example in Azureus we discovered that a change in the class `GlobalManagerImpl` or `DownloadManagerImpl` in the `core3` package, is likely to require a change in the class `MyTorrentsView` in the `ui` package.

The Evolution Radar can be used to (1) understand the overall structure of the system in terms of module dependencies, (2) examine the structure of these dependencies at the file granularity level, and (3) get an insight of the impact of changes on a module over other modules. This knowledge will help them in the following activities:

- Locating design issues such as God classes or files with a strong and long-lived dependency with a module. Examples of design issues detected in ArgoUML include the classes `GeneratorJava` in `CodeGeneration` and `ExplorerTree` in `Explorer`. `GeneratorJava` has a persistent coupling with the `Explorer` module, while `ExplorerTree` is coupled with both the `CodeGeneration` and `Diagram` modules. In Azureus we detected God classes (`MyTorrentsView` and its superclass `TableView`) having strong dependencies with files belonging to different packages.
- Deciding whether certain files should be moved to other modules. In Azureus's case we discussed why `DHTPlugin` should be moved to the `aelitis.core` package.
- Understanding the evolution of the change coupling among modules. This activity can reveal phases in the history of the system, such as restructuring phases. In ArgoUML we identified different phases—with respect to the module `Explorer`—where two of them are likely to be restructuring phases. The evolution of the dependencies, together with the information about the removed files, is also helpful to see when modules were introduced or removed from the system. We used this information to obtain an overall idea of Azureus' structure in terms of packages.
- Detecting when artifacts have been renamed or moved. For example, we discovered that in Azureus the class `MainWindow` was moved between packages. In ArgoUML we found out that most of the code of the `ModelFacade` class was moved to `NSUMLModelFacade`.

4.3 Supporting Maintenance with the Radar

As a second application of the Evolution Radar, we integrated it in an IDE, to support maintenance activities. In this case, the radar visualization is part of the code browser, to allow a developer going directly from the visualization to the code and back. The IDE we enriched with the Evolution Radar is the System Browser [RBJ97] of the Cincom Smalltalk Visualworks distribution.⁵

In Cincom Smalltalk the code is organized in bundles: A bundle can contain packages and bundles, and packages contain classes. To render the Evolution Radar, we consider the system decomposition in packages, “flattening” the bundles hierarchy. We consider packages instead of

⁵Available at <http://www.cincomsmalltalk.com>

bundles because the latter cannot contain classes, but only packages. To compute the change coupling between two classes versioned with Store (the versioning system for Cincom Smalltalk), we consider all the versions of the classes corresponding to the considered time interval, and we count how many times they changed together, *i.e.*, how often they were committed in the same transaction. This number of co-changes is the change coupling between the two classes in the considered time interval.

4.3.1 Integration in the IDE

Figure 4.13 depicts the enriched System Browser IDE. In the top part there are four panels used to browse the code. They present respectively packages, classes, protocols⁶ and methods. In the bottom part there are multiple tabbed views that present details or allow the editing of the element that is selected on top. They include a code editor, a code analysis tool, a comment editor, *etc.* We extended the IDE with a new tabbed view for the Evolution Radar.

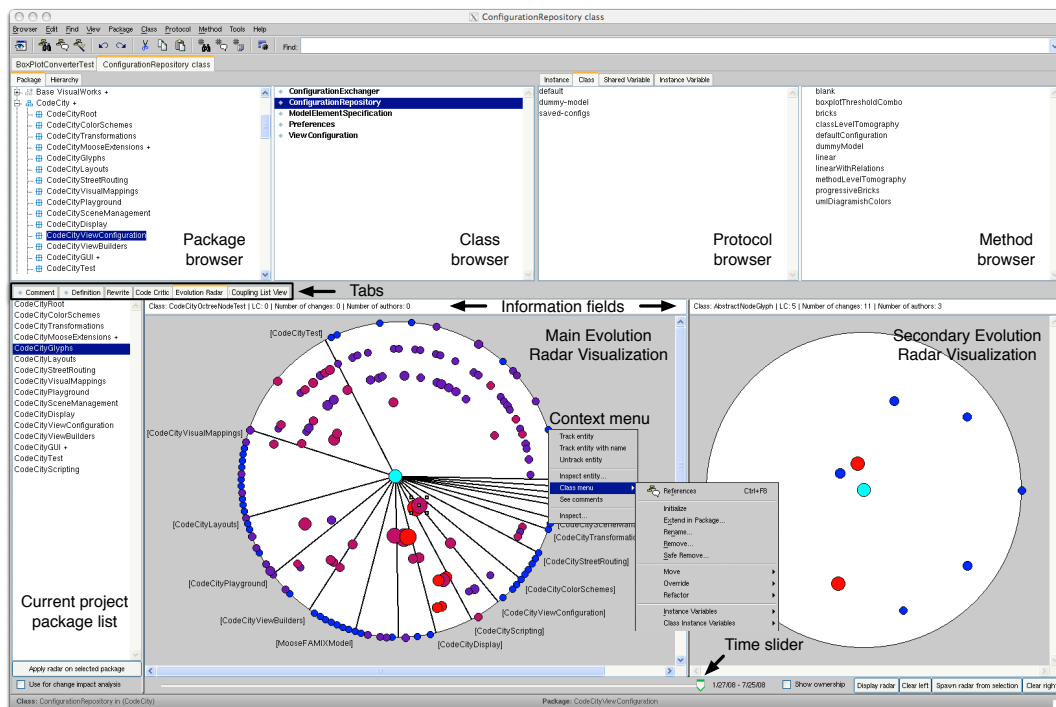


Figure 4.13. The Evolution Radar integrated in the System Browser IDE. It shows the change coupling between the CodeCityGlyphs module and the rest of CodeCity's modules.

The Evolution Radar view is composed of three panels (cf. Figure 4.13): The list of available packages in the current project, the main radar visualization and the secondary visualization in which a second radar can be spawned from a selection in the main one. The Evolution Radar panels include the information fields that show information about the entity under the mouse pointer and the time slider which supports the visual navigation through time.

⁶Protocols are a Smalltalk-specific way of grouping methods in a class.

The integrated version of the Evolution Radar adds the following interaction modes:

- When clicking on a figure, the browser in the top part displays the corresponding class or package, allowing the developer to immediately see and, in case, modify the source code. Moreover, through the context menu (see Figure 4.13), it is possible to directly move, modify and apply refactorings on the corresponding class.
- When selecting a figure or a group of figures, it is possible to: (1) Spawn a radar in the secondary view, (2) spawn a window with a code browser, (3) track the entity over time, and (4) read the commit comments corresponding to all the commits or just to the ones involved in the coupling.

To assess the usefulness of the visualization against the same coupling information presented in a list, we added another tabbed view to the IDE, called “Coupling List View”. In this view, instead of the main and secondary evolution radar visualizations, there is the list of classes coupled with the package selected in the package list (bottom left part of Figure 4.13), sorted according to the change coupling value.

This implementation of the Evolution Radar is designed to support Smalltalk developers in three types of activity: System restructuring, re-documentation and change impact estimation.

Restructuring

Having the Evolution Radar integrated into the IDE makes it easy to inspect the classes that are coupled and—if they are in the wrong place—to restructure the system, *i.e.*, move the classes to the appropriate package. This operation can be performed directly from the context menu in the Evolution Radar visualization (see Figure 4.13).

Re-documentation

The developer uses the radar to analyze the coupling of a package with the rest of the system and, by looking at the commit comments and the source code, he can discover the reasons for the coupling. Then, the developer can annotate this information directly on the involved classes and/or packages by writing a comment into the “Comment” tab (see the list of tabs in Figure 4.13). These comments are part of the system code and get versioned as every other entity.

Change impact estimation

When two classes are logically coupled, they are likely to change together in the future [SW08]. This information can be useful to a developer who is about to make a change to a class in the system because it can support him in estimating the impact of the change.

The developer can select the class (or classes) he needs to modify and see which are the classes that are coupled with it. If there is no coupling (no figures close to the center), the developer can go on with the modification. If the class is coupled with few other classes, he can obtain more insight by looking at their source code and reading the comments written in the re-documentation phase. If the class is coupled with many other classes in the system, the developer has to find out whether these couplings are due to large commits or whether the class is affected by design issues. To do so, he can exploit the information gained in the re-documentation phase or he can look at the commit comments accessible directly in the radar.

The developer can access the “change impact” functionality using the radio button in the bottom left corner of the tool (see Figure 4.13). Whenever he selects a package, a class or multiple classes in the code browser (in the upper part), an Evolution Radar having the selection in the center is generated and rendered on the fly.

4.3.2 Experimental Evaluation

To experiment the IDE version of the Evolution Radar, we asked a developer to use it, and report on his experience. The developer is Richard Wettel, a PhD student in the University of Lugano. The system to which he applied the Evolution Radar is called CodeCity,⁷ a software analysis tool that visualizes software systems as interactive 3D cities. At the time of the experiment (August 2008), CodeCity was composed of 478 classes and had about 15,000 lines of code. It was developed since April 2006 mostly by a single developer, and occasionally by two other developers, in the context of pair-programming sessions. The following, in italic, is a slightly adapted extract of his experience report, organized according to the performed maintenance activities.

Re-documentation phase

We started by analyzing coupled modules (i.e., packages in Smalltalk), looking for coupling to the core packages, in our case the ones dealing with the glyphs, layouts, and mappings. Figure 4.13 shows the coupling to the glyphs module, represented by the circle in the middle of the left visualization panel, which seems to be distributed in five levels (i.e., five concentric layers besides the external one which shows the uncoupled classes). Whenever we needed to see which are the entities inside the module in the middle coupled with a particular class, we spawned a radar from the selection (the right visualization panel in Figure 4.13) and observed these in isolation. Selecting a class circle triggers its selection also in the code editor. Integrating the information about change coupling with the code and with versioning logs allowed us to reason about the system’s evolution.

Overall, we were able to determine the cause of the unexpected change couplings with the help of the context menu option, which allows looking at the log entries of the system versions which produced the coupling. The causes were either larger commits incorporating several unrelated changes to the system or in one case the system was massively restructured. While we did not find any coupling that needed refactoring, all of them were accurately detected. We finally added all information we found to the comments of each analyzed class.

Change impact estimation phase

The second task we performed was assessing the impact of change of the AbstractLayout class (the root of the layout hierarchy), since we needed to reengineer the way the layouts communicate with the glyphs and mappings. To do so, we enabled the “change impact” functionality through the radio button, and then selected the AbstractLayout class in the code browser, which automatically generated the corresponding radars. In Figure 4.14, we see the evolution of the change coupling to the AbstractLayout class, which provides an accurate representation of the way the entire system evolved. In a first time period (top left) classes are being changed (i.e., large circles) in many packages being developed in parallel and logically coupled among each other. The following period is very unfocused, but with smaller changes (i.e., small fixes). The third period is one of coupling with classes

⁷CodeCity is available at <http://codecity.inf.usi.ch>

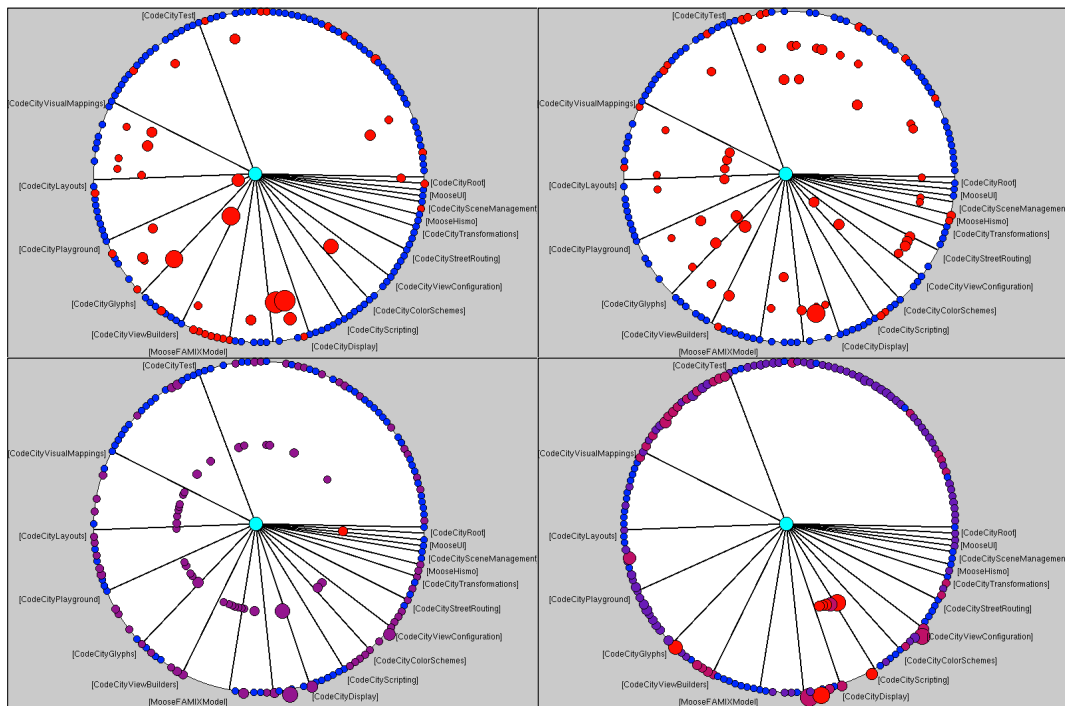


Figure 4.14. The change coupling evolution of the class AbstractGlyph

from almost all the packages, while the last period shows coupling only to few new classes all defined in the CodeCityScripting package. Reading the comments written in the re-documentation phase, we knew that these classes of CodeCityScripting—which implement a basic scripting language for 3D visualizations—are coupled with many core classes (including AbstractLayout) and the coupling is justified: The core classes changed together with the scripting classes in order to make the system compatible with the new scripting language. Since the coupling was justified, we knew that we could proceed with reengineering the layouts, but being careful to comply with the new scripting language, not to break the dependency with the CodeCityScripting package.

Comments

We were asked to try the list view and compare it with the radar view. In our experience, the list view was useful to easily spot the most coupled class with a given package. However, with the list we had difficulties in understanding the coupling at the system level and to see how it was distributed among packages and classes. We found it useful to be able to navigate in time using the radar, while keeping track of certain classes. With the list view, we did not succeed in navigating in time, because it was difficult to keep track and compare the values of the change coupling for a class over different time intervals.

4.3.3 Lessons Learned

Although we do not consider this experiment a full-fledged user study, it points to the fact that the Evolution Radar can be successfully used to support maintenance activities. The visualization itself, without the interactive features (moving through time, spawning, tracking, inspecting), is not sufficient to perform the tasks, as the developer continuously used them during the maintenance activities. In particular, the possibility to read the commit comments seems crucial.

Overall, while the first implementation of the radar as a stand alone tool is helpful to perform retrospective analysis, only through the integration in a development environment the radar exploits its full potential, as an IDE enhancement.

4.4 Summary

We presented the Evolution Radar, an approach to integrate and visualize module-level and file-level change coupling information, extracted from our evolutionary meta-model Mevo. The Evolution Radar is useful to answer questions about the evolution of the system, the impact of changes at different levels of abstraction and the need for system restructuring. The main benefits of the technique are:

1. *Integration.* The Evolution Radar shows change coupling information at different levels of abstraction, *i.e.*, files and modules, in a single visualization. This makes it possible to understand the dependency among modules and—using the spawning feature of our tool—reduce it to a small set of strongly coupled files responsible for the dependency.
2. *Control of time.* Considering the history of change coupling is helpful to uncover hidden dependencies between software artifacts. However, summarizing the information about the entire history in a single visualization may lead to imprecise results. Two artifacts that were strongly coupled in the past but not recently may appear as coupled. The Evolution Radar solves this problem by dividing the system lifetime in settable time intervals and by rendering one radar per interval. A slider is used to “move through time”. A tracking feature is provided to keep track of the same files in different visualizations.

We illustrated our approach on two large and long-lived open source software systems: ArgoUML and Azureus. We provided example scenarios of how to use the Evolution Radar to understand module dependencies and impact of changes at both file and module levels. We found design issues such as God classes, misplaced files and module dependencies not mentioned in the documentation. We also reduced these dependencies to coupling between small sets of files. These files should be considered for reengineering to decrease the coupling at the module level. The control of time allowed us to understand the overall evolution of the systems (when modules were introduced/removed) and to identify phases in their histories.

We showed how the tight integration of the Evolution Radar with an IDE can support maintenance activities like restructuring, re-documentation and change impact estimation. We described how this support works, by presenting an experience report of a developer using the Evolution Radar inside the Smalltalk System Browser IDE.

The focus of this chapter was the evolution of software artifacts. The next chapter focuses on the evolution of software defects.

Chapter 5

Analyzing the Evolution of Software Defects

Bug tracking systems play an important role in software development [MFH02; RdMF02]. They are used by developers, quality assurance people, testers, and end users to provide feedback on the system. This feedback can be reported as an incorrect or anomalous situation or as a request for enhancements. Bugs, often considered as an unwanted “side dish” of the evolution phenomenon, in fact represent a valuable source of information that can lead to interesting insights about a system, that would be hard or impossible to obtain relying exclusively on structural information.

When presenting our Mevo meta-model (cf. Section 3.1.2), we discussed the importance of modeling bugs as first class entities that can change and evolve over time. We showed, using Mozilla as an example, that bugs indeed live long (more than 50% of Mozilla bugs lived more than one year, cf. Figure 3.3), and thus in Mevo we model their histories, *i.e.*, the sequence of states that they traverse. In this chapter we present a visual approach which exploits such historical information about software defects, as captured in our meta-model. As discussed when surveying approaches for bug analysis (cf. Section 2.5.1) and software evolution visualization (cf. Section 2.3.2), researchers proposed a number of techniques that use bug data, but they focus on the structural evolution of systems and do not consider bugs as evolving entities.

Our approach consists in two interactive visualizations that support the understanding of bugs at two different levels of granularity:

1. *System Radiography*. This view renders bug information at the system level and provides indications about which parts of the system are affected by what kind of bugs at which point in time. It is a high-level indicator of the system health and serves as a basis for reverse engineering activities.
2. *Bug Watch*. This visualization provides information about a specific bug and is helpful to understand the various phases that it traversed. The view supports the characterization of bugs and the identification of the most critical ones.

The visualizations are complementary to established structural visualizations and other reverse engineering techniques.

Structure of the chapter. We present our technique for visualizing software defects in Section 5.1, discussing also the constraints of dealing with a large bug database. In Section 5.2 we apply our approach on Mozilla, showing how to perform bug analysis in the large and in the small with our tool. In Section 5.3 we discuss the benefits and limitations of our approach, and we conclude the chapter in Section 5.4.

5.1 Visualizing a Bug Database

Table 5.1 reports some statistics about the bug databases of three open-source software projects: ArgoUML, Eclipse and Mozilla.

Table 5.1. Statistics about the bug databases of ArgoUML, Eclipse and Mozilla

	ArgoUML Feb 2000–Apr 2009		Eclipse Oct 2001–Jul 2009		Mozilla Sep 1998–April 2003		All	
Number of bugs	4,280		120,531		255,302		380,025	
Activities per bug	7.1		7.8		10.6		9.7	
Bug lifetime								
< 1 day	995	23.7%	30,974	25.7%	28,236	11.1%	60,205	15.8%
1 day – 1 week	673	16.1%	25,787	21.4%	15,420	6.0%	41,880	11.0%
1 week – 1 month	641	15.2%	21,008	17.4%	19,250	7.5%	40,899	10.8%
1 – 6 months	734	17.5%	20,747	17.2%	46,644	18.3%	68,125	17.9%
6 months – 1 year	376	9.0%	7,004	5.8%	26,909	10.5%	34,289	9.0%
1 – 2 years	372	8.9%	6,517	5.4%	37,453	14.7%	44,342	11.7%
> 2 years	401	9.6%	8,494	7.1%	81,390	31.9%	90,285	23.8%
> 1 month	1,883	44.9%	42,762	35.5%	192,396	75.4%	237,041	62.4%
> 6 months	1,149	27.4%	22,015	18.3%	145,752	57.1%	168,916	44.5%

The table shows that the size of bug databases can be large, leading to a number of constraints for visualizing them. In particular, three properties of bug databases have to be considered:

1. *Bug number*: A bug database can contain more than 100,000 bug reports (Eclipse and Mozilla), therefore the visualizations have to scale.
2. *Bug liveliness*: In all the considered software projects more than 18% of bugs last more than 6 months (time between the reporting and the last registered activity) with an average number of activities ranging from 7.1 (ArgoUML) to 10.6 (Mozilla). Thus, the visualizations should not only display individual bugs, but also complementary information such as their activities and status histories.
3. *Bug importance*: Expressed with severity and priority, bugs have different impact and importance, and the visualization must help to convey this distinction.

5.1.1 The System Radiography View

The goal of the *System Radiography* view is to support the analysis of a bug database as a whole. We want to study how the open bugs (not fixed yet) are distributed in the system and over time.

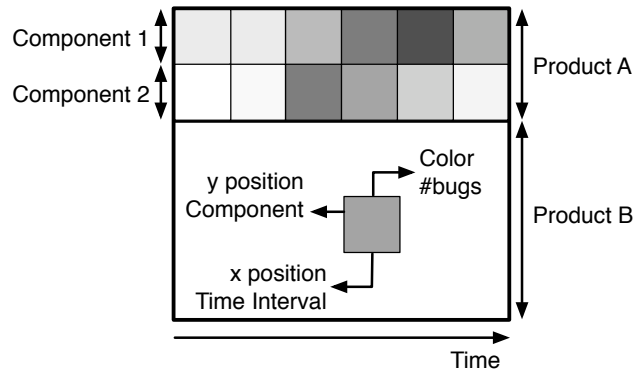


Figure 5.1. The principles of the System Radiography visualization

Visualization Principles

Figure 5.1 shows the principles of the System Radiography view. The evolving system is displayed using a matrix-based representation. Each row of the matrix represents a system component, and each group of rows, *i.e.*, components, represents a system product. We obtain this hierarchical decomposition of the system in Product::Component from the bug database itself, since each bug affects a particular component belonging to a particular product. The columns represent time from left to right. Each column corresponds to a parametrizable interval of time.

The y position of each cell represents a Product::Component pair, while the x position represents an interval of time. The color of the cells maps the number of bugs affecting the y component during the x time interval, where x and y are the position of the cell. We use a gray scale: The darker the color, the larger the number of bugs.

A bug, at any point in time (of its life), is characterized by a status. Therefore, it does not make sense to count the number of bugs during a time interval without considering their statuses. For this reason, the color of the cells represents the number of bugs with a given status (or set of statuses) during the considered time interval. This allows us to see the distribution of the bugs in the system and over time with respect to their status: For example “open” bugs (bugs with *new*, *assigned* or *reopened* status), “solved” bugs (*resolved*, *verified* or *closed*), or only *new* or *reopened* bugs, *etc.* In our tool implementation other filters can be used: For example, we can use the severity filter to count the *blocker* and *critical* bugs only.

Once the matrix is created, we apply a sorting algorithm to its rows before displaying it. The goal is to sort, for each product, its components according to the similarity of their histories, *i.e.*, the number of bugs for every time interval. Given a product p and two components c_1, c_2 , corresponding to the matrix rows r_1, r_2 of size n (number of columns), the similarity between the components is defined as the Euclidian distance of the points r_1, r_2 in a n -dimensional space, as defined by Equation 5.1.

$$d(r_1, r_2) = \sqrt{\sum_{i=1}^n (r_1(i) - r_2(i))^2} \quad (5.1)$$

The value of $r_j(i)$ is the number of bugs with a given status (and after the filtering) of the j -th component within the i -th time interval. After sorting, we obtain a matrix in which the products

are alphabetically sorted and the components within each product are sorted according to the defined similarity. This sorting allows us to detect groups of components that were affected by a large number of bugs in the same time window.

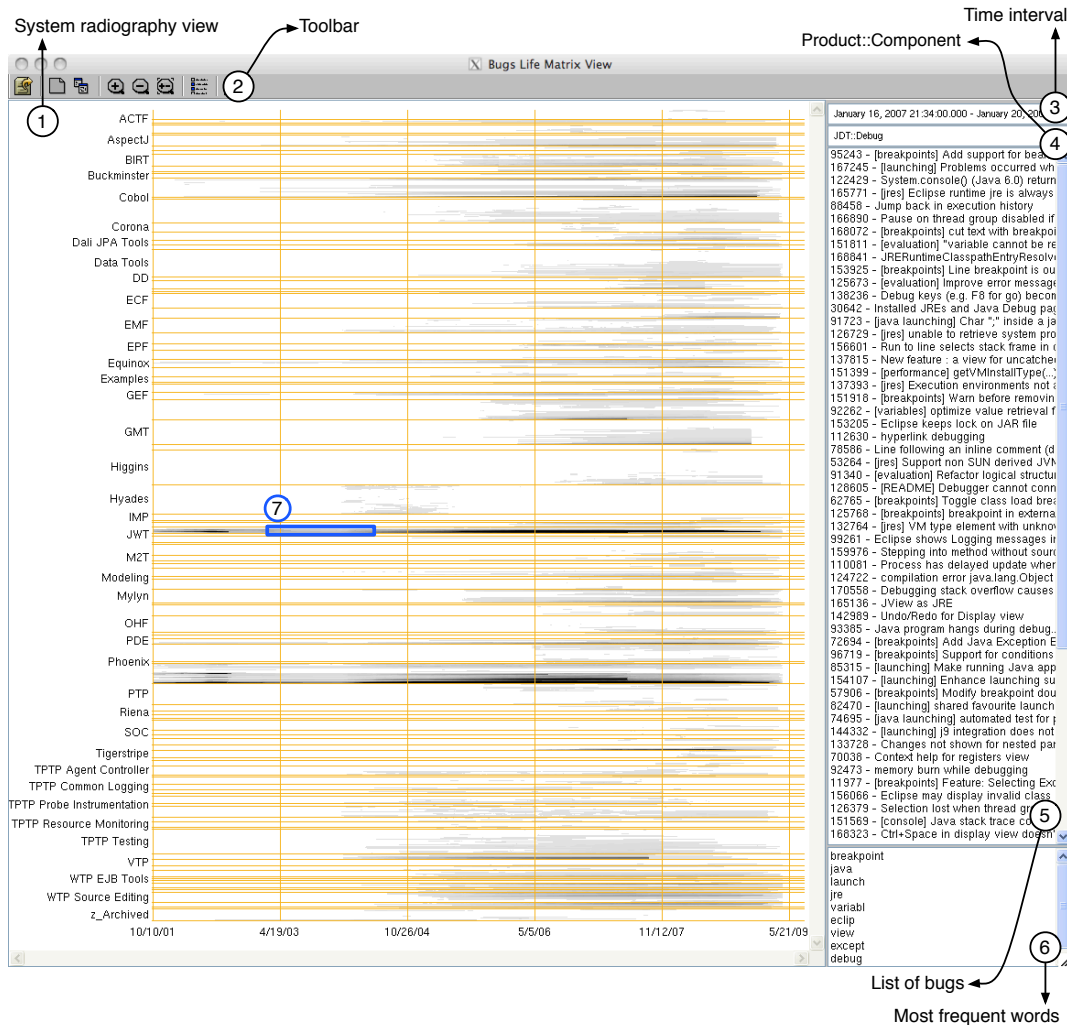


Figure 5.2. A System Radiography of Eclipse from October 2001 to July 2009. Only bugs with new, assigned or reopened statuses are considered.

Figure 5.2 shows our tool visualizing a System Radiography of Eclipse from October 2001 to July 2009, where open bugs only are considered. The time interval used is five days, meaning that each column of the matrix represents five days of time. The main window of the tool is divided in a visualization part on the left (marked as “1”), containing the actual System Radiography, and an information part on the right. This part displays the information concerning the matrix cell under the pointer: The time interval (3), the product::component pair (4), the list of bugs affecting that component during that time interval¹ (5) and the most frequent terms

¹Only the bugs with the considered statuses and severities are listed.

extracted from all bug activities and long descriptions (6). The tool provides also a toolbar (marked as “2” in Figure 5.2) to perform a number of actions such as importing the data from Churrasco, creating views, zooming and configuring visualization parameters.

The same bug can be counted in different cells of the same row. For example, if the bug 344 was in the *New* status for nine days, and the time interval is three days, then the bug is counted in three cells, as shown in Figure 5.3. On the other hand, since the product and component fields of a bug can have just one value, the same bug cannot be counted in different cells of the same column.

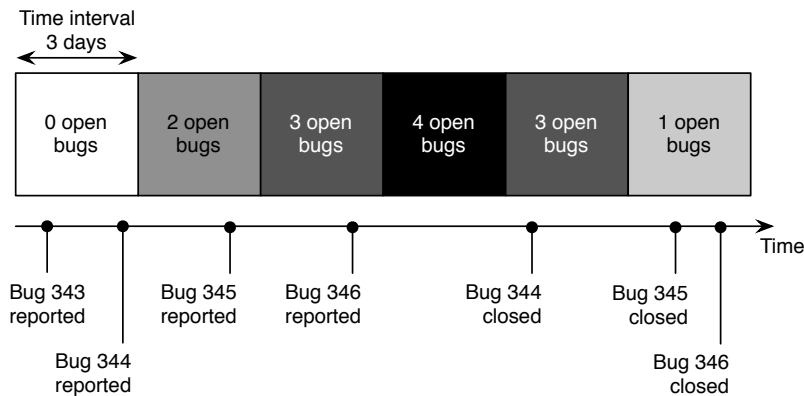


Figure 5.3. Counting open bugs over time in a row of the System Radiography view

5.1.2 The Bug Watch View

With the System Radiography view we obtain a big picture of the system from the bug perspective and we can detect the critical components. The purpose of the *Bug Watch* view is to ease the analysis of single bugs. Our goals are to characterize the bugs affecting a given set of components during a given time interval and to detect the most critical bugs. Our underlying assumption is that the criticality of a bug does not depend only on its severity and priority, but also on its life cycle. For example, a bug reopened several times indicates a problem which is hard to solve, as several attempts to fix the bug failed.

For this type of analysis we need a visualization which fulfills the following requirements:

- *Considering time.* The visualization has to be time-based in order to show the life cycle history of a bug.
- *Considering severity and priority.* These bug properties are important to detect critical bugs.

Visualization Principles

Figure 5.4 shows a *Bug Watch* visualization of a Mozilla bug, rendered in our Bug’s Life tool. The visualization technique uses a watch metaphor to represent time: The initial timestamp is mapped to 00:00 on the watch, the final timestamp is mapped to 11:59. The figure is composed of three layers:

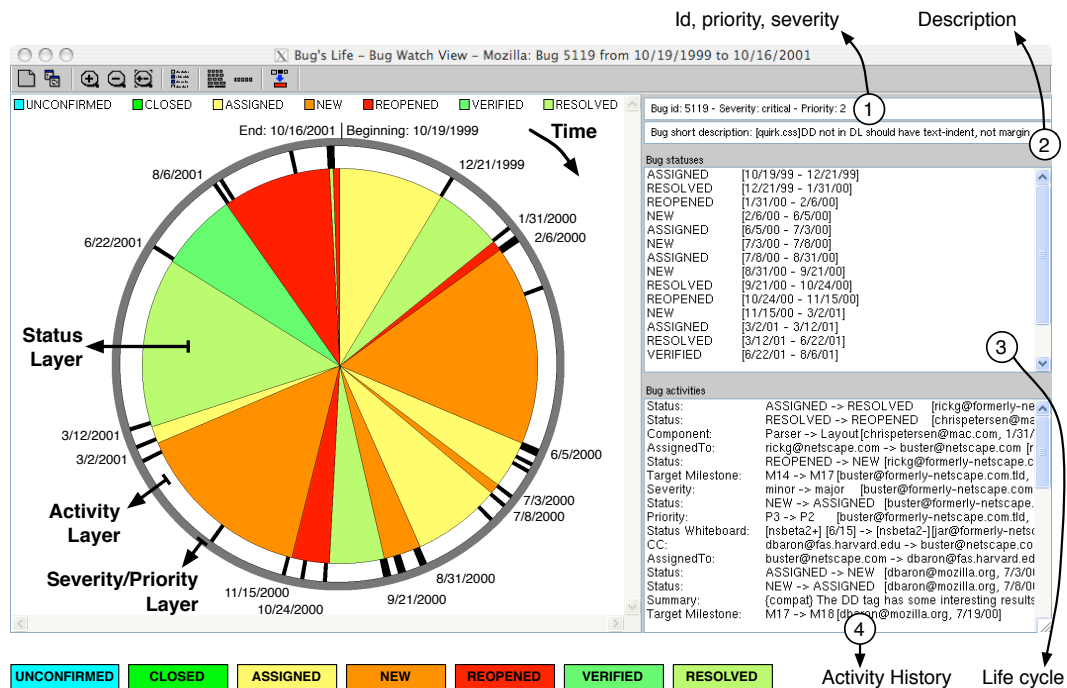


Figure 5.4. A Bug Watch Figure visualizing Bug 5119 of Mozilla between Oct 19, 1999 and Oct 16, 2001

1. The *Status layer* represents the bug life cycle. We visualize each status the bug passed through as a sector, using the following color schema (also used in Figure 3.6): Green colors represent statuses in which the bug is considered fixed (*i.e.*, *resolved*, *verified* or *closed*), while red colors represent statuses in which the bug has to be fixed (*i.e.*, *new*, *assigned* or *reopened*). The position and the size of each sector map respectively the initial timestamp and the duration of the corresponding status.
2. The *Activity layer* visualizes modifications of any bug property. We represent each activity as a black bar and we position it according to when the modification happened. Since an activity is an event, its visual representation has a fixed size. Wider bars denote several activities in the same time interval.
3. The *Severity/Priority layer* depicts information about the severity and the priority of a bug. Dark colors denote high priority and *blocker* or *critical* severity, while bright colors indicate low priority and *minor*, *trivial* or *enhancement* severity.

We can map different bug metrics on the radius of a Bug Watch. We usually choose the number of statuses in the considered time interval, which also corresponds to the number of sectors. This, besides easing the detection of bugs with an intense life cycle, also has the benefit of making the statuses more readable, as bugs with many statuses are represented with larger figures. Figure 5.5 shows an example of this: Statuses are always readable, as long as they do not last for a very short time.

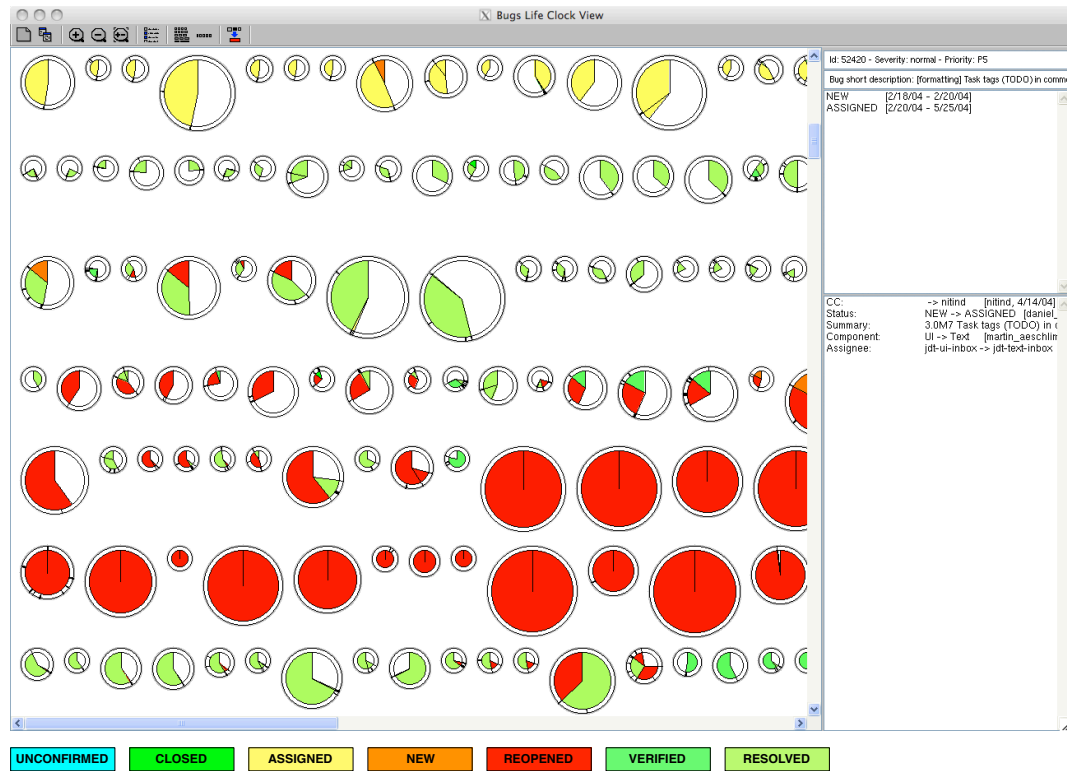


Figure 5.5. A Bug Watch visualization of the bugs affecting the JDT product of Eclipse. The reference time interval is 20/2/2003 – 5/25/2004. We represent each bug as a Bug Watch Figure.

One problem of the Bug Watch visualization is that it is not possible to distinguish between different activities, since they are all visualized in the same way. We opted against the use of different colors for different activities because this would make the figure too complex to interpret. A second problem arises when there are statuses which last for short periods of time. They are represented as narrow sectors, which are difficult to distinguish, especially in zoomed out views as for example in Figure 5.6. To overcome these issues, we provided our tool with four panels showing complementary information about the bug in focus (cf. Figure 5.4): (1) The id, priority and severity of the bug, (2) the description of the problem, (3) the life cycle, *i.e.*, the sequence of statuses the bug traversed and (4) the activity history, *i.e.*, the list of all activities performed on the bug.

Considering the high number of bugs, the question is which bugs to visualize using the Bug Watch view. The starting point is usually the previously presented System Radiography visualization: We select an area in the System Radiography that our tool converts in a set of bugs and in a time window. To create the set of bugs, we consider all the matrix cells covered by the rectangular selection, and we add the corresponding bugs. The selection—depending on its height—can cover one or several system components. The time window corresponds to the first (beginning) and last (end) matrix columns covered by the selection. We then visualize the selected bugs by means of Bug Watch figures, all with the same reference time window.

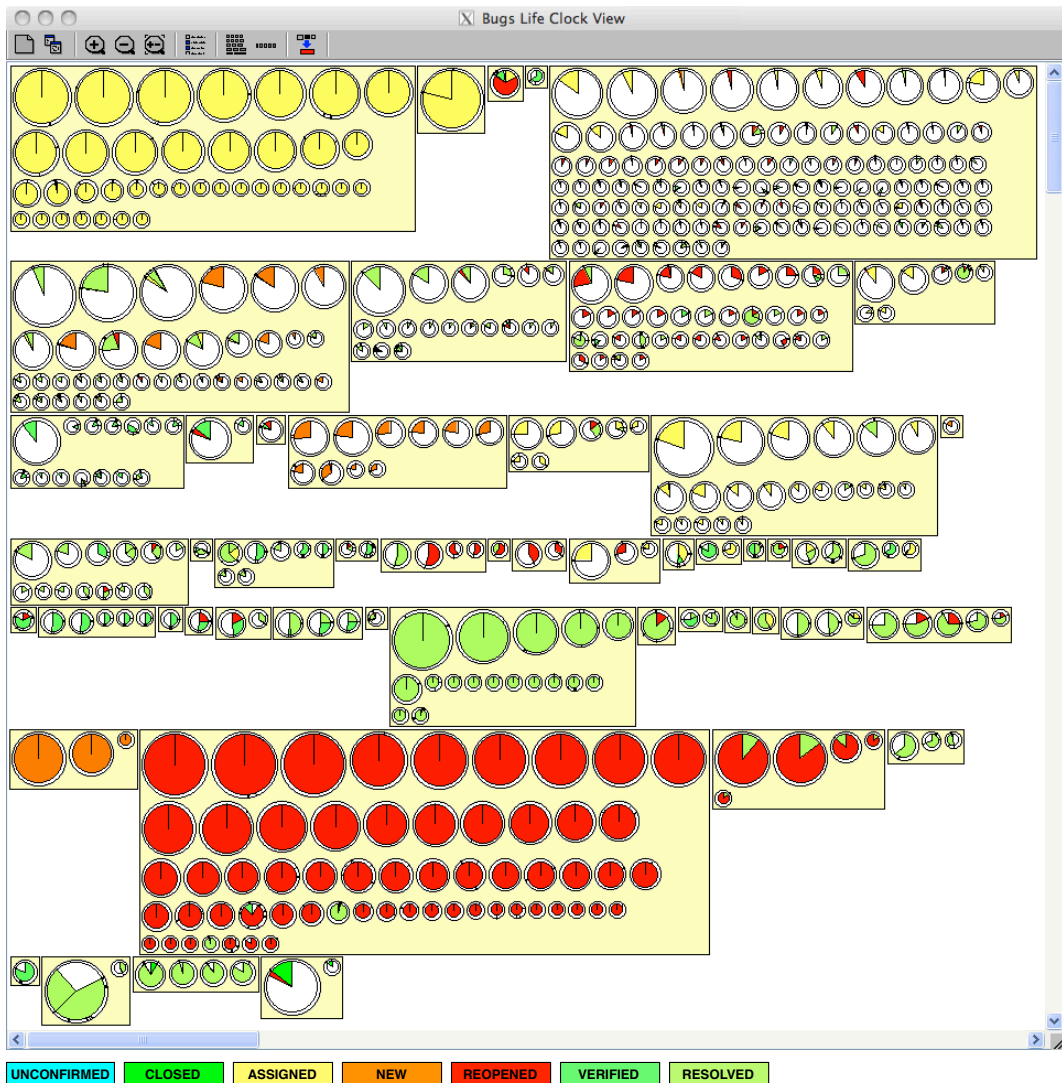


Figure 5.6. Clustering bugs according to the similarity of their life cycles. The clustering eases comparing bugs and spotting “exceptional” bugs.

Figure 5.5 shows a Bug Watch view applied to the rectangle annotated with “7” in Figure 5.2. The area covers the Networking component from November 2002 to April 2003. However, given the large amount of visualized bugs, it is difficult to identify trends in bug histories and to detect peculiar bugs. To address this issue, we devised a clustering technique that works as follows: Bugs having similar life cycles (in the considered time interval) are grouped and placed in the same box; Bugs with a diverse life cycle form single-element boxes. The algorithm then places the boxes in a way that similar boxes, with respect to the similarity of the contained bugs, stay close to each other (cf. Figure 5.6). This clustering technique facilitates the characterization of bugs, according to their life cycles, and the identification of “exceptional” bugs.

5.2 Experiments

The Bug's Life tool can visualize all the models available in the Churrasco framework, as long as they have bug data, *i.e.*, the modeled software project uses a bug tracking system. To assess the usefulness of the presented visualizations, we apply them to the bug database of Mozilla from June 1999 to April 2003, a large dataset consisting of more than 250,000 bugs and 2,700,000 bug activities. We selected this time window for the following reasons: In June 1999—the beginning—Bugzilla was introduced for the first time, and in April 2003—the end—the Mozilla Organization planned to focus the development effort on the new standalone applications (*e.g.*, Firefox and Thunderbird) rather than the Mozilla framework, adding new features and enhancements to the standalone applications only.² Therefore, from June 1999 to April 2003, we have a large dataset focusing on a single application.

Table 5.2. The properties of the five areas highlighted in Figure 5.7

Label	Product	Components	Time interval	Avg. bugs per 3 days	#bugs
1	Browser	Bookmarks, Layout:Form-Controls, Layout:Tables, Plugins, XP-Apps:GUI, Event Handling	May 2001–Jan 2003	215	5,570
2	Browser	Browser-general, Layout, XP Toolkit/Widget, Editor Core, Networking, XP Apps, OJI	Jun 1999–Apr 2003	291	24,407
3	MailNews	Networking IMA, Account Manager	May 2001–Mar 2003	145	874
4	MailNews	Address Book, Composition, Mail Back End, Mail Window Front End	Aug 1999–Apr 2003	408	9,421
5	Tech Evang.	Europe West, US General	Oct 2000–Apr 2003	250	1,871

5.2.1 Analysis in the Large

Figure 5.7 shows Bug's Life visualizing a System Radiography of Mozilla, where we consider open bugs only. The time interval used is three days, meaning that each column of the matrix represents three days of time.

Our goal with this first visualization is to understand where and when the open bugs are concentrated. In Figure 5.7 we see that Browser is the largest product—in terms of number of components—and the most affected by open bugs. It has a large number of dark rows, *i.e.*, components affected by many bugs.

We identified and annotated five system areas with the highest density of open bugs, described in Table 5.2. These areas contain system components which were affected by a large number of open bugs for a long period of time. The average number of bugs per basic time interval (three days) varies from 145 to 408, while the total amount of different bugs varies from 874 to 24,407. The shortest time window is 22 months, the longest 46 months. Such amounts and densities of bugs with such a persistency over time are bad symptoms, which indicate that the components should be reengineered to decrease the number of introduced bugs.

The second goal of our analysis in the large is to understand where and when the open and most severe bugs are located. To do so, we generate a second System Radiography by selecting the bugs with *blocker* or *critical* severity only. Figure 5.8 shows an extract of the obtained

²See the Mozilla roadmap at <http://www-archive.mozilla.org/roadmap/roadmap-02-Apr-2003.html>

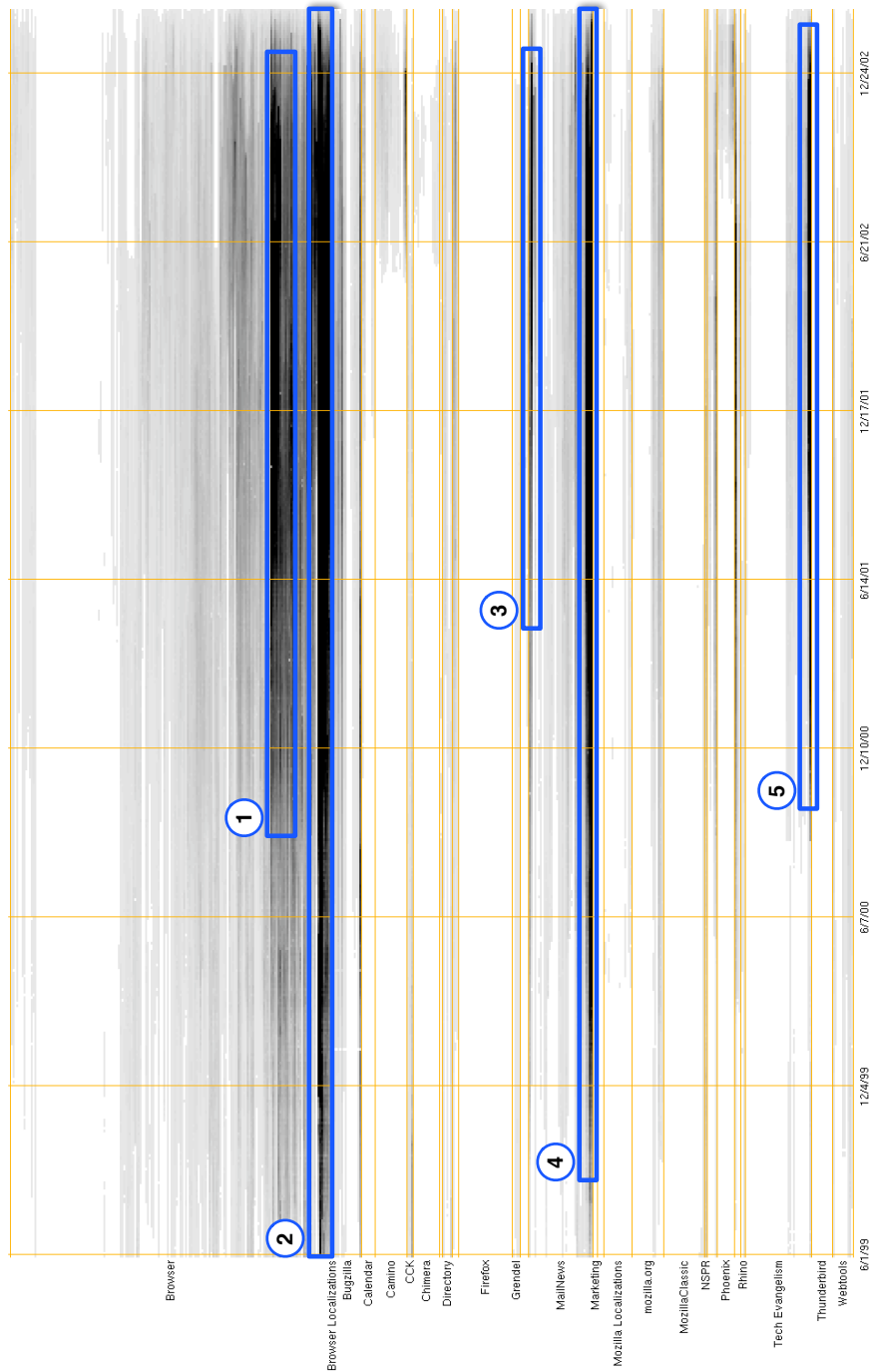


Figure 5.7. A System Radiography of Mozilla from June 1999 to April 2003. We consider bugs with *new*, *assigned* or *reopened* statuses only.

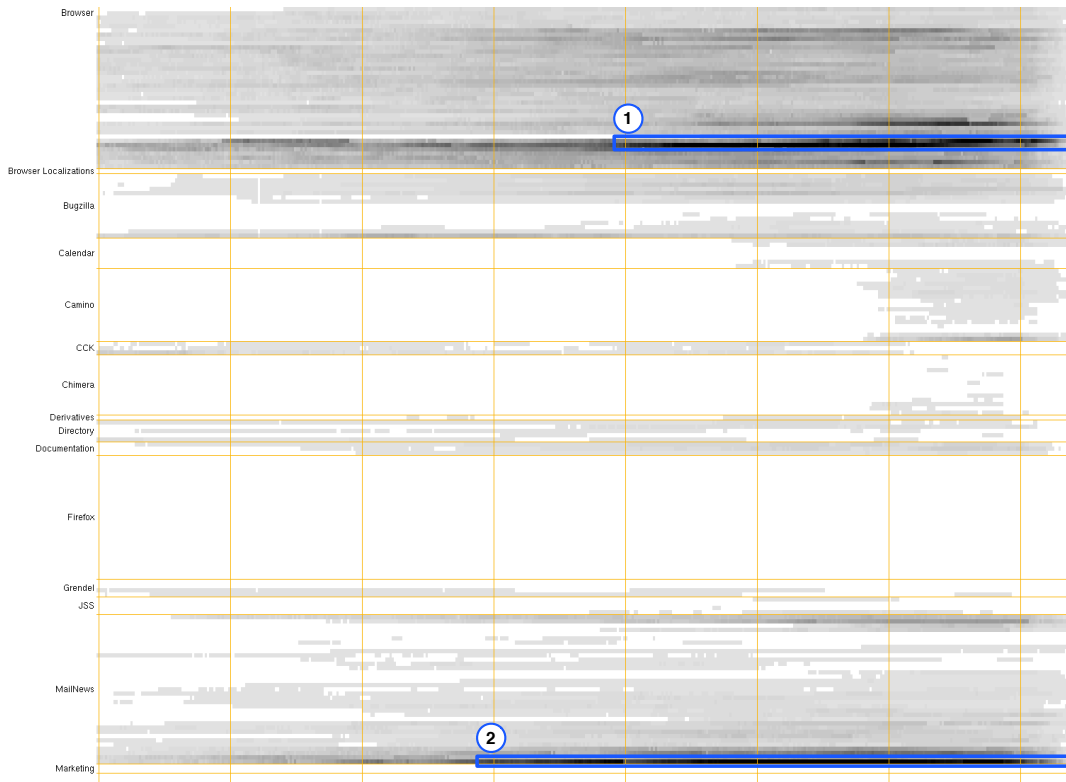


Figure 5.8. An extract of a System Radiography of Mozilla from June 1999 to April 2003, zoomed on the y axis. We consider open bugs with *critical* or *blocker* severity only.

visualization (zoomed on the y axis), where we highlight the two areas having the highest density of open bugs with a critical or blocker severity. The selection marked as “1” includes the components Layout and Networking (belonging to the Browser product), from June 2001 to April 2003. The area marked as “2” covers the Mail Window Front End component of MailNews from December 2000 to April 2003. The average numbers of bugs (per three days) are 158.9 (area 1) and 71.5 (area 2), while the total numbers of different bugs are 2,338 (area 1) and 2,096 (area 2). The three mentioned components are also part of selection “2” and “4” in Figure 5.7 (as listed in Table 5.2). We conclude that they are the most critical components, in terms of being affected by severe bugs.

Summing Up

With limited time and resources, it is important to prioritize the reengineering effort on the most troublesome and critical components of a software system. Using the System Radiography view and the interactive features of Bug’s Life, we detected three components that should have a high reengineering priority, as they were consistently affected by a large number of severe bugs. We also selected a number of components (listed in Table 5.2) which—with a lower priority—should be considered as starting points for reengineering. They have a lower priority because, although consistently affected by a large number of bugs, these bugs are not severe.

5.2.2 Analysis in the Small

With the System Radiography we detected a number of critical components in Mozilla. Now, by means of the Bug Watch visualization, we focus our analysis on one of these components, namely Networking belonging to the Browser product. Since the time window selected in the analysis in the large (June 1999 – April 2003) is long, for the analysis in the small we reduce it to the last six months.

Figure 5.9 shows a Bug Watch view of the open bugs affecting the Browser::Networking component of Mozilla from November 2002 (mapped to 00:00 on the watch) to April 2003 (mapped to 23:59). We observe five interesting facts that we annotate with numbers in the figure:

1. It is a crucial bug in the component, since it has blocker severity and maximum priority (dark severity/priority layer), and it was reopened four times. The activity layer shows that the history of the bug is rich of activities. The details of these activities (given in the information panels) tell us that the developer in charge of fixing the bug changed six times and that the bug is popular, since many e-mail addresses were added to the *CC*. The problem is related to SSL channels and proxies, as stated in the bug description.
2. All these bugs are hard to fix, since they were reopened at least one time (and at most three). They have various levels of severity and priority, but only one of them (marked also as “4”) has a critical severity.
3. These bugs have an unusual life cycle: They passed from a resolved or verified status to an unconfirmed or new status, without being reopened. By reading the activity details, we discovered that this behavior is associated with a change of person in charge (*assignedTo*) and often with a change of quality assurance person (*qa*).
4. All these bugs have the maximum level of priority and severity (critical or blocker).
5. This bug has a life cycle composed of one status only, but its history is full of activities. All these activities are additions of *CC*, meaning that the bug is very popular.

All the other bugs have a “normal” life cycle for bugs that are not yet fixed, following for example the transitions unconfirmed → new or new → assigned. None of these bugs has both maximum priority and blocker or critical severity.

Summing Up

Bugs are not all the same. When dealing with software systems as big as Mozilla, with thousands of bugs, knowing which bugs to fix first is a valuable piece of information. By using the System Radiography view one can decide where to start the reengineering of a large system. Subsequently, the Bug Watch visualization offers a visual means to determine which bugs to fix first, for example because they are severe or popular. The visualization is also useful to characterize bugs based on their life cycles, and to assess the effort required for fixing them: For example, bugs reopened several times, or bugs for which it is difficult to find the knowledgeable developer (*i.e.*, the person in charge to fix the bug changed many times) are hard to fix.

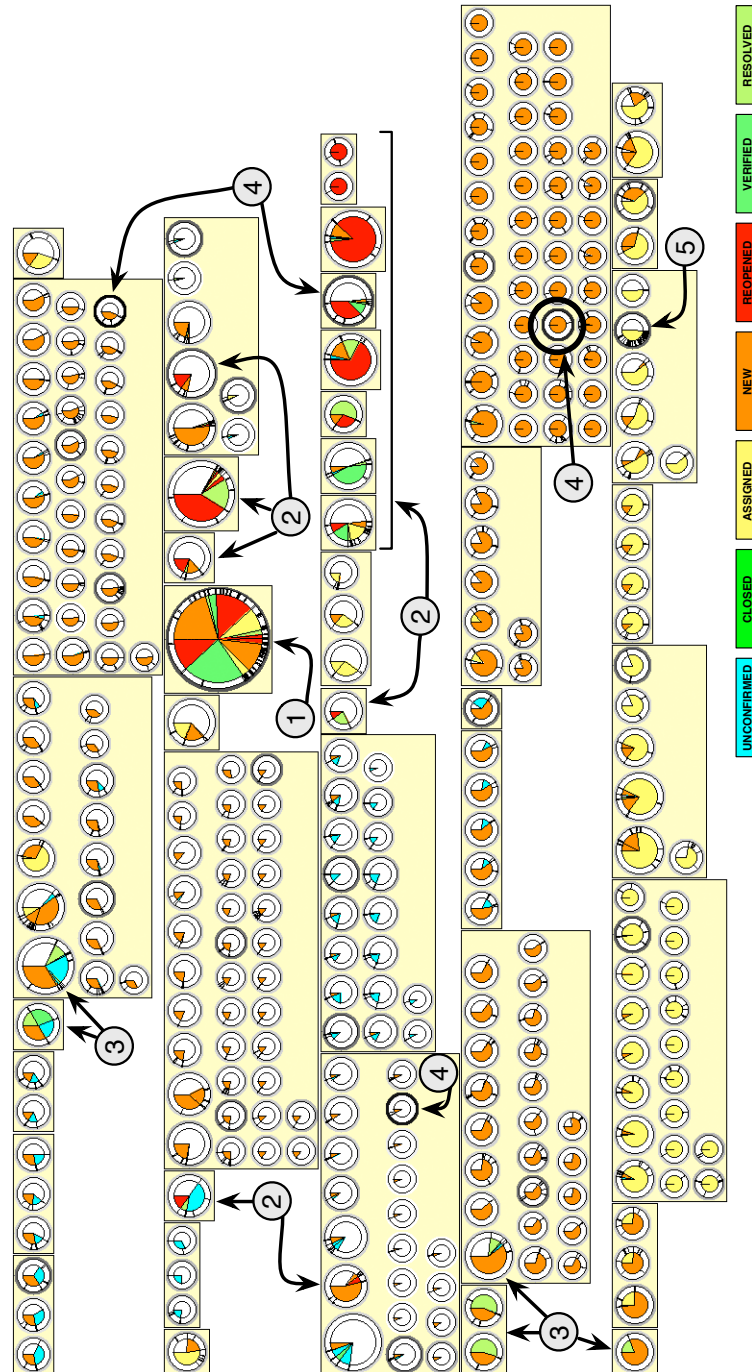


Figure 5.9. A Bug Watch visualization of the bugs affecting the Browser::Networking component of Mozilla. The reference time interval is November 2002 – April 2003.

5.3 Discussion

The proposed visualizations support the analysis of a bug database at two different but complementary levels of granularity. This has two main benefits: (1) The technique scales up to the size of Mozilla, *i.e.*, 250,000 bugs and 2,700,000 activities, and (2) it is possible to visualize and inspect any individual bug together with its history.

The interaction capabilities the tool offers, allows the user to “jump” from the large scale System Radiography to a detailed Bug Watch view, which visualizes a selected part of the system or even a single bug. Bug’s Life allows also the user to customize the views by applying filters to bug properties such as priority and severity.

Another advantage of the approach is that both visualizations include the time dimension. In particular the Bug Watch view has the time embedded in each figure. This permits any type of layout and grouping, without loosing or modifying time-based properties of bugs (*e.g.*, life cycle and activities). The last benefit comes from the sorting and grouping techniques used in both views, which ease the detection of critical system’s areas and the characterization of bugs.

Concerning limitations, the presented approach deals with bugs only, and not with source code: When a bug has a valid link to a source code artifact (as modeled in Mevo), the Bug’s Life tool does not let the user “jump” to the source code. However, while such a feature would be useful, the focus of our technique is bugs and bug histories. The System Radiography view is simple and easy to interpret. On the other hand, the Bug Watch visualization needs a trained eye to exploit its potential.

Finally, regarding the experiments that we performed with the Mozilla case study, we do not consider them as a full fledge validation of our approach, but an anecdotal evidence of it.

5.4 Summary

We presented an approach to support the analysis of a bug database, by means of two interactive visualizations: The *System Radiography* and the *Bug Watch* views. The System Radiography is aimed at studying the bug database in the large: The visualization is helpful to understand how the open bugs are distributed in the system components and over time. It also highlights the critical parts of the system, *i.e.*, the components affected by the most severe bugs. The Bug Watch view supports the analysis of the bugs affecting a limited part of the system, *e.g.*, one or few components. The visualization eases the characterization of bugs, the identification of the critical ones and the assessment of the required fixing effort.

We applied our approach on the bug database of Mozilla, consisting of more than a quarter million bugs with more than two millions activities. Through this large case study, we showed how one can use our visual technique to answer questions concerning the resource optimization problem, such as: From which components should I start the reengineering of the system? Given a certain component, which bugs should I fix first? How difficult will it be to fix this bug?

Analyzing the evolution of software defects is complementary to studying source code evolution. In the last two chapters we discussed two techniques to support these two activities in isolation. The next chapter focuses on their combination, *i.e.*, on the co-evolution of source code and software defects.

Chapter 6

Code and Bug Co-Evolution Analysis

In Chapters 4 and 5 we studied the evolution of source code and the evolution of software defects. In this chapter our goal is to analyze their co-evolution.

As discussed when presenting our Mevo meta-model (cf. Section 3.1), one of its key features is that it integrates evolutionary source code information with bug data. Now we propose a visual approach that exploits this integration and supports the analysis of code and bug co-evolution. We introduce the *Discrete Time Figure*, a visual approach that embeds information concerning development activities and bug data into one simple figure. The visualization also provides structural information (such as a directory hierarchy structure or the number of files a module contains), as it helps understand the relationships among software artifacts.

Our main goal is to make sense of the integrated information, and in particular we aim at:

- Showing the evolution of software entities at different granularity levels in the same way, *i.e.*, using the same visualization.
- Showing structural (such as software metrics), commit- and bug-related information.
- Merging all these kinds of information to obtain a clearer picture of a system's entities evolution. This is a key knowledge for a reengineering activity, since it allows an analyst to detect the system's hotspots, *i.e.*, the starting points for a reengineering process.

There are technical challenges in achieving the above goals, the main one being how to provide a large amount of information in a condensed, yet useful way. This also relates to scalability, *i.e.*, the visualization must work at all granularity levels of large and long-lived systems. We deal with the scalability challenge by abstracting the evolution of code and bugs in “*phases*” that describe trends of activity. However, phases focus on the evolution of code or bugs, but do not consider their interrelationship. Therefore, based on visual properties of Discrete Time Figures, we define a catalog of co-evolutionary patterns that describe such an interrelationship. The patterns allow the characterization and the comparison of a system's components, and they define a vocabulary that is a useful communication means.

Structure of the chapter. In Section 6.1 we discuss the motivation and principles of our approach based on Discrete Time Figures. In Section 6.2 we present a catalog of co-evolutionary patterns that characterize the co-evolution of software artifacts. We then apply our approach to

three large systems (*i.e.*, Apache, gcc and Mozilla) in Section 6.3. Before concluding the chapter in Section 6.5, we discuss the limitations of our approach in Section 6.4.

6.1 Visualizing Evolving Software

Figure 6.1 depicts a possible way to visualize the co-evolution of code and bugs: It shows a *Timeline* view (time is on the horizontal axis from left to right) where the displayed rectangles and crosses represent commits and bugs, respectively.



Figure 6.1. One way of visualizing the co-evolution of code and bugs

The figure shows the co-evolution of two specific files, where we determine the horizontal position of rectangles based on the commit timestamps, while for the position of the crosses we consider when bugs were reported. The color maps author information (we use different colors to represent different authors) for the commits and severity information (red denotes critical or blocker severity and green stands for all the other severity levels) for the bugs.

Using such a figure we can detect the files that had an intense development or that generated many bugs. However, the visualization is limited in the following ways:

1. It does not provide neither a qualitative nor a quantitative impression about the number of bugs and commits over time.
2. It is applicable to files only, and not to directories or modules. Thus, one can use it to study co-evolution at one level of abstraction only.
3. It does not scale: When the number of files, commits or bugs is high, the figures representing them are not intelligible any more.

To overcome these limitations we devised the *Discrete Time Figure*, a visualization that encapsulates commit and bug related information of a software entity. The Discrete Time Figure gives an immediate view of the history of a software entity, with respect to its development intensity (the number of commits) and its problems (the number of bugs). If the software entity is a directory or a module, the number of commits (bugs) is the sum of the number of commits (bugs) of all the files the directory or module contains. The history can be enriched with a software metric and structural information given by the view layout (*e.g.*, the directory hierarchy).

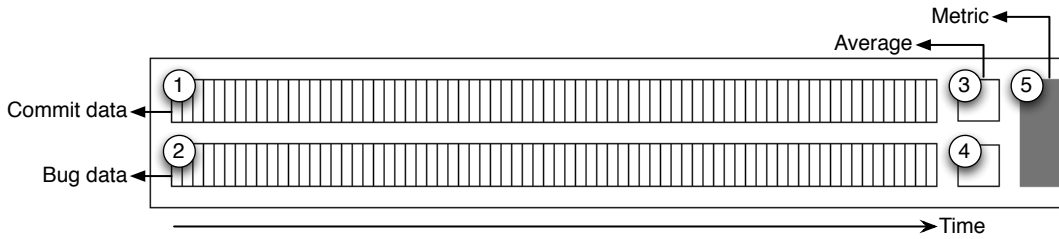


Figure 6.2. The structure of a Discrete Time Figure

Figure 6.2 shows the structure of a Discrete Time Figure. It is composed of five subfigures, marked with numbers in Figure 6.2. The subfigures marked as “1” and “2” are composed of a sequence of rectangles that represents a discretization of time. Each rectangle is associated to a precise and parametrizable interval of time, where two vertically aligned rectangles having the same horizontal position represent the same time period.

We color the rectangles using a heat map, *i.e.*, hot colors (in the red hue range) represent time periods with many commits (or many reported bugs), whereas cold colors (in the blue hue range) represent few commits (or few reported bugs). The white rectangles represent time periods without development activity or without reported bugs. The black rectangles represent time intervals after the removal of the entity from the system (the entity is “dead”). Figure 6.3 provides examples of the color mapping used in the Discrete Time Figure.

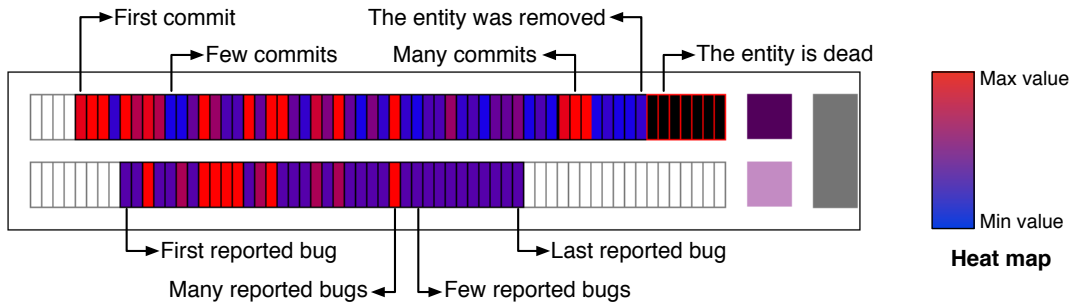


Figure 6.3. The color mapping used in the Discrete Time Figure

The choice of the color is based on a set of thresholds, which can be either manually chosen or automatically computed. For the latter we distinguish two different scenarios:

1. *Local threshold.* We compute the threshold values “locally” for each figure, taking into account only the bugs and commits concerning the target entity. We use the local thresholds to characterize software entities, as discussed in Section 6.2.
2. *Global threshold.* We compute the threshold values “globally”, taking into account all the bugs and commits relating to all the visualized entities. One can use these thresholds for comparisons, since all the visualized figures refer to the same thresholds.

In Figure 6.2 the subfigures marked as “3” and “4” depict the average development intensity and the average number of reported bugs, both over time. We color them according to these

average values: Hot colors represent an intense development (or many reported bugs) consistently over time, as opposed to cold ones, which are related to a limited development activity (or few reported bugs). For the commit box, dark colors close to black mean that the entity was removed from the system at the beginning; On the contrary, bright colors close to white imply that the entity has only recently been created in the system. The two average boxes provide an immediate overview of an artifact's history, allowing us to easily compare it with other artifacts.

In the last subfigure (marked as “5” in Figure 6.2) we map a metric measurement on its color, using the grayscale values: the darker the color, the highest the metric value. We do not always make use of it, but it proved to be useful for high-level entities like directories or modules, where we can use this box to represent—for example—the number of contained files.

Scalability issues

The Discrete Time Figure provides a lot of evolutionary information, but it still does not scale well, as the color of the rectangles becomes undecipherable in zoomed out visualizations, when the figure is small. To solve this problem we introduce the concept of *phases*, by aggregating a sequence of rectangles into a bigger one. In particular, we define the following phases, depicted in Figure 6.4:

- *Stable*. We define the Stable phase as a sequence of (at least) four adjacent time intervals during which the number of commits (or bugs) is constantly low. Color wise, a Stable phase is a sequence of at least four blue rectangles.
- *High Stable*. It is similar to the Stable phase, with the only difference that the number of commits (or bugs) is constantly high instead of low. Color wise, a High Stable phase is a sequence of at least four red rectangles.
- *Spike*. We define the Spike phase as: (1) An initial sequence of (at least) two time intervals during which the number of commits (or bugs) remains low (*i.e.*, two blue rectangles), (2) one time interval during which this number is high (*i.e.*, one red rectangle) and (3) a final sequence similar (with the same characteristics) to the initial one.

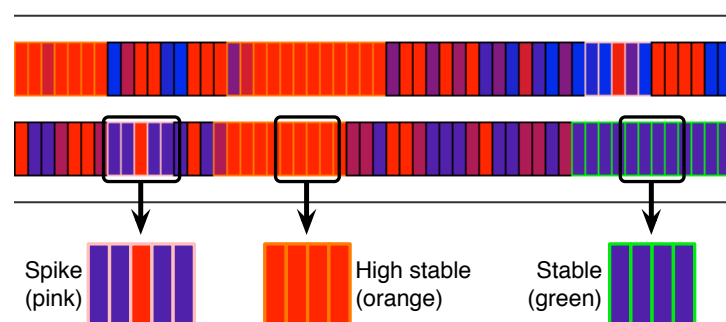


Figure 6.4. Introducing phases in a Discrete Time Figure

In a Discrete Time Figure we highlight the phases by coloring their boundaries with different colors: green for the Stable, orange for the High Stable and pink for the Spike phase. We also use the boundary color to display the fact that entities are added to and removed from a system.

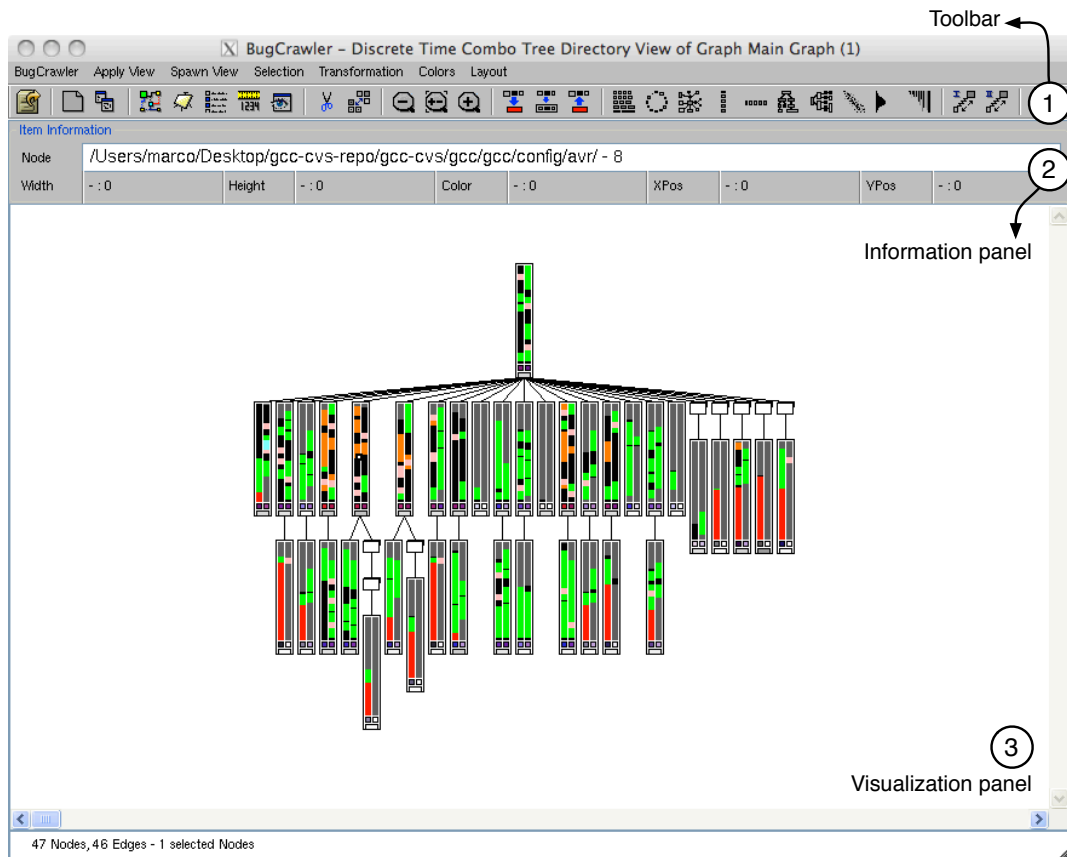


Figure 6.5. Discrete Time Figures applied to a directory tree. The internal color of the rectangles is not visible, while the color of the boundaries is. Thus, phases and addition/removal events are still intelligible.

Figure 6.3 shows that rectangles representing periods of time preceding the first commit (the addition of an entity) have gray boundaries, while rectangles following the removal of an entity have red boundaries. Gray boundaries also represent zero commits or zero reported bugs.

In large scale visualizations phases are still intelligible, as the inner color of the rectangles is not visible while the color of the boundaries is. Figure 6.5 provides an example of such a visualization: It shows a directory tree where we represent directories as vertically aligned Discrete Time Figures (or as white rectangles in case they do not contain files). In these figures the color of the boundary only is visible, *i.e.*, the phases only are intelligible. Figure 6.5 shows also our tool BugCrawler, composed of three main parts (marked with numbers in the figure): (1) The toolbar, used to perform actions such as creating visualizations, applying layouts, zooming, configuring the views, *etc.*, (2) the information panel, providing information about the selected entity and (3) the visualization panel, where the main view is rendered. BugCrawler provides a number of interactive features such as: spawning a new visualization on a selection at different granularity levels (*e.g.*, when selecting directories it is possible to spawn visualizations of files), inspecting an entity represented by a figure, inspecting bug reports, jumping to the code, *etc.*

Coloring the rectangle boundary is useful in large scale views but, at the same time, it might make the figure slightly confusing in small scale views, especially for inexperienced users. To address this problem, in BugCrawler the user can dynamically set the boundary coloring according to the visualization scale.

In our Discrete Time Figure visualizations we use many different colors with different meanings: We summarize all of them in Table 6.1.

Table 6.1. The colors used in the Discrete Time Figure and their meanings

Color	Inner area	Border	Meaning
Red	X		Many commits or many reported bugs
Blue	X		Few commits or few reported bugs
White	X		Zero commits or zero reported bugs
Black	X		The entity was removed
Gray		X	Zero commit or zero reported bugs
Red		X	The entity was removed
Orange		X	High Stable phase
Green		X	Stable phase
Pink		X	Spike phase

6.2 Co-Evolutionary Patterns

Given a software entity, its Discrete Time Figure representation shows the development activity history and the evolution of the problems affecting the entity. By using and combining these pieces of information—and especially their visual representation—we define a catalog of co-evolutionary patterns that characterize the evolution of software entities. In the following, we describe each pattern in detail, using a fixed template: First, we provide the formal definition of the pattern; then, we discuss how to interpret it; finally, we describe variations of the pattern (when applicable) and show an example.

The catalog includes two types of patterns:

1. Patterns characterizing the entire history of a software entity: persistent, day-fly, introduced for fixing, and stabilization. A software entity can have one instance only of these patterns.
2. Patterns characterizing part of the history of a software entity: addition of features, bug fixing and refactoring / code cleaning. An entity can have multiple instances of these patterns.

6.2.1 Persistent

Definition: We define an entity as *persistent* if the following two conditions hold: (1) it is still “alive” (in the current version of the system) and (2) its lifetime is greater than the 80% of the system’s lifetime, *i.e.*, the commit coverage is at least 80%.

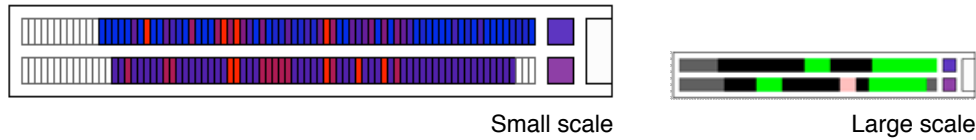


Figure 6.6. An example of a persistent pattern. The entity is also bug persistent.

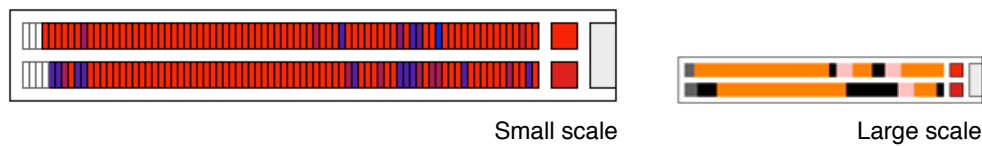


Figure 6.7. An intensive and persistent pattern. The entity is also bug persistent.

Discussion: Persistent entities are likely to play important roles in the system because they survived most of the system’s changes.

Variations: We distinguish two types of persistent patterns:

1. **Bug persistent.** We define an entity as bug persistent if it is persistent and the bug coverage is at least 80% of the commit coverage.¹

The interpretation of this pattern is twofolds: On the one hand its presence can be a good symptom, because it indicates that most of the development was performed together with testing. On the other hand the pattern might reveal that the entity was affected by problems for most of its lifetime.

2. **Intensive and persistent.** A software entity is intensive and persistent if (1) it is persistent and (2) it had an intense development (a high number of commits, corresponding to red rectangles in the Discrete Time Figure) for at least 80% of its lifetime. The intensive and persistent entities are likely to hold a have key role in the system, as changes are concentrated on them. They represent a good starting point for reverse engineering activities.

Example: Figure 6.6 and Figure 6.7 show examples of all persistent patterns: Persistent, bug persistent, and intensive and persistent.

¹ Since the commit coverage for a persistent entity is at least 80% of the system’s lifetime, the bug coverage is at least $80\% \times 80\% = 64\%$ of the system’s lifetime.

6.2.2 Day-Fly

Definition: We define a software entity as a day-fly if its commit coverage is at most 20% of the system's lifetime.

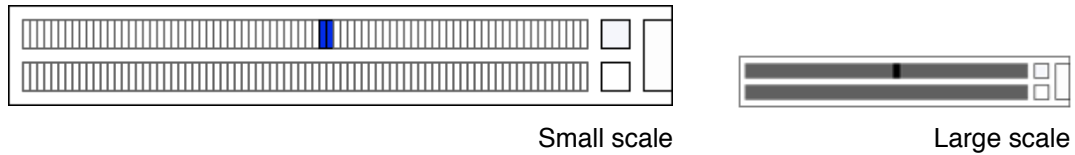


Figure 6.8. A day-fly pattern

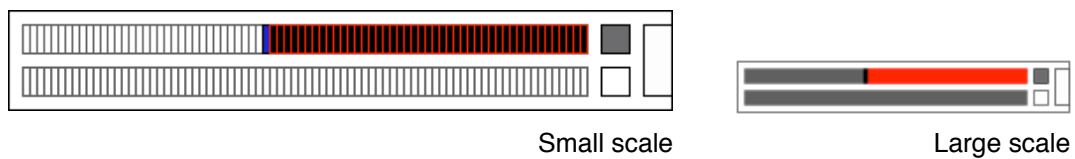


Figure 6.9. A dead day-fly pattern

Discussion: The day-fly pattern represents a specific event in a software repository: A developer added an entity (file, directory or module) in the repository, the entity experienced some changes for a short period of time and then nobody modified it anymore. This event is generic and thus we can associate it with a number of different interpretations, among which: (1) A spike solution, (2) a component no more under development, (3) a component ported from a different system and slightly adjusted to make it work, (4) an entity that—although being already part of the system—was added late to the repository.

Variations: Dead day-fly. It is an artifact removed from the system which lasted at most 20% of the system's lifetime.

As for the day-fly, we can interpret this pattern in different ways, as for example: (1) Renaming,² (2) fast prototyping, (3) a new implementation quickly removed.

Example: Figure 6.8 and Figure 6.9 show examples of respectively a day-fly and a dead day-fly patterns.

²In CVS and SVN a renaming appears as a removal of an entity (the old name) and an addition of another entity (the new name).

6.2.3 Introduced for Fixing

Definition: We define an entity as introduced for fixing if its first commit was after the first bug affecting it was reported.³ Visually, this means that the first colored rectangle in the bug subfigure is before the first one in the commit subfigure (cf. Figure 6.10).

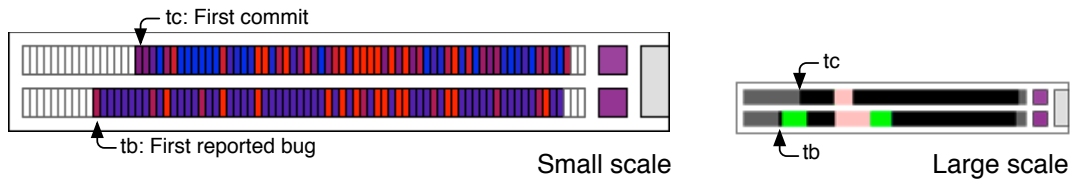


Figure 6.10. An example of the introduced for fixing pattern

Discussion: We interpret this pattern in the following way. A bug b , affecting a certain system component c , was reported at t_1 and, after that, an entity e was introduced in the system at t_2 . To fix b , a developer had to change some entities belonging to c and the entity e , implying that also e was involved in the bug b . One possible explanation for this situation is that the entity e was introduced in the system to fix the bug b affecting the component c . Since the bug b was reported when the software entity e did not exist yet, b could not be caused by e .

Variations: None.

Example: Figure 6.10.

³ This is possible for two reasons: First, in Mevo a bug can be linked with many software artifacts; Second, we establish the link between a bug and a software artifact when the bug is fixed, and not when the bug is reported.

6.2.4 Stabilization

Definition: We define an entity as stabilization if the following conditions hold: (1) The entity is still living (it was not removed), (2) it had an intensive development and it was affected by many bugs⁴ and (3) the last parts of the commits and bugs evolution are stable phases.

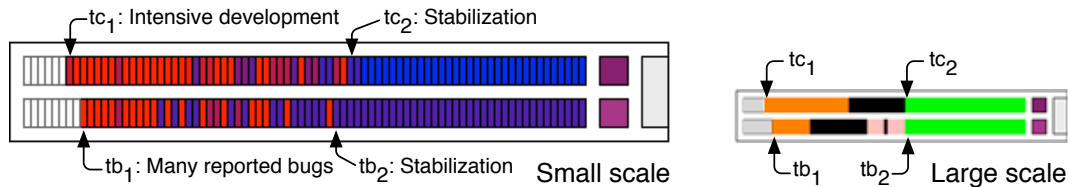


Figure 6.11. A stabilization pattern

Discussion: The stabilization pattern characterizes software entities that experienced an intensive development and caused many bugs to be introduced in the system. These entities, after a certain point, became stable, *i.e.*, their development continued in a regular way with less changes and less introduced bugs. Differently from the day-fly pattern, entities with the stabilization pattern are still under development, as the number of commits and reported bugs is small but not zero. The stabilization pattern suggests that no further analysis is required.

Variations: None.

Example: Figure 6.11.

⁴We consider intensive development (or being affected by many bugs) a high stable phase or a spike phase or (at least) five time intervals with a high number of commits or bugs.

6.2.5 Addition of Features

Definition: An addition of features pattern consists in two contextual periods of time: one with an intensive development and one with many reported bugs.

Formally, we define this pattern as the co-presence of two phases, one for the commits and one for the bugs. We also define time constraints for the two phases: We name $t_{c_{begin}} - t_{c_{end}}$ the time period of the commit phase and $t_{b_{begin}} - t_{b_{end}}$ the time period for the bug phase. We define a software entity to have an addition of features pattern in the following cases:

- Spike phase (commits) and spike phase (bugs) or spike phase (commits) and high stable phase (bugs), with the following time constraint:

$$t_{c_{begin}} \leq t_{b_{begin}} \leq t_{c_{end}} + 2 \text{ time interval}$$

- High stable phase (commits) and high stable phase (bugs) or high stable phase (commits) and spike phase (bugs), with the time constraint:

$$t_{c_{begin}} \leq t_{b_{begin}} \leq t_{c_{end}} - 2 \text{ time interval}$$

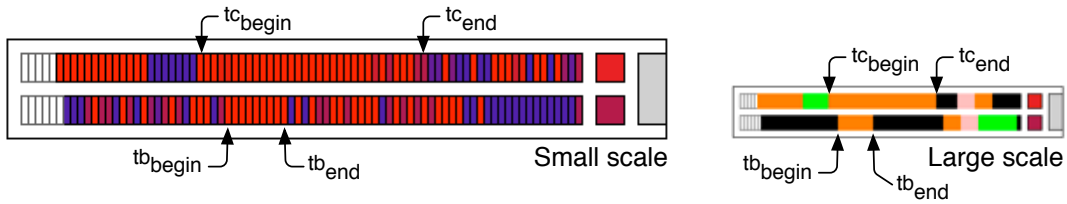


Figure 6.12. An example of the addition of features pattern with the high stable (commits) – high stable (bugs) pair of phases

Discussion: The idea behind this pattern is that a development effort—revealed by a high number of commits for a considerable period of time—caused many bugs to be introduced in the system. We associate the development effort and the addition of bugs with spike or high stable phases. We name the pattern addition of features because adding features is a bug prone development activity. However, this is only one possible interpretation: The changes committed might be related to different development activities that are bug prone, such as bug fixing performed by unexperienced developers.⁵

Variations: None.

Example: Figure 6.12.

⁵Purushothaman and Perry showed that 40% of bugs are introduced while fixing other bugs [PP05].

6.2.6 Bug Fixing

Definition: The bug fixing pattern is symmetrical to the addition of features one. It also consists in two contextual periods of time: one with an intensive development and one with few reported bugs.

Using the terminology formulated for the previous pattern, we define the bug fixing pattern as the co-presence of the following phases:

- Spike phase (commits) and stable phase (bugs), with the following constraint:

$$tc_{begin} \leq tb_{begin} \leq tc_{end} + 2 \text{ time interval}$$

- High stable phase (commits) and stable phase (bugs), with the time constraint:

$$tc_{begin} \leq tb_{begin} \leq tc_{end} - 2 \text{ time interval}$$

In addition, to have a bug fixing pattern, the following condition must hold: The number of bugs reported during the time interval before tb_{begin} is greater than zero. This condition implies that the number of reported bugs decreased at tb_{begin} .

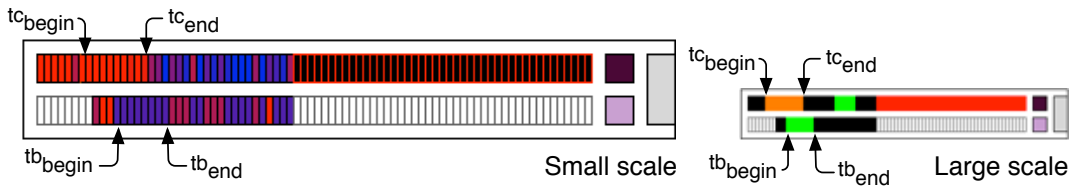


Figure 6.13. An example of the bug fixing pattern with the high stable (commits) – stable (bugs) pair of phases

Discussion: The idea of the bug fixing pattern is that an intensive development activity—revealed by a high number of commits for a considerable period of time—caused the number of reported bugs to decrease. One possible interpretation for this scenario is that the development effort was spent to fix problems. Nevertheless, other explanations are also reasonable, as for example: New features were added without introducing bugs or bugs were actually introduced but not detected or reported.

Variations: None.

Example: Figure 6.13.

6.2.7 Refactoring / Code Cleaning

Definition: We characterize a refactoring / code cleaning pattern with the following conditions:

- (1) The number of reported bugs remained low for a period of time tb and (2) within tb the number of commits increased and then remained high for a period tc included in tb .

Formally, a software entity has a refactoring / code cleaning pattern if:

- It has one of the following pairs of phases: (1) high stable (commits) and stable (bugs), (2) spike (commits) and stable (bugs).
- The pair of phases fulfills the time constraint:

$$tc_{begin} > tb_{begin} \wedge tc_{end} < tb_{end}$$

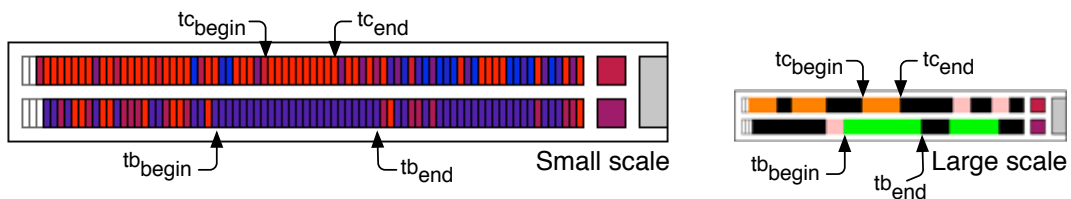


Figure 6.14. An example of the refactoring / code cleaning pattern with the high stable (commits) – stable (bugs) pair of phases

Discussion: This pattern indicates that an intense development activity did not impact the number of reported bugs. Since refactoring and code cleaning activities, if correctly performed, are behavior preserving, they should not introduce bugs. For this reason we name this pattern refactoring / code cleaning. However, we can also interpret this pattern in different ways: Developers added new features without introducing bugs, or developers added features and fixed bugs, or bugs were introduced but not reported.

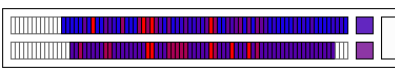
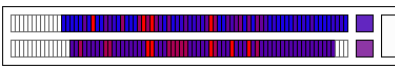
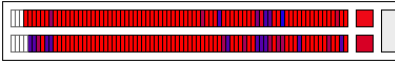
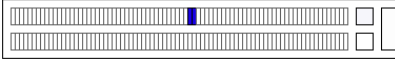

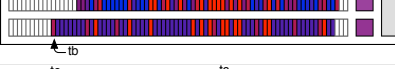
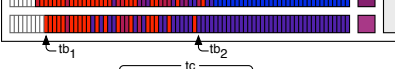
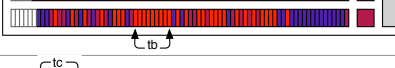
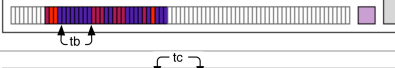
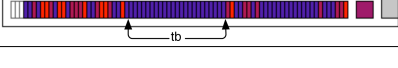
Variations: None.

Example: Figure 6.14.

6.2.8 Summing Up

Based on visual properties of Discrete Time Figures, we defined a catalog of patterns that characterize the co-evolution of code and bugs. Since the patterns are formally defined, we can detect them automatically. In our tool implementation we provide a query engine that detects and counts patterns in a visualization. We recapitulate the patterns, the conditions to detect them and their interpretations in Table 6.2.

Table 6.2. The catalog of co-evolutionary patterns

Name	Condition	Interpretation	Example
Persistent	Lifetime $\geq 80\%$ of the system's lifetime.	Survived most of the system's changes.	
Bug persistent	Persistent and bug coverage $\geq 80\%$ of commits coverage.	Development and testing together or problematic entity.	
Intensive and persistent	Persistent and intensive development for $\geq 80\%$ of its lifetime.	Key entity as changes are concentrated on it.	
Day-fly	Commit coverage $\leq 20\%$ of the system's lifetime.	Spike solution, late addition in the repository, etc.	
Dead day-fly	Day-fly and removed from the system.	Renaming, fast prototyping, etc.	
Introduced for fixing	First commit after first reported bug.	The entity was introduced to fix one or more bugs.	
Stabilization	Intensive development and many bugs followed by stable phases.	The entity got stable	
Addition of features	Intensive development contextual to an increase in the number of bugs.	New features were added and adding them generated bugs.	
Bug fixing	Intensive development contextual to a decrease in the number of bugs.	The development effort was spent to fix problems.	
Refactoring / code cleaning	Intensive development contextual to a constantly low number of bugs	The development effort was spent for refactoring or code cleaning	

The catalog of co-evolutionary patterns allows an analyst to (1) argue about a module or an entire system in terms of the patterns it includes, (2) compare software entities according to the pattern characterization and (3) detect components that need to be further analyzed.

6.3 Experiments

To validate our approach we apply the following methodology on three software systems:

1. We build a view containing all the directories of the target system. Each directory is represented as a Discrete Time Figure if it includes at least one file, or as a white rectangle otherwise.
2. We apply the query engine (provided in our tool) on the view to detect the co-evolutionary patterns. This allows us to characterize the system in terms of the number of patterns it contains.

6.3.1 Case Studies

To perform our experiments we selected the following open source software project:⁶

- Apache HTTP Server. A widely adopted web server for Unix, Linux, OS X and Windows.
- gcc. The GNU compiler collection including front ends for C, C++, Objective-C, Fortran, Java, Ada, *etc.*
- The four largest modules of Mozilla (in terms of files): SeaMonkeyCore, RaptorDist, RaptorLayout and CalendarClient.

Table 6.3 shows the dimensions of the systems in terms of size (number of source code files), code evolution (number of commits) and bugs evolution (number of bugs and bug references⁷).

Table 6.3. The dimensions of the software systems used for the experiments

System	# Source code files	# Commits	# Bugs	# Bug references
Apache	393	15,524	377	717
gcc	18,150	254,785	3,347	12,936
Mozilla modules				
SeaMonkeyCore	4,656	69,391	4,694	16,889
RaptorDist	3,446	50,033	2,753	10,028
RaptorLayout	2,925	99,899	5,797	22,865
CalendarClient	1,860	32,468	2,550	7,001

6.3.2 Characterizing the Evolution of the Systems

Table 6.4 summarizes the number of co-evolutionary patterns that we detect in Apache and gcc. In the following we discuss, for each system, a number of insights that we infer from the occurrences and frequencies of the patterns.

⁶The three software projects are available at: <http://httpd.apache.org>, <http://gcc.gnu.org> and <http://www.mozilla.org>

⁷The number of bugs counts the number of different bug reports, while the number of bug references counts the links with source code artifacts. For example, if a bug is linked with five source code artifacts, then the number of bugs is one, while the number of bug references is five.

Table 6.4. Characterizing Apache and gcc in terms of patterns detected

Pattern	Apache	gcc
Not empty directory	92	1,145
Bug persistent	0	0
High persistent	0	5
Persistent	1	54
Dead day-fly	9	101
Day-fly	12	465
Total day-fly	21	566
Introduced for fixing	2	163
Stabilization	2	1
Spike - spike phases	1	13
High stable - high stable phases	0	10
Spike - high stable phases	0	0
High stable - spike phases	1	23
Total addition of feature	2	46
Spike - stable phase	2	16
High stable - stable phases	9	52
Total bug fixing	11	68
High stable - stable phases	0	8
Spike - stable phases	0	7
Total refactoring / code cleaning	0	15

Apache

The evolution of the Apache web server is characterized by:

- A relatively high number of day-fly (~21–23%) and bug fixing patterns (~11–12%).
- A small number (between zero and two) of all the other patterns.
- Only one persistent pattern, *i.e.*, one directory only “survived” until the current version of Apache and for more than 80% of the system’s lifetime.

From these facts, we infer that the Apache web server is a very active and lively project (many day-flies), without a stable development core, *i.e.*, a set of directories with an intensive development for most of the system’s lifetime (only one persistent and two stabilizations).

gcc

The most interesting facts concerning gcc’s evolution are:

- The day-fly is the most frequent pattern, with 566 occurrences (~ 49%).
- The introduced for fixing is the second most frequent pattern, with 163 instances (~ 14%).
- There is one stabilization pattern only (~ 0.0009%).
- All the other patterns (persistent, addition of features, bug fixing and refactoring / code cleaning) are much less frequent, with a percentage ranging from 0.01% to 0.06%.

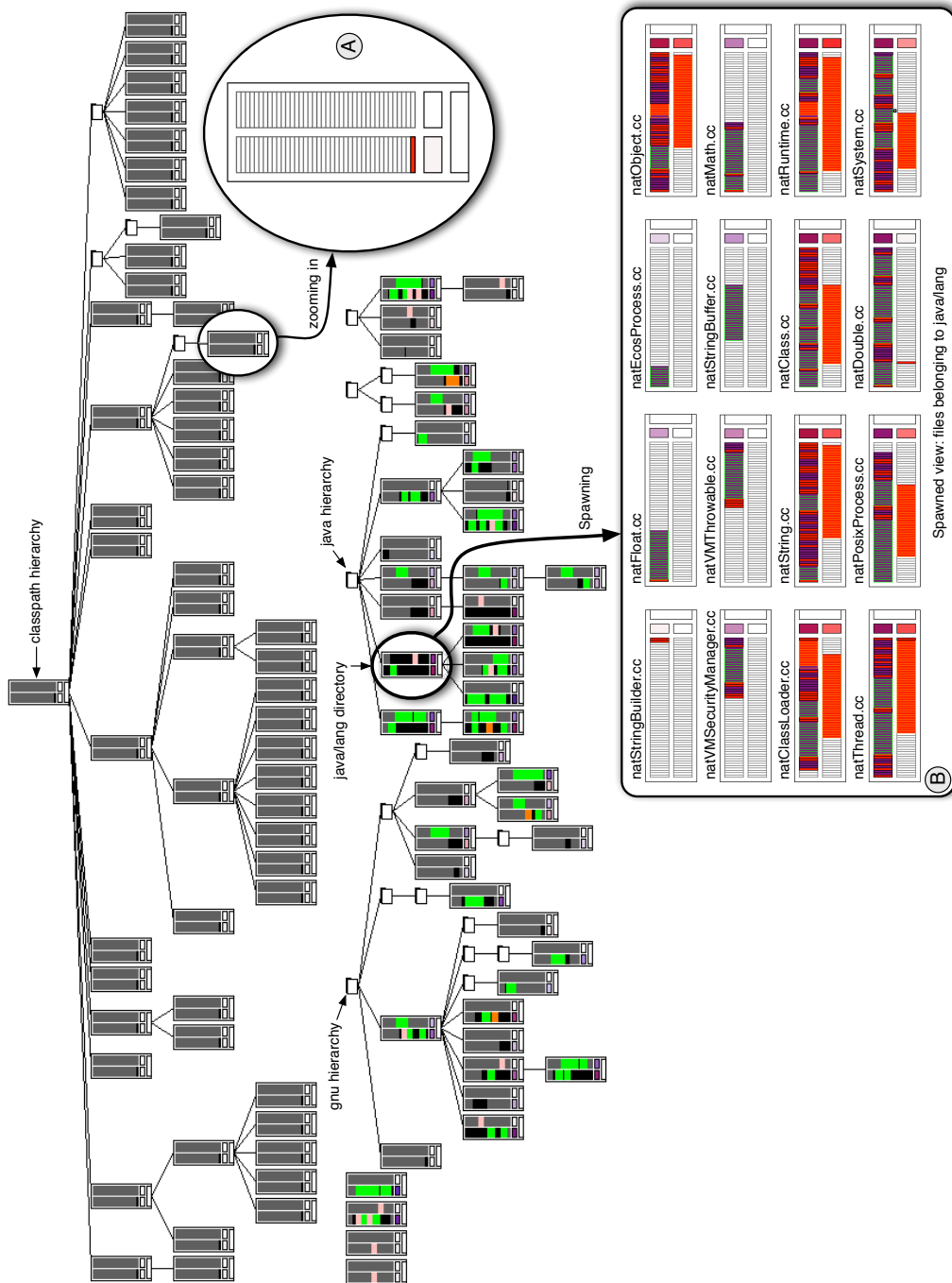


Figure 6.15. A visualization of the libjava module of gcc. The area marked as “B” is a visualization of the files belonging to the java/lang directory.

We conclude that the evolution of gcc was scattered among many different components, with condensed development activities: Half of the directories experienced an intensive development but for a short period of time. This is because gcc includes front ends for many different languages (more than 10) and supports a number of architectures (more than 50). One of the components that underwent a long-lasting and constantly intensive development is the test suite.

Figure 6.15 shows an example of how different patterns are distributed in the gcc system. The figure depicts a visualization of the libjava module of gcc, representing directories as Discrete Time Figures. We see that the entire classpath directory tree is characterized by a day-fly pattern, detailed in the area marked as “A”: All the directories were recently introduced in the system, without generating bugs (yet). The other two big hierarchies in the module are `gnu` and `java`: The directories belonging to these hierarchies experienced a long-lasting development—although seldom intensive—and generated many bugs, especially those belonging to the `java` hierarchy.

We study more closely the directory `java/lang`, characterized by a persistent and bug persistent patterns. Using the interactive features of our tool, we spawn a new visualization of the files included in the `java/lang` directory, depicted in the area marked as “B” in Figure 6.15. In the view we represent files as horizontally aligned Discrete Time Figures. We see that for some files the development was concentrated at the beginning of the `java/lang` history (`natFloat.cc`, `natEcosProcess.cc` and `natMath.cc`), while for others it was in the middle (`natStringBuffer.cc`) or in recent times (`natStringBuilder.cc`, `natVMSecurityManager.cc` and `natVMThrowable.cc`). All these files did not generate bugs. Finally, we observe that all the other files experienced a long-lasting development with periods of intensive activities. As oppose to the first group of files, they all generated bugs.

Mozilla

Table 6.5 shows the number of patterns detected in four modules of Mozilla. We observe that:

1. The `SeaMonkeyCore` module—although not the biggest in terms of not empty directories—has the maximum number of occurrences of most of the patterns (all but persistent and stabilization).
2. `RaptorDist`—the biggest module—underwent a continuous and stable development, as it has the maximum number of persistent and stabilization patterns.
3. The percentage of persistent and stabilization patterns is constant across modules, ranging from 11% to 12% (persistent) and from 0.03% to 0.04% (stabilization). The percentage of day-fly pattern is more variable: It ranges from 23% (`RaptorDist`) to 31% (`CalendarClient`). Since in all the modules the day-fly pattern is more frequent than the persistent and stabilization ones, we conclude that the software artifacts that experienced a long-lasting development, surviving for most of the system’s lifetime, are a minority.
4. Mozilla went through substantial changes during its evolution, since the instances of the addition of features pattern are much more than the bug fixing and refactoring / code cleaning ones.

To provide an anecdotal example of our approach, we present a visualization of one of the Mozilla’s module, namely `CalendarClient` (see Figure 6.16). We inspect the view to understand how and where the various patterns are distributed within the module. In particular we spot the following sets of directories:

Table 6.5. Characterizing the four biggest modules of Mozilla in terms of patterns detected

Patterns	SeaMonkeyCore	RaptorDist	RaptorLayout	CalendarClient
Not empty directory	476	520	382	269
Bug persistent	47	35	36	25
High persistent	15	0	10	3
Persistent	59	60	46	29
Dead day-fly	68	98	68	37
Day-fly	62	26	39	49
Total day-fly	130	124	107	86
Introduced for fixing	77	50	57	50
Stabilization	16	25	14	12
Spike - spike phases	10	8	6	2
High stable - high stable phases	56	15	41	12
Spike - high stable phases	3	0	2	2
High stable - spike phases	46	11	30	13
Total addition of feature	115	34	79	29
Spike - stable phases	7	10	7	5
High stable - stable phases	20	13	20	11
Total bug fixing	27	23	27	16
High stable - stable phases	32	11	20	11
Spike - stable phases	21	13	10	5
Total ref. / code cleaning	53	24	30	16

- *Removed.* A large amount of directories were removed from the module: They are concentrated in the areas marked as “1”, “3”, “5”, “6” and “9” in Figure 6.16 Moreover, most of the directories in “1” and several of them in “5” and “6” were removed at the same time (approximately at half), suggesting that the module experienced a considerable restructuring at that point in time.
- *Day-Fly.* A considerable amount of directories have a day-fly pattern (areas “2” and “7”). Development activities for most of them were concentrated in recent times. Eight directories only generated bugs, while seven were introduced for fixing bugs.
- *Persistent.* The most interesting part of the module is delimited by the yellow areas (marked as “4” and “8”): It underwent a long-lasting and intensive development and it generated the highest number of bugs. In this part of the module we detect a number of different patterns: addition of features, bug fixing, refactoring / code cleaning, stabilization and introduced for fixing. The areas “4” and “8” are hotspots, where one can start the reverse engineering of the CalendarClient module.

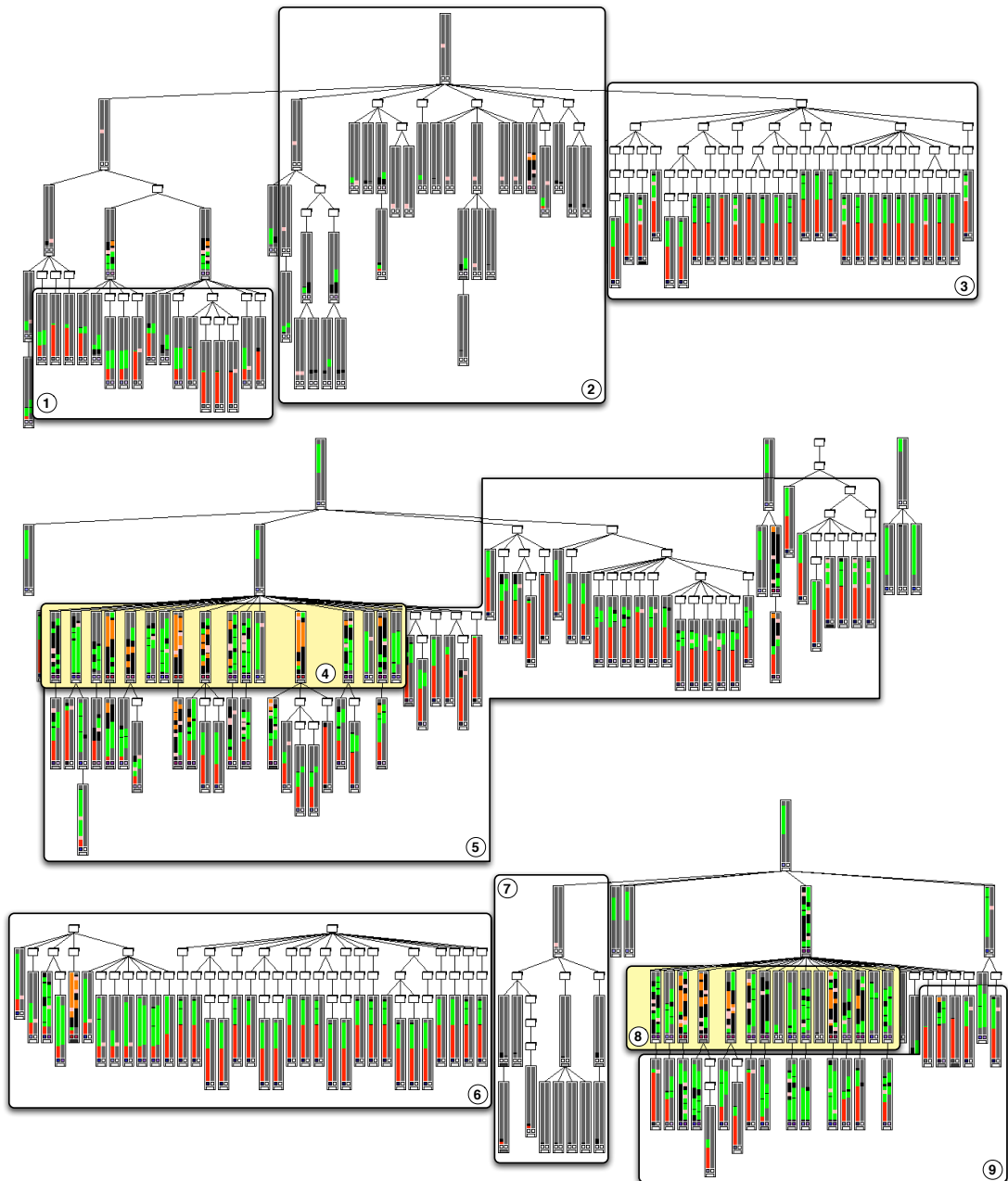


Figure 6.16. A visualization of the CalendarClient module of Mozilla

6.4 Limitations

Large commits. The approach that we presented in this chapter, and especially the interpretation of the co-evolutionary patterns, rely on the data gathered from software repositories and processed with our Churrasco framework. As a consequence, the quality of this data unavoidably impacts the quality of the results produced with our approach.

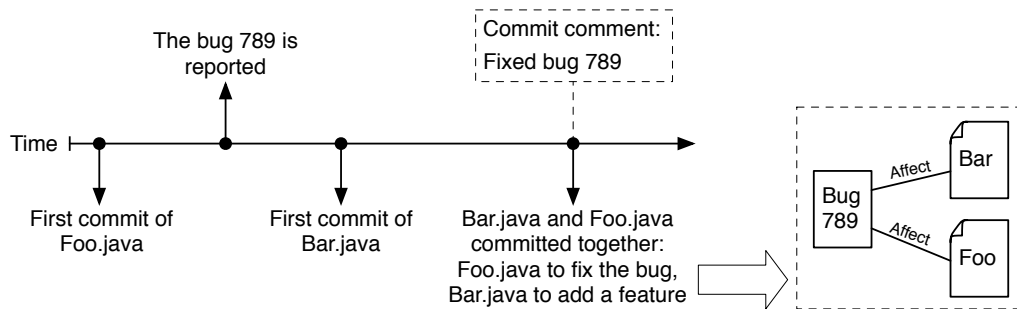


Figure 6.17. An example of a wrong classification: Bar.java is wrongly defined as an introduced for fixing. The cause of the error is that the developer committed a bug fix together with other changes.

For example, we consider the scenario depicted in Figure 6.17: A developer changed the file Foo.java to fix the bug 897 and she also changed the file Bar.java to add a new feature, committing both files together in a unique transaction at time t_3 . Since the bug 897 was reported at time t_1 , the first commit of Bar.java occurred at time t_2 and $t_1 < t_2$, we define Bar.java as an introduced for fixing (a bug was reported before the entity was introduced in the system). Nonetheless, the definition is wrong and the error arises because the developer committed a big fix together with other changes, thus impacting the quality of the links between software artifacts and bugs.

We already extensively discussed this problem (called the “Large commits” problem) when we explained how we retrieve and process data from software repositories (cf. Section 3.2.3). Specifically, we showed that the impact of large commits is limited, as they are very rare.

Names of the patterns. The names that we assigned to the co-evolutionary patterns are only names, and do not imply that a developer performed the activity suggested by the pattern name. For instance, the fact that we detect an addition of features pattern does not imply that a developer actually added some features: The co-evolutionary patterns describe the relationship between the evolution of code and bugs, and their names are one possible interpretation of such a relationship. Defining a pattern vocabulary is useful to characterize a system and, in general, as a communication means: Mentioning a bug fixing pattern is equivalent of reporting that the development activity was intensive while the number of bugs decreased.

Experiments. We applied our technique on three software systems: the Apache web server, the GNU compiler collection (gcc) and the four largest modules of Mozilla. The experiments that we reported are not a complete and in-depth analysis of the systems, but they provide anecdotal evidence of our approach.

6.5 Summary

In this chapter we presented a visual approach to study the relationship between the evolution of software artifacts and the way they are affected by problems. The approach is based on the application of *Discrete Time Figures* at any level of granularity. To tackle the scalability issue, we abstract the commit and bug trends by means of *Phases*.

The Discrete Time Figure indicates patterns of particular interest in the study of a software system's evolution, such as: persistent, day-fly, introduced for fixing, stabilization, addition of features, bug fixing, refactoring / code cleaning. These patterns include some of the possible relationships between the histories of development activities and bugs, providing them a precise and useful meaning. In our tool implementation we provide a query engine that allows the user to automatically detect the patterns presented in this chapter.

An analyst or a project manager can use the Discrete Time Figure visualization and the catalog of co-evolutionary patterns to support the following tasks: (1) characterize a system (or a system's component) in terms of patterns detected, (2) identify areas of interest within a component (for example an analyst can be interested in the entities that exhibit an addition of features pattern) and (3) compare different system's modules in terms of patterns.

In this part of the thesis we studied the evolution of software artifacts (cf. Chapter 4), the evolution of software defects (cf. Chapter 5) and their co-evolution (cf. Chapter 6) to support the retrospective analysis of a software system and infer causes of problems. In the next part, we exploit historical information about source code and bugs to predict future characteristics of a software system.

Part III

Predicting the Future

In the previous part of this dissertation, we introduced three retrospective analysis techniques. The techniques, based on interactive visualizations, supports various maintenance activities aimed at detecting reengineering candidates and inferring the causes of problem in a software system.

In this part of our dissertation, we focus on predicting the future of software systems. To this aim, we present a set of analysis techniques, created—as the ones described in the previous part—on top of our Churrasco framework.

In Chapter 7, we propose two novel defect prediction approaches based on the evolution of various source code metrics over multiple versions of a system. Further, we introduce a publicly available benchmark for comparing prediction techniques. In Chapter 8, we enrich existing defect prediction models with information extracted from development mailing lists. To do so, we extend the Mevo meta-model with e-mail information.

Thereafter, we investigate the relationships between certain characteristics of software entities, known to have a negative impact on quality attributes, and software defects, a tangible effect of low software quality. Such relationships—once understood—can support defect prediction, as they point to defect-prone software artifacts. In Chapter 9, we study the correlation between change coupling and defects. As the correlation holds, we augment defect prediction models with change coupling information. In Chapter 10, we inspect the impact of several design flaws on the presence of software defects. We also analyze the evolution of the flaws over a system's history, to investigate whether adding design flaws is likely to introduce defects.

Chapter 7

Predicting Defects with the Evolution of Source Code Metrics

Defect prediction generated widespread interest for a considerable period of time, leading to more than a hundred publications in the last ten years [MK10]. The driving scenario is resource allocation: Time and manpower being finite resources, it makes sense to assign personnel and resources to software components that are likely to generate bugs.

Researchers proposed a variety of approaches to tackle the problem, relying on diverse information, such as code metrics [BBM96; OA96; BDW99; EMM01; SK03; GFS05; NB05a; NBZ06; OW02; OWB04; OWB07] (*e.g.*, lines of code, complexity), process metrics [NB05b; Has09; MPS08; BEP07] (*e.g.*, number of changes, recent activity) or previous defects [KZWZ07; OWB05; HH05]. The jury is still out on the relative performance of these approaches. Most of them were evaluated in isolation, or were compared to only few other approaches. As a matter of fact, comparing defect prediction techniques is onerous: Different approaches require different types of data from different repositories, *e.g.*, software metrics from the source code, process metrics from the versioning system repository and defect information from the bug database. Moreover, a significant portion of the evaluations cannot be reproduced since the data used by them came from commercial systems and is not available for public consumption. As a consequence, researchers reached opposite conclusions: For example, in the case of size metrics, Gyimothy *et al.* reported good results [GFS05], as opposed to the findings observed by Fenton and Ohlsson [FO00].

What is missing is a baseline against which approaches can be compared. Our integrated Mevo meta-model describes various aspects of a software system's evolution, containing the information required to evaluate several approaches across the bug prediction spectrum. Therefore, exploiting the Mevo meta-model, we provide a baseline to compare defect prediction techniques in the form of a publicly available benchmark. We do so by gathering an extensive dataset composed of several open-source systems. In particular, we provide—for five open-source software systems and over a five-year period—the following data: (1) process metrics on all the files of each system, (2) system metrics on bi-weekly versions of each system, (3) defect information related to each system file, and (4) bi-weekly source code models of each system version if new metrics need to be computed.

We use our benchmark to evaluate a representative selection of defect prediction approaches from the literature. Moreover, we devise two novel defect prediction techniques based on in-

formation extracted from bi-weekly samples of the source code, available in Mevo as well as in our benchmark. These techniques extend previous defect prediction approaches: The first one measures code churn as deltas of source code metrics instead of line-based code churn. The second one extends Hassan's concept of entropy of changes [Has09] to source code metrics.

We evaluate the two novel techniques on the benchmark, obtaining the best and most stable prediction results in our comparison.

Structure of the chapter. In Section 7.1 we motivate the need of a baseline to compare defect prediction techniques. We describe our experiments and evaluation procedure in Section 7.2. In Section 7.3, we detail the approaches that we reproduce and the ones that we introduce. We detail the benchmark dataset in Section 7.4 and report on the performance of the compared approaches in Section 7.5. In Section 7.6, we discuss possible threats to the validity of our findings, and we conclude in Section 7.7.

7.1 Motivations

In Section 2.5.2 we surveyed approaches to defect prediction, the kind of data they require and the various datasets on which they were validated. Here we recall the main bug prediction families and observe why techniques are difficult to compare. Defect prediction approaches can be classified in three main families:

1. *SCM approaches* use information extracted from versioning systems, assuming that recently or frequently changed files are the most probable source of future bugs. These techniques exploit measures such as code churn, entropy of changes (measuring the complexity of code changes), number of authors, file size, file age, *etc.* Other approaches combine versioning system data with information extracted from defect archives, where the hypothesis is that software artifacts presenting defects in the past will suffer them also in the future.
2. *Single-version approaches* assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics. One standard set of metrics used is the Chidamber and Kemerer (CK) metrics suite [CK94]. Other used metrics include the McCabe's cyclomatic complexity, Briand's coupling metrics [BDW99], object-oriented metrics (number of classes, methods, attributes, fan in, fan out and others), *etc.*
3. *Other approaches* exploit different types of data such as network metrics computed on developer-artifact networks or graphs of binary dependencies, cohesion measurements based on information retrieval techniques, call structure metrics, *etc.*

Surveying defect prediction approaches, we observe that they are difficult to compare for three main reasons:

1. *The case study varies.* Varying case studies make a comparative evaluation of the results difficult. Validations performed on industrial systems are not reproducible, because it is not possible to obtain the data that was used. There is also some variation among open-source case studies, as some approaches have more restrictive requirements than others.

2. *The granularity of approaches varies.* Some techniques predict defects at the class level, others consider files, while others consider modules or directories (subsystems), or even binaries.
3. *The prediction task varies.* While some approaches predict the presence or absence of bugs for each component, others predict the amount of bugs affecting each component in the future, producing a ranked list of components. The first ones are usually evaluated using precision, recall, f-measure and accuracy; Techniques belonging to the second group are evaluated with the Pearson's or Spearman's correlation coefficients.

The above reasons explain the lack of comparison between approaches and the occasional diverging results when comparisons are performed.

7.2 Experiments

In this section, we give an overview of the prediction task we perform and justify our choices, then we detail the evaluation strategy for our task, before presenting the tool that we implemented to support bug prediction.

7.2.1 Prediction Task

We compare different bug prediction approaches in the following way: *Given a release x of a software system s , released at date d , the task is to predict, for each class of x , the number of post release defects, i.e., the number of defects reported from d to six months later.* We chose the last release of the system in the release period and perform class-level defect prediction, and not package- or subsystem-level defect prediction, for the following reasons:

- Predictions at the package-level are less helpful since packages are significantly larger. The review of a defect-prone package requires more work than the one of a class.
- Classes are the building blocks of object-oriented systems, and are self-contained elements from the point of view of design and implementation.
- Package-level information can be derived from class-level information, while the opposite is not true.

We predict the number of bugs in each class—not the presence/absence of bugs—as this better fits the resource allocation scenario, where we want an ordered list of classes. We use post-release defects for validation (*i.e.*, not all defects in the history) to emulate a real-life scenario. As Zimmermann *et al.* we use a six months time interval for post-release defects [ZPZ07].

7.2.2 Evaluating the Approaches

To compare bug prediction approaches we apply them on the same software systems and, for each system, on the same dataset. We consider the last major releases of the software systems and compute the predictors up to the release dates.

We base our predictions on generalized linear regression models [DB08] built from the metrics we computed. The independent variables (used for the prediction) are the set of metrics

under study for each class, while the dependent variable (the predicted one) is the number of post-release defects. Following the methodology proposed by Nagappan *et al.* [NBZ06]—and also used by Zimmermann *et al.* [ZN08]—we perform principal component analysis, build regression models, and evaluate explanative and predictive power.

Principal Component Analysis (PCA). PCA [Jac03] is a standard statistical technique to avoid the problem of multicollinearity among the independent variables. This problem comes from intercorrelations amongst these variables and can lead to an inflated variance in the estimation of the dependent variable. We do not build the regression models using the actual variables (e.g., metrics) as independent variables, but instead we use sets of principal components (PC). PC are independent and do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible. We select sets of PC that account for a cumulative sample variance of at least 95%.

Building Regression Models. To evaluate the predictive power of the regression models we do cross-validation: We use 90% of the dataset, *i.e.*, 90% of the classes (training set), to build the prediction model, and the remaining 10% of the dataset (validation set) to evaluate the efficacy of the built model. For each model we perform 50 “folds”, *i.e.*, we create 50 random 90%-10% splits of the data.

Evaluating Explanative Power. To evaluate the explanative power of the regression models we use the adjusted R^2 coefficient. The (non-adjusted) R^2 is the ratio of the regression sum of squares to the total sum of squares. R^2 ranges from 0 to 1, and the higher the value is, the more variability is explained by the model, *i.e.*, the better the explanative power of the model is. The adjusted R^2 , takes into account the degrees of freedom of the independent variables and the sample population. As a consequence, it is consistently lower than R^2 . When reporting results, we only mention the adjusted R^2 . We test the statistical significance of the regression models using the F-test. All our regression models are significant at the 99% level ($p < 0.01$).

Evaluating Predictive Power. To evaluate the predictive power of the regression models, we compute Spearman’s correlation (r_{spm}) between the predicted number of post-release defects and the actual number. The Spearman’s correlation is computed on two lists and is an indicator of the similarity of their order. It ranges from 1 (perfect correlation) to -1 (perfect inverse correlation), where 0 indicates no correlation. We decided to measure the correlation with the Spearman’s coefficient (instead of, for example, the Pearson’s coefficient), as it is recommended with skewed data. Our lists with number of actual and predicted bugs per class are skewed, because most of the classes have no bugs. Such evaluation approach was broadly used to assess the predictive power of a number of predictors [OW02; OWB04; NBZ06; ZPZ07; OWB07].

In the cross-validation, for each random split, we use the training set (90% of the dataset) to build the regression model, and then we apply the obtained model on the validation set (10% of the dataset), producing for each class the predicted number of post-release defects. Then, to evaluate the performance of the performed prediction, we compute the Spearman’s correlation, on the validation set, between the lists of classes ranked according to the predicted and actual number of post-release defects. Since we perform 50 folds cross-validation, the final values of the Spearman’s correlation and adjusted R^2 are averages over 50 folds.

7.2.3 Tool Support

To conduct the experiments presented in the third part of our dissertation, we implemented a tool that we named “*Pendolino*”. *Pendolino* remotely accesses the data residing in the Churrascode framework, *i.e.*, Mevo models, and computes a number of attributes about a given model. For example, *Pendolino* calculates all the metrics used as predictors in the experiments discussed in this chapter. The tool also computes several other metrics that we present in the following chapters.

Pendolino is written in Smalltalk. It does not feature a user interface, but instead it provides a scriptable console to compute metrics and to export them as csv files. Like this, we can import and use the metrics in statistical tools such as Matlab, R or SPSS.¹

We carry out the bug prediction task detailed above by means of Matlab scripts. Such scripts perform Principal Component Analysis, extract generalized regression models, do cross validation, and compute explanative and predictive power for the extracted models.

7.3 Bug Prediction Approaches

Table 7.1 summarizes the bug prediction approaches that we compare. In the following we detail each approach.

Table 7.1. Categories of bug prediction approaches

Type	Rationale	Used by
Change metrics	Bugs are caused by changes.	Moser [MPS08]
Previous defects	Past defects predict future defects.	Kim [KZWZ07]
Source code metrics	Complex components are harder to change, and hence error-prone.	Basili [BBM96]
Entropy of changes	Complex changes are more error-prone than simpler ones.	Hassan [Has09]
Churn (source code metrics)	Source code metrics are a better approximation of code churn.	Novel
Entropy (source code metrics)	Source code metrics better describe the entropy of changes.	Novel

7.3.1 Change Metrics

We selected the approach of Moser *et al.* as a representative, and describe three additional variants.

MOSER. We use the catalog of file-level change metrics introduced by Moser *et al.* [MPS08] listed in Table 7.2. The metric *NFIX* represents the number of bug fixes as extracted from the versioning system log files, not the defect archive. It uses a heuristic based on pattern matching

¹See <http://www.mathworks.com>, <http://www.r-project.org> and <http://www.spss.com>

Table 7.2. Change metrics used by Moser *et al.*

Change Metrics	
NR	Number of revisions (number of versions of an artifact)
NREF	Number of times a file has been refactored
NFIX	Number of times a file was involved in bug-fixing
NAUTH	Number of authors who committed the file
LINES	Lines added and removed (sum, maximum, average)
CHURN	Codechurn (sum, maximum and average)
CHGSET	Change set (transaction) size (maximum and average)
AGE	Age and weighted age

on the comments of every commit. To be recognized as a bug fix, the comment must match the string “%fix%” and not match the strings “%prefix%” and “%postfix%”. The bug repository is not required, because all the metrics are extracted from the CVS/SVN logs, thus simplifying data extraction.

NFIX. Zimmermann *et al.* showed that the number of past defects has the highest correlation with number of future defects [ZPZ07]. We inspect whether the set of change metrics can be reduced to this variable only—an approximation of the actual defect count—and how much precision is lost in the process.

NR. In the same fashion, since Graves *et al.* showed that the best generalized linear models for defect prediction are based on number of changes [GKMS00], we isolate the number of revisions as a predictive variable.

NFIX + NR. We combine the previous two approaches.

7.3.2 Previous Defects

This approach relies on a single metric to perform its prediction. We also describe a more fine-grained variant exploiting the categories present in defect archives.

BUGFIXES. The bug prediction approach based on previous defects, proposed by Zimmermann *et al.* [ZPZ07], states that the number of past bug fixes extracted from the repository is correlated with the number of future fixes. They then use this metric in the set of metrics with which they predict future defects. This measure is different from the metric used in *NFIX* and *NFIX+NR*: For *NFIX*, we perform pattern matching on the commit comments. For *BUGFIXES*, we also perform the pattern matching, which in this case produces a list of potential defects. Using the defect id, we check whether the bug exists in the bug database, we retrieve it and we verify the consistency of timestamps (*i.e.*, if the bug was reported before being fixed).

Variant: BUG-CATEGORIES. We also use a variant in which, as predictors, we use the number of bugs belonging to five categories, according to severity and priority. The categories are: All bugs, non trivial bugs (severity>trivial), major bugs (severity>major), critical bugs (critical or blocker severity) and high priority bugs (priority>default).

7.3.3 Source Code Metrics

Many approaches in the literature use the CK metrics. We compare them with additional object-oriented metrics, and *LOC*. Table 7.3 lists all source code metrics we use. The metrics are computed on the latest version of the system for each eligible class.

Table 7.3. Class level source code metrics

CK metrics	
WMC	Weighted Method Count
DIT	Depth of Inheritance Tree
RFC	Response For Class
NOC	Number Of Children
CBO	Coupling Between Objects
LCOM	Lack of Cohesion in Methods
Other object-oriented metrics	
FanIn	Number of other classes that reference the class
FanOut	Number of other classes referenced by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
LOC	Number of lines of code
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

CK. Many bug prediction approaches are based on metrics, in particular the Chidamber & Kemerer suite [CK94].

OO. An additional set of object-oriented metrics.

CK+OO. The combination of the two sets of metrics.

LOC. Gyimothy *et al.* and Ostrand *et al.* showed that lines of code (*LOC*) is one of the best metrics for fault prediction [GFS05; OW02; OWB04; OWB07]. We treat it as a separate predictor.

7.3.4 Entropy of Changes

Hassan predicts defects using the entropy (or complexity) of code changes [Has09]. The idea consists in measuring, over a time interval, how distributed changes are in a system. The more spread, the higher is the complexity. The intuition is that one change affecting one file only is simpler than one affecting many different files, as the developer who performs the change has

to keep track of all of them. Hassan proposed to use the Shannon Entropy defined as:

$$H_n(P) = - \sum_{k=1}^n p_k * \log_2 p_k \quad (7.1)$$

where p_k is the probability that the file k changes during the considered time interval. Figure 7.1 shows an example with three files and three time intervals.

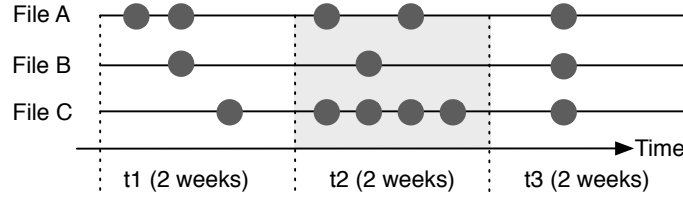


Figure 7.1. An example of entropy of code changes

In the first time interval $t1$, we have a total of four changes, and the change frequencies of the files (*i.e.*, their probability of change) are $p_A = \frac{2}{4}, p_B = \frac{1}{4}, p_C = \frac{1}{4}$. The entropy in $t1$ is therefore:

$$H = -\frac{2}{4} * \log_2 \frac{2}{4} - \frac{1}{4} * \log_2 \frac{1}{4} - \frac{1}{4} * \log_2 \frac{1}{4} = 1$$

In $t2$, the entropy is higher:

$$H = -\frac{2}{7} * \log_2 \frac{2}{7} - \frac{1}{7} * \log_2 \frac{1}{7} - \frac{4}{7} * \log_2 \frac{4}{7} = 1.378$$

As in Hassan's approach [Has09], to compute the probability that a file changes, instead of simply using the number of changes, we take into account the amount of change by measuring the number of modified lines (lines added plus deleted) during the time interval. Hassan defined the Adaptive Sizing Entropy as:

$$H' = - \sum_{k=1}^n p_k * \log_{\bar{n}} p_k \quad (7.2)$$

where n is the number of files in the system and \bar{n} is the number of recently modified files. To compute the set of recently modified files we use previous periods (*e.g.*, modified in the last six time intervals).

To use the entropy of code change as a bug predictor, Hassan defined the History of Complexity Metric (*HCM*) of a file j as:

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} HCPF_i(j) \quad (7.3)$$

where $\{a,..,b\}$ is a set of evolution periods and *HCPF* is defined as:

$$HCPF_i(j) = \begin{cases} c_{ij} * H'_i, & j \in F_i \\ 0, & \text{otherwise} \end{cases} \quad (7.4)$$

where i is a period with entropy H'_i , F_i is the set of files modified in the period i and j is a file belonging to F_i . According to the definition of c_{ij} , we test the following metrics:

- *HCM*: $c_{ij} = 1$, every file modified in the considered period i gets the entropy of the system in the considered time interval.
- *WHCM*: $c_{ij} = p_j$, each modified file gets the entropy of the system weighted with the probability of the file being modified.
- $c_{ij} = \frac{1}{|F_i|}$, the entropy is evenly distributed to all the files modified in the i period. We do not use this definition since Hassan showed that it performs less well than the others.

Concerning the periods used for computing the History of Complexity Metric, we employ two weeks time intervals.

Variants. We define three further variants based on *HCM*, with an additional weight for periods in the past. In *EDHCM* (Exponentially Decayed HCM, introduced by Hassan), entropies for earlier periods of time, *i.e.*, earlier modifications, have their contribution exponentially reduced over time, modeling an exponential decay model. Similarly, *LDHCM* (Linearly Decayed) and *LGDHCM* (LoGarithmically Decayed) have their contributions reduced over time in a respectively linear and logarithmic fashion. Both are novel. The definition of the variants follows (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHCM_{\{a,\dots,b\}}(j) = \sum_{i \in \{a,\dots,b\}} \frac{HCPF_i(j)}{e^{\phi_1 * (|\{a,\dots,b\}| - i)}} \quad (7.5)$$

$$LDHCM_{\{a,\dots,b\}}(j) = \sum_{i \in \{a,\dots,b\}} \frac{HCPF_i(j)}{\phi_2 * (|\{a,\dots,b\}| + 1 - i)} \quad (7.6)$$

$$LGDHCM_{\{a,\dots,b\}}(j) = \sum_{i \in \{a,\dots,b\}} \frac{HCPF_i(j)}{\phi_3 * \ln(|\{a,\dots,b\}| + 1.01 - i)} \quad (7.7)$$

7.3.5 Churn of Source Code Metrics

Using churn of source code metrics to predict post release defects is novel. The intuition is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. We sample the history of the source code every two weeks and compute the deltas of source code metrics for each consecutive pair of samples.

For each source code metric, we create a matrix M where the rows are the classes, the columns are the sampled versions (the FAMIX models in Mevo), and each cell is the value of the metric for the given class at the given version. If a class does not exist in a version, we indicate that by using a default value of -1. We only consider the classes that exist at release x for the prediction.

Starting from the matrix M , we generate a matrix of deltas D , where each cell is the absolute value of the difference between the values of a metric in two subsequent versions (two consecutive cells of the same row of M). If the class does not exist in one or both of the versions (at least one value is -1), then the delta is also -1.

Figure 7.2 shows an example of deltas matrix for three classes. The numbers in the squares are metrics while the numbers in the circles are deltas. After computing the deltas matrix for each source code metric, we compute churn as:

$$CHU(i) = \sum_{j=1}^C \begin{cases} 0, & D(i, j) = -1 \\ PCHU(i, j), & \text{otherwise} \end{cases} \quad (7.8)$$

$$PCHU(i, j) = D(i, j) \quad (7.9)$$

where i is the index of a row in the deltas matrix D (corresponding to a class), C is the number of columns of the matrix (corresponding to the number of samples considered), $D(i, j)$ is the value of the matrix at position (i, j) and $PCHU$ stands for partial churn. In other words, for each class, we sum all the cells over the columns, excluding the ones with the default value of -1. In this fashion we obtain a set of churns of source code metrics at the class level, which we use as predictors of post release defects.

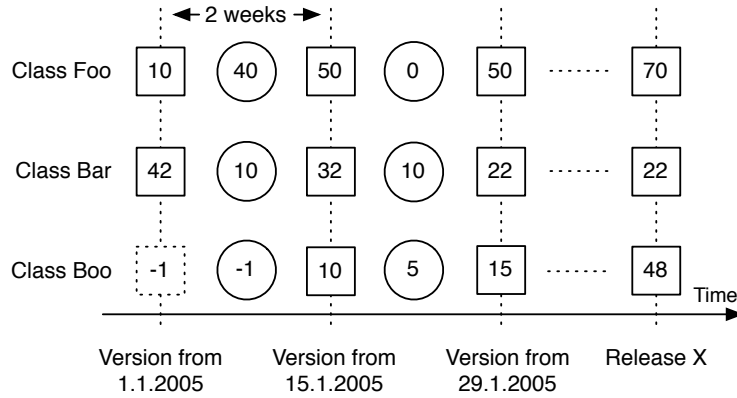


Figure 7.2. Computing metric deltas from sampled versions (FAMIX models) of a system

Variants. We define several variants of the partial churn of source code metrics ($PCHU$): The first one weights more the frequency of change (*i.e.*, the fact that $\Delta > 0$) than the actual change (the delta value). We call it $WCHU$ (weighted churn), using the following partial churn:

$$WPCHU(i, j) = 1 + \alpha * D(i, j) \quad (7.10)$$

where α is the weight factor, set to 0.01 in our experiments. This avoids that a delta of 10 in a metric has the same impact on the churn as ten deltas of 1. In fact, we consider many small changes more relevant than few big changes. Other variants are based on weighted churn ($WCHU$) and take into account the decay of deltas over time, respectively in an exponential ($EDCHU$), linear ($LDCHU$) and logarithmic manner ($LGDPCHU$), with the following partial churns (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDPCHU(i, j) = \frac{1 + \alpha * D(i, j)}{e^{\phi_1 * (C - j)}} \quad (7.11)$$

$$LDPCHU(i, j) = \frac{1 + \alpha * D(i, j)}{\phi_2 * (C + 1 - j)} \quad (7.12)$$

$$LGDPCHU(i, j) = \frac{1 + \alpha * D(i, j)}{\phi_3 * \ln(C + 1.01 - j)} \quad (7.13)$$

7.3.6 Entropy of Source Code Metrics

In this bug prediction approach we extend the concept of code change entropy [Has09] to the source code metrics listed in Table 7.3. The idea is to measure the complexity of the variants of a metric over subsequent sample versions. The more distributed over multiple classes the variant of the metric is, the higher the complexity. For example, if in the system the WMC changed by 100, and one class only is involved, the entropy is minimum, whereas if 10 classes are involved with a local change of 10 WMC, then the entropy is higher. To compute the entropy of source code metrics we start from the deltas matrix, computed as for the churn metrics. We define the entropy—for instance for WMC—for the column j of the deltas matrix, *i.e.*, the entropy between two subsequent sampled versions of the system, as:

$$H'_{WMC}(j) = - \sum_{i=1}^R \begin{cases} 0, & D(i, j) = -1 \\ p(i, j) * \log_{\bar{R}_j} p(i, j), & \text{otherwise} \end{cases} \quad (7.14)$$

where R is the number of rows of the matrix, \bar{R}_j is the number of cells of the column j greater than 0 and $p(i, j)$ is a measure of the frequency of change (viewing frequency as a measure of probability, similarly to Hassan) of the class i , for the given source code metric. We define it as:

$$p(i, j) = \frac{D(i, j)}{\sum_{k=1}^R \begin{cases} 0, & \text{deltas}(k, j) = -1 \\ D(k, j), & \text{otherwise} \end{cases}} \quad (7.15)$$

Equation 7.14 defines an adaptive sizing entropy, because we use \bar{R}_j for the logarithm, instead of R (number of cells greater than 0 instead of number of cells). In the example in Figure 7.2 the entropies for the first two columns are:

$$H'(1) = -\frac{40}{50} * \log_2 \frac{40}{50} - \frac{10}{50} * \log_2 \frac{10}{50} = 0.722$$

$$H'(2) = -\frac{10}{15} * \log_2 \frac{10}{15} - \frac{5}{15} * \log_2 \frac{5}{15} = 0.918$$

Given a metric, for example WMC, and a class corresponding to a row i in the deltas matrix, we define the history of entropy as:

$$HH_{WMC}(i) = \sum_{j=1}^c \begin{cases} 0, & D(i, j) = -1 \\ PHH_{WMC}(i, j), & \text{otherwise} \end{cases} \quad (7.16)$$

$$PHH_{WMC}(i, j) = H'_{WMC}(j) \quad (7.17)$$

where PHH stands for partial historical entropy.

Compared to the entropy of changes, the entropy of source code metrics has the advantage that it is defined for every considered source code metric. If we consider “lines of code” (LOC), the two metrics are very similar: HCM has the benefit that it is not sampled, *i.e.*, it captures all changes recorded in the versioning system, whereas HH_{LOC} —being sampled—might lose precision. For instance, if in the considered time interval one class has first an addition of 10 LOC and then a removal of 10 LOC , for HH_{LOC} the class does not change at all, while with HCM we measure a change of 20 lines. However, using a sample rate of two weeks we do not lose too many changes. On the other hand, HH_{LOC} is more precise, as it measures the real number of lines of code (by parsing the source code), while HCM measures it from the change log, including comments and whitespaces.

Variants. In Equation 7.17 each class that changes between two versions (delta greater than 0) gets the entire system entropy. To take into account also how much the class changed, we define the history of weighted entropy *HWH*, by redefining *PHH* as:

$$HWH(i, j) = p(i, j) * H'(j) \quad (7.18)$$

We also define three other variants by considering the decay of the entropy over time, as for the churn metrics, in an exponential (*EDHH*), linear (*LDHH*), and logarithmic (*LGDHH*) fashion. We define their partial historical entropies as (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHH(i, j) = \frac{H'(j)}{e^{\phi_1 * (C-j)}} \quad (7.19)$$

$$LDHH(i, j) = \frac{H'(j)}{\phi_2 * (C + 1 - j)} \quad (7.20)$$

$$LGDHH(i, j) = \frac{H'(j)}{\phi_3 * \ln(C + 1.01 - j)} \quad (7.21)$$

Based on the previous equations, we define several prediction models using several object-oriented metrics: *HH*, *HWH*, *EDHHK*, *LDHH* and *LGDHH*.

7.4 Benchmark Dataset

Our dataset is composed of the change, bug and version information of the five systems detailed in Table 7.4. All systems are written in Java to ensure that all the code metrics are defined identically for each system. By using the same parser, we can avoid issues due to behavior differences in parsing, a known problem for reverse engineering tools [KSS02]. We filter out test classes from our dataset, since they are not relevant for the defect prediction task.

Table 7.4. Systems in the benchmark

System url	Prediction release	Time period	#Classes	#Versions	#Transactions	#Post-rel. defects
Eclipse JDT Core www.eclipse.org/jdt/core/	3.4	1.01.2005 6.17.2008	997	91	9,135	463
Eclipse PDE UI www.eclipse.org/pde/pde-ui/	3.4.1	1.01.2005 9.11.2008	1,562	97	5,026	401
Equinox framework www.eclipse.org/equinox/	3.4	1.01.2005 6.25.2008	439	91	1,616	279
Mylyn www.eclipse.org/mylyn/	3.1	1.17.2005 3.17.2009	2,196	98	9,189	677
Apache Lucene lucene.apache.org	2.4.0	1.01.2005 10.08.2008	691	99	1,715	103

Figure 7.3 shows the types of information needed by the compared bug prediction approaches: (1) versioning system data to extract process metrics, (2) source code snapshots to compute source code metrics and (3) defect information linked to classes for both the prediction and validation. All this data can be extracted from a model conforming to the Mevo meta-model

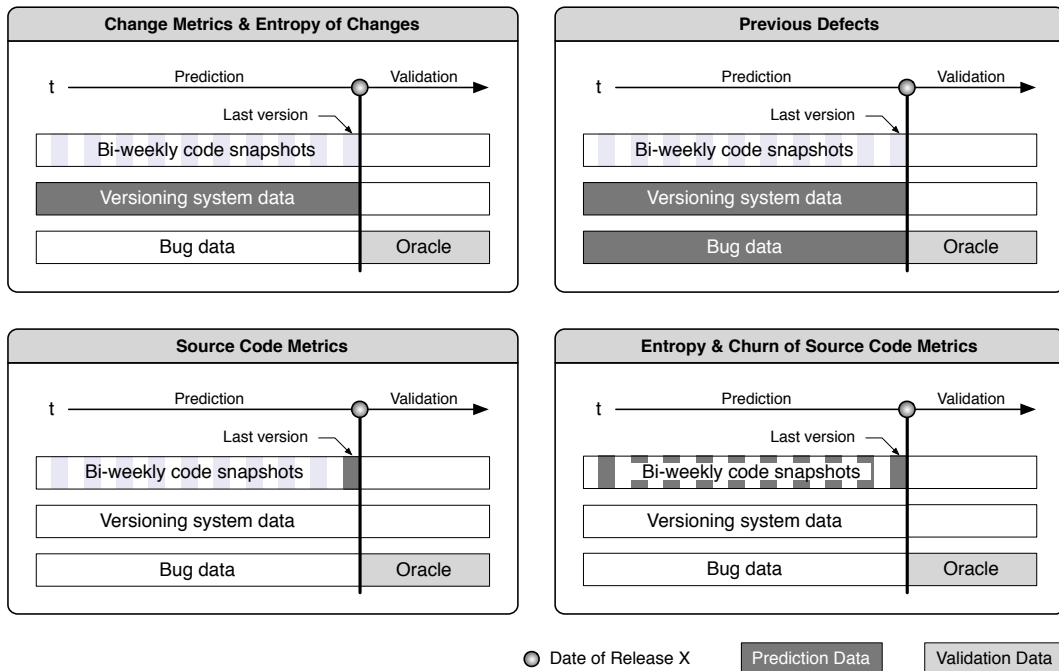


Figure 7.3. The types of data used by different bug prediction approaches.

specification, and thus it is accessible in our Churrascode framework. However, since computing the metrics needed for the prediction is onerous, to provide a benchmark ready to use and to ease the reproducibility of our experiments we distribute the dataset as a self-contained bundle, composed of csv and mse² files. In particular, the bundled dataset contains:

- Bi-weekly snapshots of the systems as FAMIX models.
- Bi-weekly values of 17 source code metrics (*CK* and *OO*) for each class.
- Categorized—with severity and priority—pre- and post-release defect counts for each class.
- Values of 15 change metrics for each class.
- Churn of source code metrics (*CK* and *OO*) over bi-weekly versions of the systems, plus weighted, linear, exponential and logarithmic variants (for each class).
- Entropy of source code metrics (*CK* and *OO*) over bi-weekly versions of the systems, plus weighted, linear, exponential and logarithmic variants (for each class).
- Values of the entropy of code change metric, plus weighted, linear, exponential and logarithmic variants (for each class).

All the metrics included in the dataset are computed by the Pendolino tool, which also generates the csv files. The benchmark is publicly available at <http://bug.inf.usi.ch>.

²mse is a file format used to exchange FAMIX models: <http://scg.unibe.ch/wiki/projects/fame/mse>

Table 7.5. Explanative power for all the bug prediction approaches

Adjusted R^2 - Explanative power						
Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	Score
Change metrics (Section 7.3.1)						
MOSER	0.454	0.206	0.596	0.517	0.57	9
NFIX	0.143	0.043	0.421	0.138	0.398	-3
NR	0.38	0.128	0.52	0.365	0.487	2
NFIX+NR	0.383	0.129	0.521	0.365	0.459	2
Previous defects (Section 7.3.2)						
BF (short for BUGFIXES)	0.487	0.161	0.503	0.539	0.559	5
BUG-CAT	0.455	0.131	0.469	0.539	0.559	5
Source code metrics (Section 7.3.3)						
CK+OO	0.419	0.195	0.673	0.634	0.379	8
CK	0.382	0.115	0.557	0.058	0.368	0
OO	0.406	0.17	0.619	0.618	0.209	6
LOC	0.348	0.039	0.408	0.04	0.077	-3
Entropy of changes (Section 7.3.4)						
HCM	0.366	0.024	0.495	0.13	0.308	-2
WHCM	0.373	0.038	0.34	0.165	0.49	-1
EDHCM	0.209	0.026	0.345	0.253	0.22	-4
LDHCM	0.161	0.011	0.463	0.267	0.216	-4
LGDHCM	0.054	0	0.508	0.209	0.141	-3
Churn of source code metrics (Section 7.3.5)						
CHU	0.445	0.169	0.645	0.628	0.456	8
WCHU	0.512	0.191	0.645	0.608	0.478	<u>11</u>
LDCHU	0.557	0.214	0.581	0.616	0.458	<u>11</u>
EDCHU	0.509	0.227	0.525	0.598	0.467	<u>11</u>
LGDCHU	0.473	0.095	0.642	0.486	0.493	5
Entropy of source code metrics (Section 7.3.6)						
HH	0.484	0.199	0.667	0.514	0.433	7
HWH	0.473	0.146	0.621	0.641	0.484	8
LDHH	0.531	0.209	0.596	0.522	0.343	8
EDHH	0.485	0.226	0.469	0.515	0.359	5
LGDHH	0.479	0.13	0.66	0.447	0.419	4
Combined approaches						
BF+CK+OO	0.492	0.213	0.707	0.649	0.586	<u>13</u>
BF+WCHU	0.536	0.193	0.645	0.627	0.594	<u>13</u>
BF+LDHH	0.561	0.217	0.615	0.601	0.592	<u>15</u>
BF+CK+OO+WCHU	0.559	0.25	0.734	0.661	0.61	<u>15</u>
BF+CK+OO+LDHH	0.587	0.262	0.73	0.68	0.618	<u>15</u>
BF+CK+OO+WCHU+LDHH	0.62	0.277	0.754	0.691	0.65	<u>15</u>

7.5 Results

In Tables 7.5 and 7.6, we report the results of each approach on each case study, in terms of explanative power (adjusted R^2), and predictive power (Spearman's correlation r_{spm}).

Table 7.6. Predictive power for all the bug prediction approaches

Spearman's correlation r_{spm} - Predictive power						
Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	Score
Change metrics (Section 7.3.1)						
MOSER	0.323	0.284	0.534	0.165	0.238	6
NFIX	0.288	0.148	0.429	0.113	0.284	-1
NR	0.364	0.099	0.548	0.245	0.296	5
NFIX+NR	0.381	0.091	0.567	0.255	0.277	4
Previous defects (Section 7.3.2)						
BF (short for BUGFIXES)	0.41	0.159	0.492	0.279	0.377	<u>10</u>
BUG-CAT	0.434	0.131	0.513	0.284	0.353	9
Source code metrics (Section 7.3.3)						
CK+OO	0.39	0.299	0.453	0.284	0.214	8
CK	0.377	0.226	0.484	0.256	0.216	4
OO	0.395	0.297	0.49	0.263	0.214	6
LOC	0.38	0.222	0.475	0.25	0.172	2
Entropy of changes (Section 7.3.4)						
HCM	0.416	-0.001	0.526	0.244	0.308	5
WHCM	0.401	0.076	0.533	0.273	0.288	7
EDHCM	0.371	0.07	0.495	0.258	0.306	3
LDHCM	0.377	0.064	0.581	0.28	0.275	6
LGDHCM	0.364	0.03	0.562	0.263	0.33	5
Churn of source code metrics (Section 7.3.5)						
CHU	0.371	0.226	0.51	0.251	0.292	5
WCHU	0.419	0.279	0.56	0.278	0.285	<u>13</u>
LDCHU	0.395	0.275	0.563	0.307	0.293	<u>11</u>
EDCHU	0.362	0.259	0.464	0.294	0.28	6
LGDCHU	0.442	0.188	0.566	0.189	0.29	7
Entropy of source code metrics (Section 7.3.6)						
HH	0.405	0.277	0.484	0.266	0.318	9
HWH	0.425	0.212	0.48	0.266	0.263	5
LDHH	0.408	0.272	0.53	0.296	0.333	<u>13</u>
EDHH	0.366	0.273	0.586	0.304	0.337	<u>11</u>
LGDHH	0.421	0.185	0.492	0.236	0.347	8
Combined approaches						
BF+CK+OO	0.439	0.277	0.547	0.282	0.362	15
BF+WCHU	0.448	0.265	0.533	0.282	0.31	<u>11</u>
BF+LDHH	0.422	0.221	0.533	0.305	0.352	<u>12</u>
BF+CK+OO+WCHU	0.425	0.306	0.524	0.31	0.298	<u>11</u>
BF+CK+OO+LDHH	0.44	0.291	0.571	0.312	0.377	<u>15</u>
BF+CK+OO+WCHU+LDHH	0.408	0.326	0.592	0.289	0.341	<u>15</u>

We also compute an overall score in the following way: For each case study, add three to the score if R^2 or r_{spm} is within 90% of the best value, one if it is between 75–90%, and subtract one when it is less than 50%. We use this score, rather than an average of the values, to promote consistency: An approach performing very well on a case study, but bad on others

will be penalized. We use the same criteria to highlight the results in Table 7.5 and Table 7.6: R^2 and r_{spm} within 90% of the best value are bolded, the ones within 75% have a dark gray background, while values less than 50% of the best have a light gray background. Scores of 10 or more denote good overall performance; they are underlined.

A general observation is the discrepancy between the R^2 score and the r_{spm} score for entropy approaches (*HCM-LGDHCM*): This is because *HCM* and its variations are based on a single metric, the number of changes, hence it explains a comparatively smaller portion of the variance, despite performing well. Based on the results in the tables, we answer several questions.

What is the overall best performing approach? If we do not consider the amount of data needed to compute the metrics and instead compare absolute predictive power, we can infer the following: The best classes of metrics on all the datasets are the churn and the entropy of source code, with *WCHU* and *LDHH* in particular scoring most of the times in the top 90% in prediction, and *WCHU* having also a good and stable explanative power. Then the previous defects approaches, *BUGFIXES* and *BUG-CAT* follow. Next comes the single-version code metrics *CK+OO*, followed by the entropy of changes (*WHCM*) and change metrics (*MOSER*).

Approaches based on churn and entropy of source code metrics have good and stable explanative and predictive power, better than all the other applied approaches.

What is the best approach, data-wise? If we take into account the amount of data and computational power needed, one might argue that downloading and parsing several versions of the source code is a costly process. It took several days to download, parse and extract the metrics for about ninety versions of each software system. Two more lightweight approaches, which work well in most of the cases, are based on previous defects (*BUGFIXES*) and source code metrics extracted from a single version (*CK+OO*). However, approaches based on bug or multiple versions data have limited usability, as the history of the system is needed, which might be inaccessible or—for newly developed systems—not even existent. This problem does not hold for the source code metrics *CK+OO*, as only the last version of the system is necessary to extract them.

Using the source code metrics (CK+OO) to predict bugs has several advantages: They are lightweight to compute, have good explanative and predictive power and do not require historical information.

What are the best source code metrics? The *CK* and *OO* metrics fare comparably in predictive power (with the exception of *Mylyn*), whereas the *OO* metrics have the edge in explanative power. However, the combination of the two metric sets *CK+OO* is a considerable improvement over them separated, as the performance is more homogeneous across all case studies. In comparison, using lines of code (*LOC*) only, even if it is simple, yields a poor predictor, as its behavior is unstable among systems: Its performance is reasonable on *Eclipse*, *Equinox* and *PDE*, but sub-par on *Mylyn* and *Lucene*.

Using the CK and the OO metric sets together is preferable to using them in isolation, as the performances are more stable across case studies.

Is there an approach based on a single metric with good and stable performances? We have just seen that *LOC* is a predictor of variable accuracy. All approaches based on a single metric, *i.e.*, *NR*, *BUGFIXES*, *NFIX* and *HCM* (and variants) have the same issues: The results are not stable for all the case studies. However, among them *BUGFIXES* is the best one.

Bug prediction approaches based on a single metric are not stable over the case studies.

What is the best weighting for past metrics? In multi-version approaches and entropy of changes, weighting has an impact on explanative and predictive power. Our results show that the best weighting is linear, as models with linear decay have better predictive power and better or comparable explanative power than models with exponential or logarithmic decay (for entropy of changes, churn and entropy of source code metrics).

The best weighting for past metrics is the linear one.

Are bug fixes extracted from the versioning system a good approximation of actual bugs? If we compare the performance of *NFIX* with respect to *BUGFIXES* and *BUG-CAT*, we see that the heuristic searching bugs from commit comments is a poor approximation of actual past defects. On the other hand, there is no improvement in categorizing bugs.

Using string matching on versioning system comments, without validating it on the bug database, decreases the accuracy of bug prediction.

Can we go further? One can argue that bug information is anyways needed to train the model. We investigated whether adding this metric to our best performing approaches would yield improvements at a moderate cost. We tried various combinations of *BUGFIXES*, *CK+OO*, *WCHU* and *LDHH*. We display the results in the lower part of Table 7.5 and Table 7.6, and see that this yields an improvement, as the *BUGFIXES+CK+OO* approach scores a 15 (instead of a 10 or an 8), despite being lightweight. The combinations involving *WCHU*, exhibit a gain in explanative but not in predictive power: The Spearman's correlation score is worse for the combinations (11) than for *WCHU* alone (13). One combination involving *LDHH*, *BF+CK+OO+LDHH*, yields a gain both in explanative and predictive power (15 for both). The same holds for the combination of all the approaches (*BF+CK+OO+WCHU+LDHH*).

Combining bug and OO metrics improves predictive power. Adding this data to WCHU improves explanation, but degrades prediction, while adding it to LDHH improves both explanation and prediction.

7.6 Threats to Validity

Threats to Construct Validity regard the relationship between theory and observation, *i.e.*, the measured variables may not actually measure the conceptual variable. These threats concern the way we link bugs with software artifacts [BBA⁺09] and the noise affecting Bugzilla repositories [AADP⁺08]. We already considered them in Section 3.2.3, when discussing the limitations of populating Mevo models.

Threats to Statistical Conclusion Validity concern the relationship between the treatment and the outcome. In our experiments we used the Spearman’s correlation coefficient to evaluate the performances of the predictors. All the correlations are significant at the 0.01 level.

Threats to External Validity concern the generalization of the findings. We applied the prediction techniques to open-source software systems only. There are certainly differences between open-source and industrial development, and in particular because some industrial settings enforce standards of code quality. We minimized this threat by using parts of Eclipse in our benchmark, a system that while being open-source has a strong industrial background. A second threat concerns the language: All considered software systems are written in Java. Adding non-Java systems to the benchmark would increase its value, but would introduce problems since the systems would need to be processed by different parsers, producing variable results.

To decrease the impact of a specific technology/tool, in our dataset we included systems developed using different versioning systems (CVS and SVN) and different bug tracking systems (Bugzilla and Jira). Moreover, the software systems in our benchmark are developed by independent development teams and emerged from the context of two unrelated communities (Eclipse and Apache).

7.7 Summary

Bug prediction concerns the resource allocation problem: Having an accurate estimate of the distribution of bugs across components helps project managers to optimize the available resources by focusing on the problematic system parts. Different approaches were proposed to predict future defects in software systems, which vary in the data sources they use and in the systems they were validated on, *i.e.*, no baseline to compare such approaches exists.

We introduced a benchmark to allow for common comparison, which provides all the data needed to apply several prediction techniques proposed in the literature. Our dataset, publicly available at <http://bug.inf.usi.ch>, allows the reproduction of the experiments reported in this chapter and their comparison with novel defect prediction approaches. For example, Mende used the dataset and replicated our entire set of experiments [Men10], reporting results consistent with ours.

We evaluated a selection of representative approaches from the literature, some novel approaches we introduced, and a number of variants. Our results showed that the best performing techniques are *WCHU* (Weighted Churn of source code metrics) and *LDHH* (Linearly Decayed Entropy of source code metrics), two novel approaches we proposed. They gave consistently good results—often in the top 90% of the approaches—across all five systems. As *WCHU* and *LDHH* require a large amount of data and computation, past defects and source code metrics are lightweight alternatives with overall good performance. Our results provided evidence that prediction techniques based on a single metric do not work consistently well across all systems.

In this chapter we exploited all the information included in Mevo: Versioning system data to extract process metrics, multiple FAMIX models over time to compute the churn and entropy of source code metrics, and bug information to count pre- and post-release defects. In the following chapter, we go one step further: We extend Mevo to include e-mail archive data and investigate whether such data can be used for defect prediction.

Chapter 8

Improving Defect Prediction with Information Extracted from E-Mails

In the previous chapter, we discussed a number of approaches proposed by researchers to predict software defects, exploiting a variety of sources of information, such as source code metrics, code churn, process metrics extracted from versioning system repositories and past defects. In this chapter, we investigate whether an unexplored source of information, *i.e.*, development mailing lists, can be used for defect prediction. To this aim, we extend the Mevo meta-model with e-mail data.

Due to the increasing extent and complexity of software systems, it is common to see large teams, or even communities, of developers working on the same project in a collaborative fashion. In such cases e-mails are the favorite media for the coordination between the participants. Mailing lists, which are preferred over person-to-person e-mails, store the history of inter-developers, inter-users, and developers-to-users discussions: Issues range from low-level decisions (*e.g.*, bug fixing, implementation issues) up to high-level considerations (*e.g.*, design rationales, future planning).

Development mailing lists of open source projects are easily accessible and they contain information that can be exploited to support a number of activities. For example, researchers can improve the understanding of software systems by adding sparse explanations enclosed in e-mails [ACC⁺02]; the rationale behind the system design can be extracted from the discussions that took place before the actual implementation [LFGT09]; one can assess the impact of source code changes by analyzing the effect on the mailing list [PBD08]; hidden coupling of entities that are not related at code level can be discovered if often mentioned together in discussions.

One challenge when dealing with mailing lists as a source of information is correctly linking each e-mail to any source code artifact it discusses. For this task, we use a lightweight grep-based technique that is able to reach an acceptable level of precision in the linking [BDLR09]. Such a technique allows us to link e-mails with FAMIX classes.

At this point, the question is: Is the information contained in mailing lists relevant for defect prediction? The source code of software systems is only written by developers, who must follow a rigid and terse syntax to define abstractions they want to include. On the other side of the spectrum, mailing lists, even those specifically devoted to development, archive e-mails written by both programmers and users. Thus, the entities discussed are not only the most relevant from a development point of view, but also the most exploited during the use of a software system.

In addition, the content of e-mails is expressed using natural language, which does not require the writer to carefully explain all the abstractions using the same level of importance, but easily permits to generalize some concepts and focus on others. For this reason, we expect information extracted from mailing lists to be independent from those provided by the source code. Thus, e-mails can add valuable information to established defect prediction approaches.

We present different “popularity” metrics that express the importance of each source code entity in discussions taking place in development mailing lists. Our hypothesis is that such metrics are an indicator of possible flaws in software components, thus being correlated with the number of defects. We aim at answering the following research questions:

- *Q1: Does the popularity of software components in discussions correlate with software defects?*
- *Q2: Is a regression model based on popularity metrics a good predictor for software defects?*
- *Q3: Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*

We provide the answers to these questions by validating our approach on four different open source software systems.

Structure of the chapter. In Section 8.1 we articulate the methodology that we follow to conduct our experiments: How we collect, process and analyze the data in order to construct and test popularity metrics. In Section 8.2 we describe the dataset of our case study, how we evaluate the popularity metrics and what are the results achieved. We discuss our findings in Section 8.3. In Section 8.4 we list the possible threats to the validity of our experiments and how we strived to reduce them. We review related work in Section 8.5 and conclude in Section 8.6.

8.1 Methodology

Our goal is first to inspect if popularity metrics correlate with software defects, and then to study whether existing bug prediction approaches can be improved using such metrics. To do so, we follow the methodology depicted in Figure 8.1:

- We extract e-mail data, link it with source code entities (*i.e.*, FAMIX classes) and compute popularity metrics (marked as “1” in Figure 8.1). We extract and evaluate source code and change metrics from Mevo models using our Pendolino tool. The metrics are the same used in Chapter 7, listed in Table 7.2 and Table 7.3.
- We quantify the correlation of popularity metrics with software defects extracted from Mevo (marked as “2”), using as baseline the correlation between source code metrics and software defects (marked as “3”).
- We build regression models with popularity metrics as independent variables and the number of post-release defects as the dependent variable (marked as “4”). We evaluate the performance of the models using the Spearman’s correlation between the predicted and the reported bugs (marked as “5”). We create regression models based on source code metrics and change metrics alone, and later enrich these sets of metrics with popularity metrics, to measure the improvement given by them.

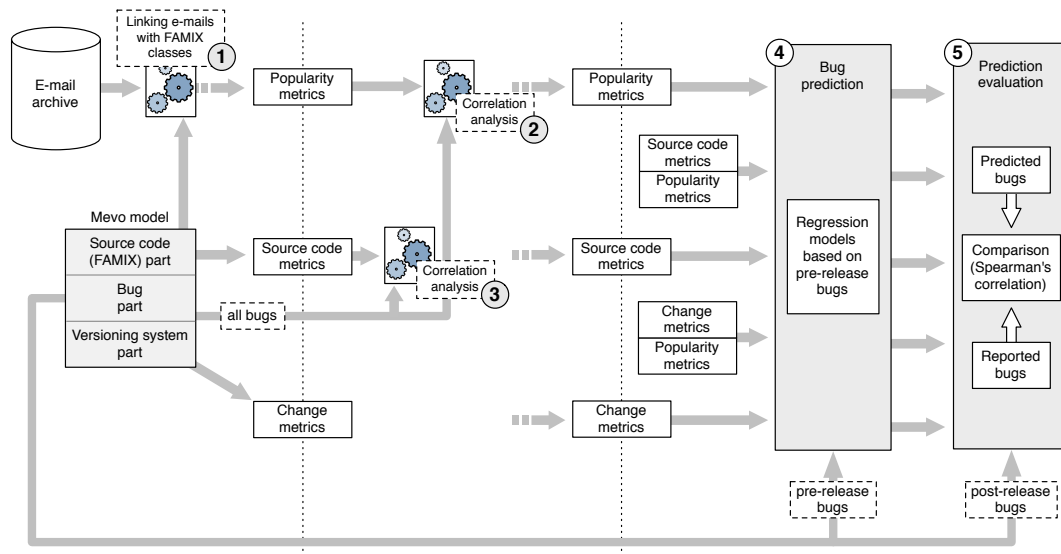


Figure 8.1. Overall schema of our approach

In the experiments, we focus on object-oriented Java software systems using classes as target entities. We decided to focus on classes, and not for example on packages, for the same reasons discussed in Section 7.2.1. We do not filter out test classes, because they are mentioned in e-mail discussions.

To measure the correlation between metrics and defects we consider all the defects, while for bug prediction only post-release defects, *i.e.*, the ones reported within a six months time interval after the considered release of the software system (as in the experiments reported in Chapter 7).

The extraction of popularity metrics, given a software system and its mailing lists, is done in two steps: First it is necessary to extend Mevo to model e-mail data, *i.e.*, link each FAMIX class with all the e-mails discussing it, then the metrics must be computed using the links obtained.

8.1.1 Extending Mevo to Model E-Mail Data

Figure 8.2 shows the technique to link e-mails to classes. First, we parse the target e-mail archive to build an e-mail model including body, headers and additional data about the inter messages relationships, *i.e.*, thread details.

Then, we link each FAMIX class extracted from Mevo with any e-mail referring it, using lightweight linking techniques based on regular expressions, which were proved to be effective [BDLR09]. We obtain a Mevo model enriched with all the connections and information about classes stored in the e-mail archive. Through this model we can extract a catalog of popularity metrics presented next.

Implementation wise, augmenting Mevo to include e-mail data consists in extending the meta-model description: The Meta-base component, part of our Churrasco framework, then automatically generates the importer and exporter to store and retrieve the new data to/from the Churrasco backend database.

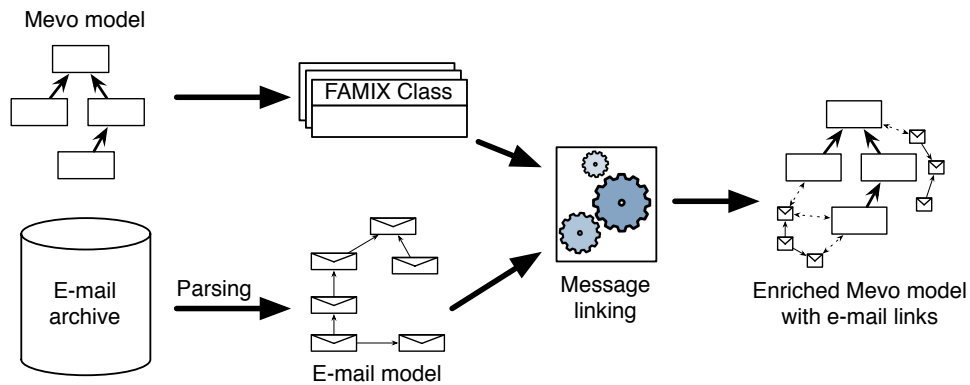


Figure 8.2. Linking e-mails and classes

8.1.2 Popularity Metrics

Table 8.1 lists the popularity metrics that we devised to answer our research questions. For each popularity metric that we propose, we also provide the rationale behind its creation and a high-level description of its implementation, using the enriched Mevo meta-model.

Table 8.1. Class-level popularity metrics

Popularity Metrics	
POP-NOM	Number of e-mails
POP-NOCM	Number of characters in e-mails
POP-NOT	Number of threads
POP-NOMT	Number of e-mails in threads
POP-NOA	Number of authors

POP-NOM: To associate the popularity of a class with discussions in mailing lists, we count the number of e-mails that mention it. Since we consider development mailing lists, we presume that classes are mainly mentioned in discussions about failure reporting, bug fixing and feature enhancements, thus they can be related to defects. Thanks to the enriched Mevo model we generate, it is simple to compute this metric. Once the mapping from classes to e-mails is completed, and the model contains the links, we count the number of links of each class.

POP-NOCM: Development mailing lists can also contain other topics than technical discussions. For example, while manually inspecting part of our dataset, we noticed that voting about whether and when to release a new version occurs quite frequently in the Lucene, Maven and Jackrabbit mailing lists. Equally, announcements take place with a certain frequency. Usually this kind of messages are characterized by a short content (e.g., “yes” or “no” for voting, “congratulations” for announcements). The intuition is that e-mails discussing flaws in the source code could present a longer amount of text than e-mails about other topics. We consider the length of messages taking into account the number of characters in the text of e-mails: We eval-

uate the *POP-NOCM* metric by adding the number of characters in all the e-mails related to a given class.

POP-NOT: It is a long tradition in mailing lists to divide discussions in *threads*. Our hypothesis is that all the messages that form a thread discuss the same topic: If an author wants to start talking about a different subject she can create a new thread. We suppose that if developers are talking about one defect in a class they will continue talking about it in the same thread. If they want to discuss about an unrelated or new defect (even in the same class) they would open a new thread. The number of threads, then, could be a popularity metric whose value is related to the number of defects. After extracting e-mails from mailing lists, our e-mail model also contains the information about threads. Once the related e-mails are available in the Mevo model, we retrieve this thread information from the messages related to each class and count the number of different threads. If two, or more, e-mails related to the same class are part of the same thread, they are counted as one.

POP-NOMT: Inspecting sample e-mails from our dataset, we noticed that short threads are often characteristic of: (1) “announcements” e-mails, (2) simple e-mails about technical issues experimented by new users of the systems or (3) updates about the status of development. We hypothesize that longer threads could be symptoms of discussions that raise the interest of developers, such as those about defects or changes in the code. For each FAMIX class, we consider the threads of all the referring e-mails, and we count the total number of e-mails in each thread. If a thread is composed of more than one e-mail, but only one is referring the class, we still count all the e-mails inside the thread, since it is possible that following e-mails reference the same class implicitly.

POP-NOA: A high number of authors talking about the same class suggests that it is subject to broad discussions. For example, a class frequently mentioned by different users can hide design flaws or stability problems. Also, a class discussed by many developers might be not well-defined, comprehensible or correct, thus more defect prone. For each class, we count the number of different authors that wrote in referring e-mails (*i.e.*, if the same author wrote two, or more, e-mails we count only one).

8.2 Experiments

We conducted our experiments on the software systems depicted in Table 8.2. We considered systems that deal with different domains and have distinct characteristics (*e.g.* popularity, number of classes, e-mails, and defects) to mitigate some of the threats to external validity. These systems are stable projects, under active development, and have a history with several major releases. All are written in Java to ensure that all the code metrics are defined identically for each system.

Public development mailing lists used to discuss technical issues are available for all the systems, and are separated from lists specifically thought for system user issues. We consider e-mails starting from the creation of each mailing list until September 2009. Messages automatically generated by bug tracking and revision control systems are filtered out, and we report the resulting number of e-mails and the number of those referring to classes according to our linking

Table 8.2. Dataset

System URL	Description	#Classes	Mailing lists			Bug Data	
			Creation	#E-Mails		Time period	#Bugs
				Total	Linked		
Equinox eclipse.org/equinox	Plugin system for the Eclipse project	439	Feb '03	5,575	2,383	Feb '03 - Jun '08	1,554
Jackrabbit jackrabbit.apache.org	Implementation of the Content Repository for Java Technology API	1,913	Sep '04	11,901	3,358	Sep '04 - Aug '09	975
Lucene lucene.apache.org	Text search engine library	1,279	Sep '01	17,537	8,800	Oct '01 - Sep '09	1,274
Maven maven.apache.org	Tool for build management of Java projects	301	Nov '02	65,601	4,616	Apr '04 - Aug '09	616

techniques. All systems have public bug tracking systems, that were usually created along with the mailing lists.

8.2.1 Correlations Analysis

To answer the question Q1 “Does the popularity of software components correlate with software defects?”, we compute the correlation between class level popularity metrics and the number of defects per class. We compute the correlation in terms of both the Pearson’s and the Spearman’s correlation coefficient (r_{prs} and r_{spm} , respectively). Contrarily to Pearson’s correlation, Spearman’s one is less sensitive to bias due to outliers and does not require data to be metrically scaled or of normality assumptions [Tri06]. Including the Pearson’s correlation coefficient augments the understanding of the results: If r_{spm} is higher than r_{prs} , we might conclude that the variables are consistently correlated, but not in a linear fashion. If the two coefficients are very similar and different from zero, there is indication of a linear relationship. Finally, if the r_{prs} value is significantly higher than r_{spm} , we can deduce that there are outliers in the dataset. This information first helps us to discover threats to construct validity, then highlights single elements that are heavily related. For example, a high r_{prs} can indicate that, among the classes with the highest number of bugs, we can find also the classes with the highest number of related e-mails.

We compute the correlation between class level source code metrics and number of defects per class, to compare the correlation to a broadly used baseline. We only show the correlation for the source code metric *LOC*, as previous research showed that it is one of the best metrics for defect prediction [GFS05; OW02; OWB04; OWB07]. Table 8.3 shows the correlation coefficients between the different popularity metrics and the number of bugs of each system.

We put in bold the highest values achieved for both r_{spm} and r_{prs} , by system. Results provide evidence that the two metrics are rank correlated, and correlations over 0.4 are considered to be strong in fault prediction studies [ZN08]. The r_{spm} in our study exceed this value for three systems, *i.e.*, Equinox, Lucene, and Maven. In the case of Jackrabbit, the maximum coefficient

Table 8.3. Spearman’s and Pearson’s correlation coefficients between number of defects per class and class level popularity metrics (and *LOC*)

System	POP-NOM		POP-NOCM		POP-NOT		POP-NOTM		POP-NOA		LOC	
	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}	r_{spm}	r_{prs}
Equinox	.52	.51	.52	.42	.53	.54	.52	.48	.53	.50	.73	.80
Jackrabbit	.23	.35	.22	.36	.24	.36	.23	-.02	.23	.34	.27	.54
Lucene	.41	.63	.38	.57	.41	.57	.42	.68	.41	.54	.17	.38
Maven	.44	.81	.39	.78	.46	.78	.44	.81	.45	.78	.55	.78

is 0.24, which is similar to the value reached using *LOC*. The best performing popularity metric depends on the software system: For example in Lucene, *POP-NOTM*, which counts the length of threads containing e-mails about the classes, is the best choice, while *POP-NOT*, number of threads containing at least one e-mail about the classes, is the best performing for other systems.

8.2.2 Defect Prediction

To answer the research question Q2 “*Is a regression model based on the popularity metrics a good predictor for software defects?*”, we create and evaluate regression models in which the independent variables are the class level popularity metrics, while the dependent variable is the number of post-release defects per class. We create regression models based on source code metrics and change metrics alone, as well as models in which these metrics are enriched with popularity metrics, where the dependent variable is always the number of post-release defects per class. We then compare the prediction performances of such models to answer research question Q3 “*Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques?*”

We follow the same methodology used in Chapter 7 and detailed in Section 7.2.2, consisting of: Principal component analysis, building regression models, performing 50 folds cross-validation and evaluating explanative (adjusted R^2) and predictive power (r_{spm}).

Results

Tables 8.4 and 8.5 display the results we obtained for the defect prediction, considering respectively adjusted R^2 values and Spearman’s correlation coefficients. The first row shows the results achieved using all the popularity metrics defined in Section 8.1. In the following four blocks, we report the prediction results obtained through the source code and change metrics, first alone, then by incorporating each single popularity metric, and finally incorporating all the popularity metrics. For each system and block of metrics, when popularity metrics augment the results of other metrics, we put in bold the highest value reached.

Analyzing the results of the sole popularity metrics, we notice that, in terms of correlation, Equinox and Maven still present a strong correlation, *i.e.*, higher than .40, while Lucene is less correlated. The popularity metrics alone are not sufficient for performing predictions in the Jackrabbit system. Looking at the results obtained by using other metrics, we first note that Jackrabbit’s results are much lower if compared to those reached in other systems, especially for the R^2 , and partly for the r_{spm} . Only the r_{spm} reached with change metrics reach good results in this system.

Table 8.4. Defect prediction results: Adjusted R^2

Metrics	Adjusted R^2				
	Equinox	Jackrabbit	Lucene	Maven	Avg
All popularity metrics	.23	.00	.31	.55	.27
All change metrics (MOSER)	.55	.06	.43	.71	.44
MOSER + POP-NOM	.56	.06	.43	.71	.44
MOSER + POP-NOCM	.58	.06	.43	.70	.44
MOSER + POP-NOT	.56	.06	.43	.71	.44
MOSER + POP-NOMT	.56	.06	.43	.70	.44
MOSER + POP-NOA	.56	.06	.43	.70	.44
MOSER + All POP	.61	.06	.45	.71	.46
<i>Improvement</i>	<i>11%</i>	<i>0%</i>	<i>+5%</i>	<i>0%</i>	<i>+4%</i>
OO metrics	.61	.03	.27	.42	.33
OO + POP-NOM	.62	.03	.33	.59	.39
OO + POP-NOCM	.62	.04	.32	.56	.38
OO + POP-NOT	.61	.03	.31	.57	.38
OO + POP-NOMT	.62	.03	.35	.60	.40
OO + POP-NOA	.61	.04	.30	.56	.38
OO + All POP	.62	.03	.37	.61	.41
<i>Improvement</i>	<i>+2%</i>	<i>+25%</i>	<i>+37%</i>	<i>+45%</i>	<i>+27%</i>
CK metrics	.54	.01	.39	.28	.31
CK + POP-NOM	.56	.02	.40	.54	.38
CK + POP-NOCM	.57	.02	.40	.50	.37
CK + POP-NOT	.56	.01	.40	.51	.37
CK + POP-NOMT	.57	.01	.40	.56	.39
CK + POP-NOA	.56	.02	.40	.51	.37
CK + All POP	.57	.02	.42	.58	.40
<i>Improvement</i>	<i>+6%</i>	<i>+50%</i>	<i>+8%</i>	<i>+107%</i>	<i>+43%</i>
CK + OO metrics	.66	.04	.44	.45	.40
CK + OO + POP-NOM	.67	.04	.45	.60	.44
CK + OO + POP-NOCM	.66	.04	.45	.56	.43
CK + OO + POP-NOT	.66	.04	.44	.57	.43
CK + OO + POP-NOMT	.67	.04	.44	.62	.44
CK + OO + POP-NOA	.66	.04	.44	.57	.43
CK + OO + All POP	.67	.04	.46	.63	.45
<i>Improvement</i>	<i>+2%</i>	<i>0%</i>	<i>+5%</i>	<i>+40%</i>	<i>+12%</i>

Going back to the other systems, the adjusted R^2 values are always increased and the best results are achieved when using all popularity metrics together. The increase with respect to the other metrics varies from 2%, when other metrics already reach high values, up to 107%. Spearman's coefficients also increase by using the information given by popularity metrics: Their values augment, on average, more than fifteen percent. However, there is not a single popularity metric that outperforms the others, and their union does not give the best results.

Table 8.5. Defect prediction results: Spearman's correlation coefficient

Metrics	Spearman's correlation r_{spm}				
	Equinox	Jackrabbit	Lucene	Maven	Avg
All popularity metrics	.43	.04	.27	.52	.32
All change metrics (MOSER)	.54	.30	.36	.62	.45
MOSER + POP-NOM	.53	.32	.38	.69	.48
MOSER + POP-NOCM	.57	.31	.43	.60	.48
MOSER + POP-NOT	.54	.31	.39	.59	.46
MOSER + POP-NOMT	.53	.29	.41	.60	.46
MOSER + POP-NOA	.58	.29	.37	.43	.42
MOSER + All POP	.52	.30	.38	.43	.41
<i>Improvement</i>	<i>+7%</i>	<i>+7%</i>	<i>+19%</i>	<i>+11%</i>	<i>+11%</i>
OO metrics	.51	.17	.31	.52	.38
OO + POP-NOM	.53	.14	.35	.52	.38
OO + POP-NOCM	.51	.15	.36	.60	.41
OO + POP-NOT	.49	.15	.38	.52	.38
OO + POP-NOMT	.55	.14	.33	.43	.36
OO + POP-NOA	.53	.12	.38	.70	.43
OO + All POP	.58	.14	.32	.52	.39
<i>Improvement</i>	<i>+14%</i>	<i>-12%</i>	<i>+23%</i>	<i>+35%</i>	<i>+15%</i>
CK metrics	.51	.13	.36	.60	.40
CK + POP-NOM	.48	.13	.35	.69	.41
CK + POP-NOCM	.50	.17	.33	.42	.35
CK + POP-NOT	.53	.13	.34	.52	.38
CK + POP-NOMT	.52	.14	.25	.49	.35
CK + POP-NOA	.52	.14	.41	.53	.40
CK + All POP	.51	.16	.30	.52	.37
<i>Improvement</i>	<i>+4%</i>	<i>+31%</i>	<i>+14%</i>	<i>+15%</i>	<i>+16%</i>
CK + OO metrics	.48	.15	.35	.36	.33
CK + OO + POP-NOM	.59	.15	.34	.62	.43
CK + OO + POP-NOCM	.51	.16	.30	.31	.32
CK + OO + POP-NOT	.50	.14	.35	.52	.38
CK + OO + POP-NOMT	.53	.14	.35	.34	.34
CK + OO + POP-NOA	.51	.15	.34	.43	.36
CK + OO + All POP	.51	.16	.33	.52	.38
<i>Improvement</i>	<i>+23%</i>	<i>+7%</i>	<i>+0%</i>	<i>+72%</i>	<i>+26%</i>

8.3 Discussion

Based on the results presented above, we answer our three research questions.

Q1: Does the popularity of software components in discussions correlate with software defects? Three software systems out of four show a strong rank correlation, *i.e.*, coefficients ranging from .42 to .53, between defects of software components and their popularity in e-mail discussions. Only Jackrabbit is less rank correlated with a coefficient of .23.

Popularity of software components do correlate with software defects.

Q2: Is a regression model based on popularity metrics a good predictor for software defects? In the second part of our results, consistently with the correlation analysis, the quality of predictions on Jackrabbit using popularity metrics are extremely low, both for the adjusted R^2 values and for the Spearman's correlation coefficients. On the contrary, our popularity metrics applied to the other three systems lead to different results: Popularity metrics are able to predict defects. However, if used alone, they do not compete with the results obtained through other metrics. The best average results are shown by the change metrics, corroborating previous research stating the quality of such predictors [MPS08; BEP07].

Popularity can predict software defects, but without major improvements over previous established techniques.

Q3: Does the addition of popularity metrics improve the prediction performance of existing defect prediction techniques? We obtained the best results by integrating the popularity information into other techniques. This creates more reliable and complete predictors that significantly increase the overall results: The improvements on correlation coefficients are, on average, more than fifteen percent higher, with peaks over 30% and reaching the top value of 72%, to those obtained without popularity metrics. This corroborates our initial assumption that popularity metrics measure an aspect of the development process that is different from those captured by other techniques.

Results put in evidence that, given the considerable difference of the prediction performance across different software projects, bug prediction techniques that exploit popularity metrics should not be applied in a “black box” way. As suggested by Nagappan *et al.* [NBZ06], the prediction approach should be first validated on the history of a software project, to see which metrics work best for predictions.

Popularity metrics do improve prediction performances of existing defect prediction techniques.

8.4 Threats to validity

Threats to construct validity. We already considered some of these threats in Section 3.2.3, when discussing the limitations of populating Mevo models.

Another threat concerns the procedure for linking e-mails to discussed classes. We use linking techniques whose effectiveness was measured [BDLR09], and it is known that they cannot produce a perfect linking. The enriched Mevo model can contain wrongly reported links or miss connections that are present. We alleviated this problem by manually inspecting all the classes that showed an exceptional number of links (*i.e.*, outliers) and, whenever necessary, adjusted the regular expressions composing the linking techniques to correctly handle such unexpected situations. We removed from our dataset any e-mail automatically generated by the bug tracking system and the revision control system, because they could bias the results.

Threats to statistical conclusion validity. In our experiments all the Spearman's correlation coefficients and all the regression models were significant at the 99% level.

Threats to external validity. We analyzed only open-source software projects, however the development in an industrial environment may differ and conduct to different compartments in the developers, thus to different results. Another external validity threat concerns the language:

All the software systems are developed in Java. Although this alleviates parsing bias [KSS02], communities using other languages could have different developer cultures and the style of e-mails can vary.

8.5 Related work

We already surveyed related work in the area of defect prediction in Section 2.5.2. Here we present previous approaches that mine data from e-mail archives.

Li *et al.* first introduced the idea of using the information stored in mailing lists as an additional predictor for finding defects in software systems [LHS05]. They conducted a case study on a single system, used a number of previously known predictors and defined new mailing list predictors. Mainly such predictors counted the number of messages to different mailing lists during the development of software releases. One predictor, called *TechMailing* and based on number of messages to the technical mailing list during development, was found to be the most highly rank correlated with the number of defects, among all the predictors evaluated. Our work differs in genre and granularity of defects we predict: We focus on defects on small source code units that can be easily reviewed, analyzed, and improved. Moreover, Li *et al.* did not remove the noise from the mailing lists, focusing only on source code related messages.

Pattison *et al.* were the first to introduce the idea of studying software entity (function, class, *etc.*) names in e-mails [PBD08]. They used a linking technique based on simple name matching, and found a high correlation between the amount of discussions about entities and the number of changes in the source code. However, Pattison *et al.* did not validate the quality of their links between e-mails and source code. Our work was the first to measure the effectiveness of linking techniques for e-mails and source code [BDLR09].

To our knowledge, this research is the first work that uses information from development mailing lists at class granularity to predict and to find correlation with source code defects. Other works also analyzed development mailing lists but extracting a different kind of information: social structures [BGD⁺06], developers participation [MFH02], inter-projects migration [BGD⁺07], and emotional content [RH07].

8.6 Summary

We extended our Mevo meta-model to describe e-mail information. Based on the extended meta-model, we devised a novel approach to correlate popularity of source code artifacts within e-mail archives to software defects. We also investigated whether such metrics could be used to predict post-release defects. We showed that, while there is a significant correlation, popularity metrics by themselves do not outperform source code and change metrics in terms of prediction power. However, we demonstrated that, in conjunction with source code and change metrics, popularity metrics increase both the explanative and predictive power of existing defect prediction techniques.

The focus of this chapter was popularity metrics extracted from e-mail discussions: We investigated whether they correlate with software defects and if they can improve previous defect prediction techniques. In the following chapter, we conduct a similar analysis, but concentrating on a different type of information: We study the relationship between change coupling and software defects.

Chapter 9

On the Relationship Between Change Coupling and Software Defects

In Chapter 4 we presented the Evolution Radar, a visual approach to analyze change coupling information. Change coupling is the implicit and evolutionary dependency between two, or more, software artifacts that, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view. Through two case studies, we showed that change coupling information can be used to pinpoint architectural design issues in a software system.

Also other researchers observed that change coupling is a bad symptom in a software system: At a fine grained level, because a developer who changes an entity might forget to change related entities [ZWDZ05; BZ06], or—at the system level—because high change coupling among modules points to design issues such as architecture decay [GHJ98; GJK03; PGFL05].

All the mentioned approaches assume that change coupling, indeed, is a cause of issues in a software system. However, the relationship between change coupling and a tangible effect of software issues was not studied yet. To perform such an investigation, one needs an objective quantification of the issues that affect a software system and its components. A defect repository, which records all the known issues about a software system, provides such a quantification. Eaddy *et al.* performed a similar study linking cross-cutting concerns with defects [EZS⁺08], but the specific case of change coupling remains unaddressed.

We define several measures of change coupling and analyze their correlations with software defects, using as baseline the correlation between source code metrics and software defects. We provide empirical evidence, through three case studies, that the defined change coupling metrics correlate with defects, and investigate the relationships of such metrics with defects based on the severity of the reported bugs. Finally, we study whether the performance of defect prediction models, based on source code metrics, can be improved with change coupling information.

Structure of the chapter. We define change coupling measures in Section 9.1 and describe our dataset in Section 9.2. We analyze the correlation of the defined coupling measures with software defects in Section 9.3. In Section 9.4 we discuss how to enrich defect prediction models with change coupling information and which improvements they provide. We address the threats to validity in Section 9.5 and conclude in Section 9.6.

9.1 Measuring Change Coupling

The Mevo meta-model includes all the pieces of information needed to perform our study: Versioning system data to extract change coupling dependencies, source code information to compute code metrics, and bug data to count the number of defects per software entity. However, to quantify the correlation of change coupling with software defects, we need to measure the change coupling. It is the implicit dependency of software artifacts that have been observed to frequently change together during the evolution of a software system. The more they changed together, the stronger the change coupling dependency is. Still, there is no consensus on the formal definition of change coupling, and several alternative measures exist. We formally define four measures of change coupling emphasizing different aspects.

We need change coupling measures that are defined for each class in the system. We decided to carry out our analysis at the class level because classes are a cornerstone of the object-oriented paradigm, and we want to be able to compare change coupling with object oriented metrics.

The measures we define concern the coupling of a class with the entire system. An alternative is a measure of change coupling for each pair of entities in the system. However, since bugs are often mapped to one entity only, we opted for a coupling measure involving one entity only. We can define a measure of coupling of a class with the entire system simply by aggregating the pairwise coupling measures.

In the following definitions we use the concept of *n-coupled* classes. We define two classes as *n-coupled* if there are at least *n* transactions that include both the classes. As a consequence, all our change coupling measures are functions of *n*. Given two classes c_1 and c_2 , we consider them *n-coupled* if the following condition holds:

$$|\{t \in T | c_1 \in t \wedge c_2 \in t\}| \geq n \quad (9.1)$$

where T is the set of all the transactions. Given a class c , we define the *set of coupled classes* (SCC) as:

$$SCC(c, n) = \{c_i | c_i \neq c \wedge c \text{ is } n\text{-coupled with } c_i\} \quad (9.2)$$

Figure 9.1 shows an example scenario with five classes and six transactions. In this case, $SCC(c2, 3) = \{c1, c5\}$, $SCC(c2, 4) = \{c1, c5\}$ and $SCC(c2, 5) = \{c1\}$.

Number of Coupled Classes (NOCC)

The first per-class measure of change coupling is the number of classes *n-coupled* with a given class c . This measure emphasizes the raw number of classes with which a given class is coupled with. We define *NOCC* as:

$$NOCC(c, n) = |SCC(c, n)| \quad (9.3)$$

The *NOCC* measure is the cardinality of the set of coupled classes. In the example in Figure 9.1 $NOCC(c2, 3) = 2$.

Sum of Coupling (SOC)

The sum of coupling is the sum of the shared transactions between a given class c and all the classes *n-coupled* with c . We define *SOC* as:

$$SOC(c, n) = \sum_{c_i \in SCC(c, n)} |\{t \in T | c_i \in t \wedge c \in t\}| \quad (9.4)$$

The *SOC* measure is the sum of the cardinalities of the sets of transactions that include the class c and the classes n -coupled with c . Compared to *NOCC*, *SOC* also takes into account the strength of the couplings. In Figure 9.1 $SOC(c2, 3) = 4 + 5 = 9$.

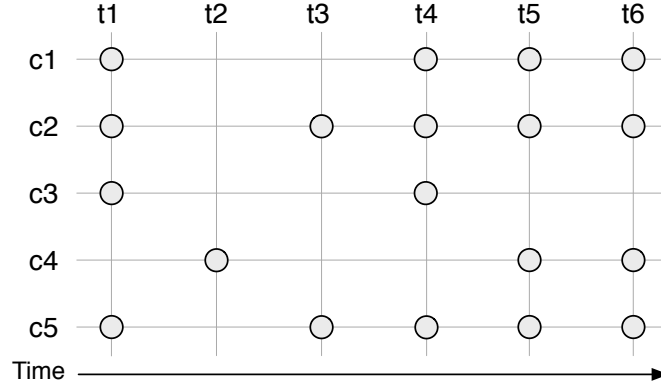


Figure 9.1. Sample scenario of classes and transactions

Exponentially Weighted Sum of Coupling (*EWSOC*)

EWSOC is a variation of *SOC*, where the shared transactions are exponentially weighted according to their distance in time: Recent changes are emphasized over past changes, following an exponential decay model. We define *EWSOC* as:

$$EWSOC(c, n) = \sum_{c_i \in SCC(c, n)} EWC(c_i, c), \text{ where} \quad (9.5)$$

$$EWC(c_i, c) = \sum_{t_k \in T(c)} \begin{cases} 0 & \text{if } c_i \notin t_k \\ \frac{1}{2^{|T(c)|-k}} & \text{if } c_i \in t_k \end{cases} \quad (9.6)$$

$T(c)$ is the set of all the transactions, sorted by time, that include the class c . Figure 9.2 shows an example of *EWSOC* computation for the class $c2$ with $n = 3$. In this case, $T(c2) = \{t1, t3, t4, t5, t6\}$, $|T(c2)| = 5$ ($t2$ is not included in the computation since c is absent in it), and therefore:

$$EWSOC(c2, 3) = EWC(c2, c1) = \frac{1}{2^{5-5}} + \frac{1}{2^{5-4}} + \frac{1}{2^{5-3}} + 0 + \frac{1}{2^{5-1}}$$

Linearly Weighted Sum of Coupling (*LWSOC*)

The last per-class measure of change coupling is another variation of *SOC*, in which the shared transactions are linearly weighted according to their distance in time. As *EWSOC*, *LWSOC* emphasizes recent changes, but following a linear decay model, and thus penalizing past changes less than *EWSOC*. We define *LWSOC* as:

$$LWSOC(c, n) = \sum_{c_i \in SCC(c, n)} LWC(c_i, c), \text{ where} \tag{9.7}$$

$$LWC(c_i, c) = \sum_{t_k \in T(c)} \begin{cases} 0 & \text{if } c_i \notin t_k \\ \frac{1}{|T(c)|+1-k} & \text{if } c_i \in t_k \end{cases} \tag{9.8}$$

In Figure 9.2, $LWSOC(c2, 3)$ is equal to $LWC(c2, c1)$

$$LWSOC(c2, 3) = LWC(c2, c1) = \frac{1}{6-5} + \frac{1}{6-4} + \frac{1}{6-3} + 0 + \frac{1}{6-1}$$

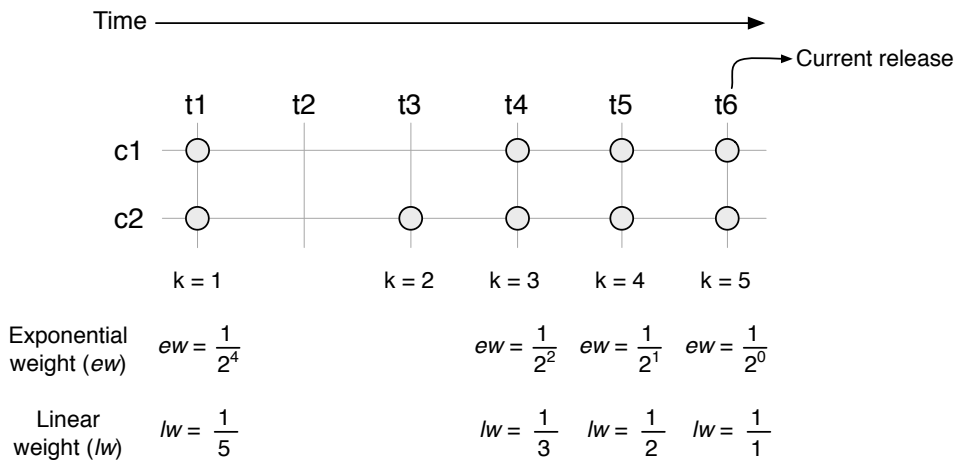


Figure 9.2. Example EWSOC and LWSOC computations

Common Behaviors and Differences

All the measures are defined on a class-by-class level, and aggregated to recover a measure of the coupling of one class with the entire system. All the defined metrics decrease if n increases, as the set of coupled classes at the value n shrinks if n increases. We compute the coupling measures by means of our Pendolino tool.

Beyond that, these four measures emphasize different aspects of change coupling: *NOCC* measures only the number of co-change occurrences of a class with all the other classes that exceed the threshold n . On the other hand, *SOC* takes into account the magnitude of each coupling relationship beyond the threshold, so that a pair of classes changing extremely often together is weighted differently. *EWSOC* and *LWSOC* function similarly, but consider the recency of the co-change relationships. A reason for this is that two classes may have been co-changed heavily in the past, but then have been refactored to not depend on each other: Their past behavior should not affect their current coupling value. *EWSOC* discounts the past more quickly than *LWSOC* does.

9.2 Case Studies

To study the relationship between change coupling and software defects we analyzed three large Java software systems: ArgoUML, Eclipse JDT Core, and Mylyn. Table 9.1 shows their size in terms of classes and transactions.

Table 9.1. Measures of the studied software systems

System url	Description	#Classes	#Transactions	#Transactions per class (avg)	#Shared transactions per class (avg)
ArgoUML http://argouml.tigris.org	UML modeling tool	2,197	15,257	14.30	0.37
Eclipse JDT Core www.eclipse.org/jdt/core/	Java Infrastructure of the Java IDE	1,193	13,186	68.00	5.30
Mylyn www.eclipse.org/mylyn/	Task management framework for Eclipse	3,050	9,373	11.70	0.39

In computing the change coupling measures, one problem that we encountered concerns large commits, *i.e.*, transactions involving a large number of artifacts, typically license updates. These transactions involve totally unrelated artifacts, and thus might alter change coupling measures. We already discussed the problem in Section 3.2.3, when describing how we populate Mevo models, concluding that the solution is to filter out large commits. In our experiments we filtered out all transactions involving more than 100 classes, which were 86 for ArgoUML (0.6%), 59 for Eclipse JDT Core (0.4%), and 102 for Mylyn (1.1%). We manually inspected the commit comments of these transactions, the vast majority of which concerned license changes, Javadoc and documentation updates.

9.3 Correlation Analysis

The goal of the correlation analysis is to answer the following questions:

1. Does change coupling correlate with software defects? If so, which change coupling measure correlates best?
2. Does change coupling correlate more with severe defects than with minor ones?

We compute the values of the correlation between the number of defects per class (or number of defects with a given severity) and the various measures of change coupling. We quantify the correlation in terms of the Spearman's correlation coefficient, which is less sensitive to bias due to outliers and recommended with skewed data. In computing the correlation we consider all the defects in the history of the systems.

To make a comparison with a broadly used baseline, we also compute the correlation between the number of defects per class and the following metrics: The Chidamber & Kemerer object oriented metrics suite (listed in Table 7.3), a selection of other object-oriented metrics (NOA: Number Of Attributes, NOM: Number Of Methods, Fan in, Fan out, LOC: Lines Of Code) and the number of changes to a class (Changes).

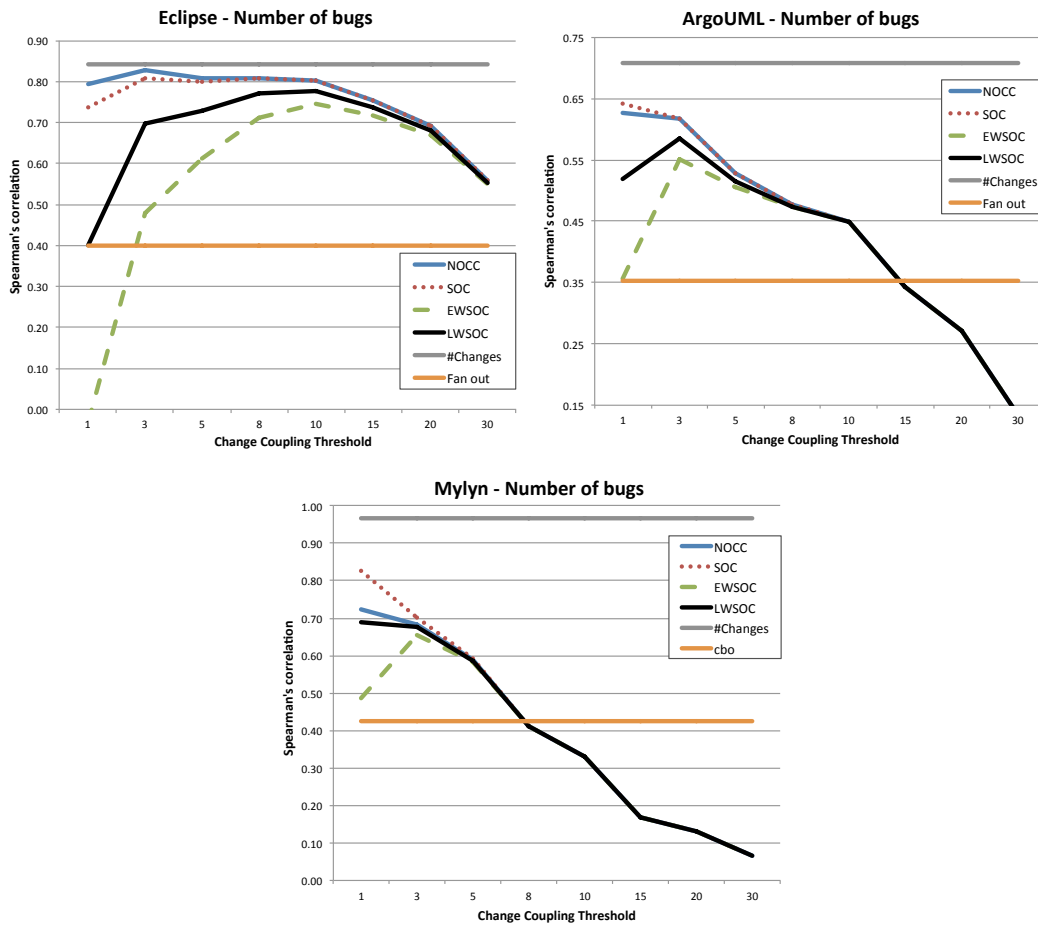
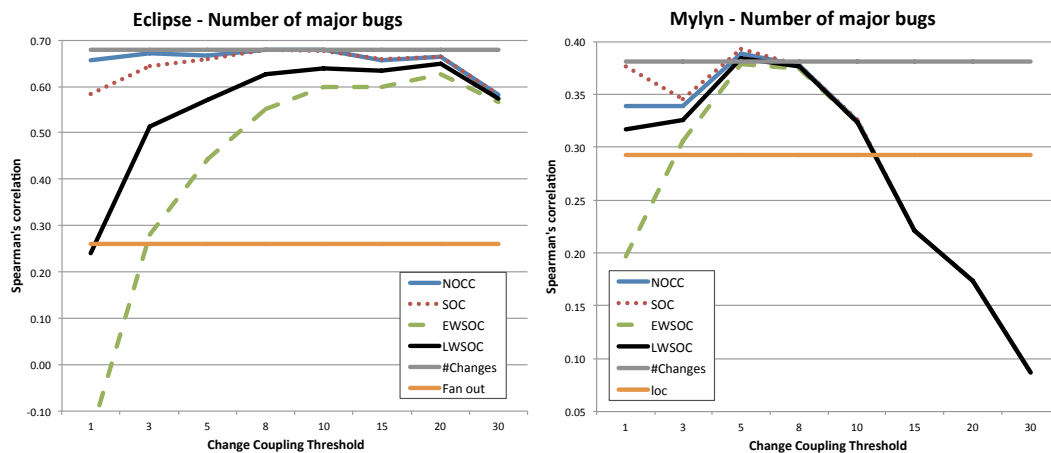


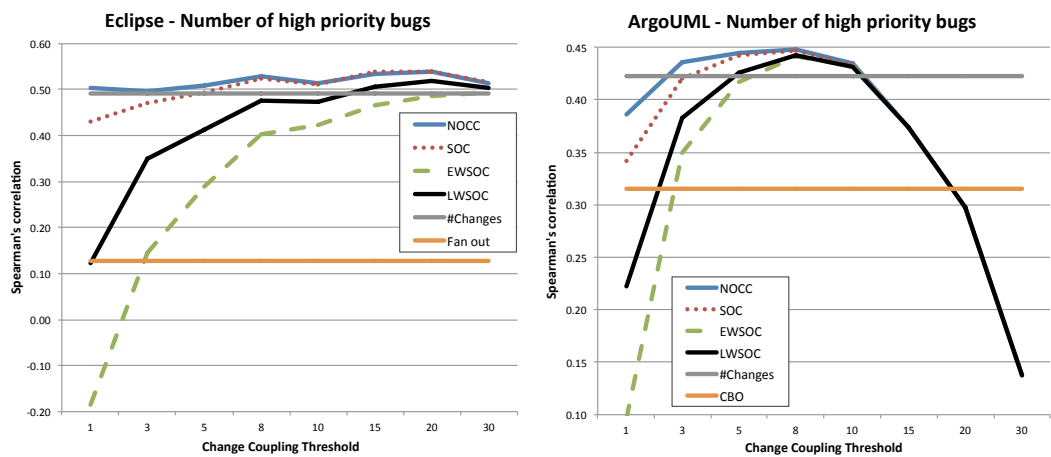
Figure 9.3. Correlations between number of defects per class and class level change coupling measures, number of changes, and the best object-oriented metric (*i.e.*, Fan out for Eclipse and ArgoUML, and CBO for Mylyn). The correlations are measured with the Spearman's coefficient.

9.3.1 Results

Figure 9.3 and Figure 9.4 show the Spearman's correlation of the number of defects with the metrics we tested across the three case studies. Figure 9.3 displays the correlation for all levels, while Figure 9.4 shows it for selected categories of bugs, according to their labels in the bug tracking system (major bugs and high priority bugs). All the graphs follow the same format: The Spearman's correlation is indicated on the y axis, while the x axis indicates the threshold used for the computation of change couplings metrics (*i.e.*, the value of n used as a basis to compute n -coupled classes). For example, all the change coupling measures at the x position of 3 are computed using the set of 3-coupled classes. Metrics that do not depend on this threshold (such as Changes, Fan out, or CBO) are hence flat lines. The metrics on each graph are the four coupling measures (NOCC, SOC, EWSOC, LWSOC), the number of changes metric, and the best performing among the object-oriented metrics for each project and each bug category.



(a) Major bugs



(b) Bugs with high priority

Figure 9.4. Spearman's correlations between number of major/high priority bugs and change coupling measures

Correlation with all types of bugs. Figure 9.3 shows the correlation of metrics with all types of bugs, for all systems. The best performing object-oriented metrics are: Fan out for Eclipse and ArgoUML, and CBO for Mylyn. In all the software systems change coupling indeed correlates with the number of defects, since the Spearman's coefficient reaches values above 0.5, especially for Eclipse where the maximum Spearman's is above 0.8. The SOC measure is the best for ArgoUML and Mylyn, and the second best for Eclipse. All the coupling measures decrease after a certain value of n : 3 for ArgoUML and Mylyn, 10 for Eclipse. EWSOC and LWSOC do not correlate for low values of n , while they are comparable with NOCC and SOC for $n \geq 3$ in ArgoUML and Mylyn, $n \geq 10$ for Eclipse.

Correlation with major bugs. Figure 9.4(a) shows the Spearman's correlation between the number of major bugs and change coupling measures. We consider a bug as major if its severity is major, critical or blocker. We also show the correlations with number of changes and the best object-oriented metric: Fan out for Eclipse and *LOC* for Mylyn. For Eclipse, with $3 \leq n \leq 20$ *NOCC* and *SOC* are very close to number of changes (about 0.7). *EWSOC* and *LWSOC* have bad performances with $n < 10$, while starting from 10 they are above 0.6. In the case of Mylyn the correlations are lower, with a maximum of circa 0.4. For $n = 5$ all the change coupling measures are at the maximum and above number of changes, and for $n > 8$ they rapidly decrease. We do not show the result for ArgoUML because the number of major bugs is not large enough to obtain significant correlations.

Correlation with high priority bugs. Figure 9.4(b) shows the Spearman's correlation for the number of high priority bugs, *i.e.*, bugs having priorities above the default value (P3). This time, the best object-oriented metrics are Fan out for Eclipse and *CBO* for ArgoUML. For this particular type of bugs, the correlations are weaker, with a maximum around 0.55 for Eclipse and 0.45 for ArgoUML. The change coupling measures are often better than the number of changes. In the case of Eclipse, *NOCC* is always better, *SOC* is better for $n \geq 5$ and *LWSOC* for $n \geq 15$, while *EWSOC* is always worse. For ArgoUML all the change coupling measures have a maximum for $n = 8$, which is greater than the correlation of the number of changes. After that, for $n > 8$ the correlations rapidly decrease. We do not show the result for Mylyn because the number of high priority bugs is not large enough to obtain significant correlations.

9.3.2 Discussion

Based on the data presented in Figure 9.3 and Figure 9.4, we derive the following insights.

Change coupling works better than source code metrics. From Figure 9.3 we see that, for every system, there is a range of values of n in which change coupling measures indeed correlate with number of defects. They correlate more than the *CK* and other object-oriented metrics, but less than the number of changes. The fact that number of changes correlates with number of defects was already assessed by previous research [NB05b; MPS08]. One possible reason why the number of changes correlates more is that this information is defined for every class in the system, while only some classes have change coupling measures greater than 0. This also explains why change coupling measures peak at a given index and then decrease in accuracy, as very few classes have a change history large enough to exceed moderately high thresholds of co-change. Further, since not all the bugs are related to a change coupling relationship, all in all the number of changes have a higher correlation with defects. Similar to this situation, Gyimóthy *et al.* found that *LOC* is among the best metrics to predict defects [GFS05], since it is defined for all entities in the system. In conclusion, we can answer the first question:

Change coupling correlates with defects, more than source code metrics but less than number of changes.

Change proneness plays a role. Another observable fact in Figure 9.3 is that for ArgoUML and Mylyn the correlation of the change coupling measures rapidly decreases with $n \geq 5$, while for Eclipse this happens with $n \geq 20$. The reason behind this is that Eclipse classes have, on average, many more changes and more shared transactions than classes in the other two systems. In

Eclipse the average number of changes per class is 68, while in ArgoUML it is 14.3 and in Mylyn 11.7. The average number of shared transactions per class is 5.3 for Eclipse, 0.37 for ArgoUML and 0.39 for Mylyn. Since we consider three systems, we cannot derive a general formula, but limit ourselves to note that the correlation depends on the change proneness of the system. In short the insight is the following:

The correlation between change coupling measures and defects varies with n. The trend and the maximum correlation values depend on the software system and, in particular, on its change proneness.

Change coupling is harmful. The situation in Figure 9.4 is different from the one in Figure 9.3. The average value of the Spearman's correlation is lower when considering only major or high priority bugs than with all the bugs. This is not surprising, since there is a smaller amount of data and therefore the correlation is less precise. The interesting fact here is the delta between the number of changes and the change coupling measures: It is lower for major and high priority bugs, with respect to all the bugs, and it is often negative, *i.e.*, change coupling measures correlate with number of major/high priority bugs more than number of changes. One possible explanation is that change coupling can be detected only in the evolution of a system. As such, this type of dependency is often hidden and might be related to bugs with a high priority or a high severity. The answer to the second question is then:

On average the correlation between change coupling measures and number of major/high priority bugs is lower than with all the bugs. For these particular bugs change coupling measures are always better than code metrics and, in many cases, than number of changes.

Sometimes it is better *not* to forget the past. One last observation from both Figure 9.3 and Figure 9.4 is that the correlation for *EWSOC* is always below the one for *LWSOC*, and the latter one is always below *NOCC* and *SOC*. From this we infer that “penalizing” couplings in the past does not work in correlating with number of defects, *i.e.*, couplings in the past also correlate with defects. *EWSOC*, which penalizes the past more than *LWSOC*, correlates less with defects. The second part of the answer to the first question is:

“Penalizing” change coupling in the past decreases the correlation with number of defects. The best change coupling metrics are then NOCC and SOC.

9.4 Regression Analysis

The goal of the regression analysis is to answer the following questions:

1. Does the use of change coupling information improve explanative and predictive powers of bug prediction models based on software metrics?
2. Is the improvement greater for severe bugs?

To answer these questions, we create and evaluate different regression models in which the independent variables (for predicting) are respectively code metrics, change coupling measures, number of changes and their combinations, while the dependent variable (the predicted one) is the number of bugs, the number of major bugs and the number of high priority bugs.

Once again, we follow the methodology proposed by Nagappan *et al.* [NBZ06] that we already used in Chapters 7 and 8. The methodology, detailed in Section 7.2.2, consists in the following steps: Principal component analysis, building regression models, evaluating explanative power and evaluating predictive power.

However, in the previous two chapters we predicted the number of post-release defects, emulating a real-life scenario. In the following experiments, we consider all the bugs, building regression models from 90% of the classes (training set) and evaluating them on the remaining 10% (validation set). Even if these settings do not emulate a real-life scenario, we opt for them for two reasons: First, our goal is comparing regression models to inspect whether change coupling information can improve established defect prediction approaches. Applying the models on the same dataset, although not emulating real-life settings, achieve our goal. Second, since we want to study the impact of bug severity on the prediction, we have to perform experiments considering major or high priority bugs only. Unfortunately, in our dataset, these types of bugs in a post release period (six months) are not enough to produce significant correlations. Therefore, we consider the entire history.

9.4.1 Results

Figure 9.5 shows the results of our experiments for Eclipse in terms of explanative power (R^2) and predictive power (Spearman's correlation). We show the results for the regression models built using the following sets of variables: (1) source code metrics, (2) code metrics and number of changes, (3) code metrics and *NOCC*, (4) code metrics and *SOC*, (5) code metrics and *EWSOC*, (6) code metrics and *LWSOC*, (7) all *NOCC*, *i.e.*, the *NOCC* metrics for each value of n and (8) all *CC* measures, *i.e.*, all the measures of change coupling for each value of n . We show the results only for Eclipse and only for major bugs, since the results for the high priority bugs and for the other two systems (ArgoUML and Mylyn) are on the same line with the ones presented in Figure 9.5. We do not show the adjusted R^2 values, since it tends to remain comparable to R^2 .

9.4.2 Discussion

Regression models based on source code metrics and change coupling information have a greater explanative and predictive power than models based only on code metrics. However, the model based on code metrics and number of changes has a slightly better prediction power than “all *CC* measures” and “*NOCC* all”, and a slightly worse explanative power than “all *CC* measures”. This answers our first question:

Using change coupling information improves explanative and predictive powers of bug prediction models based on source code metrics. However, it does not yield a significant improvement over models based on code metrics and number of changes.

When considering only major bugs, the overall performance is lower, but the models based on change coupling are better than the one based on number of changes. The model based on “all *CC* measures” is the best in terms of explanative power, but it also suffers from overfitting, since its prediction performance is much lower. On the other hand, the model based on “*NOCC* all” is the best in terms of prediction (slightly better than number of changes), but not in terms of R^2 . We can answer our second question:

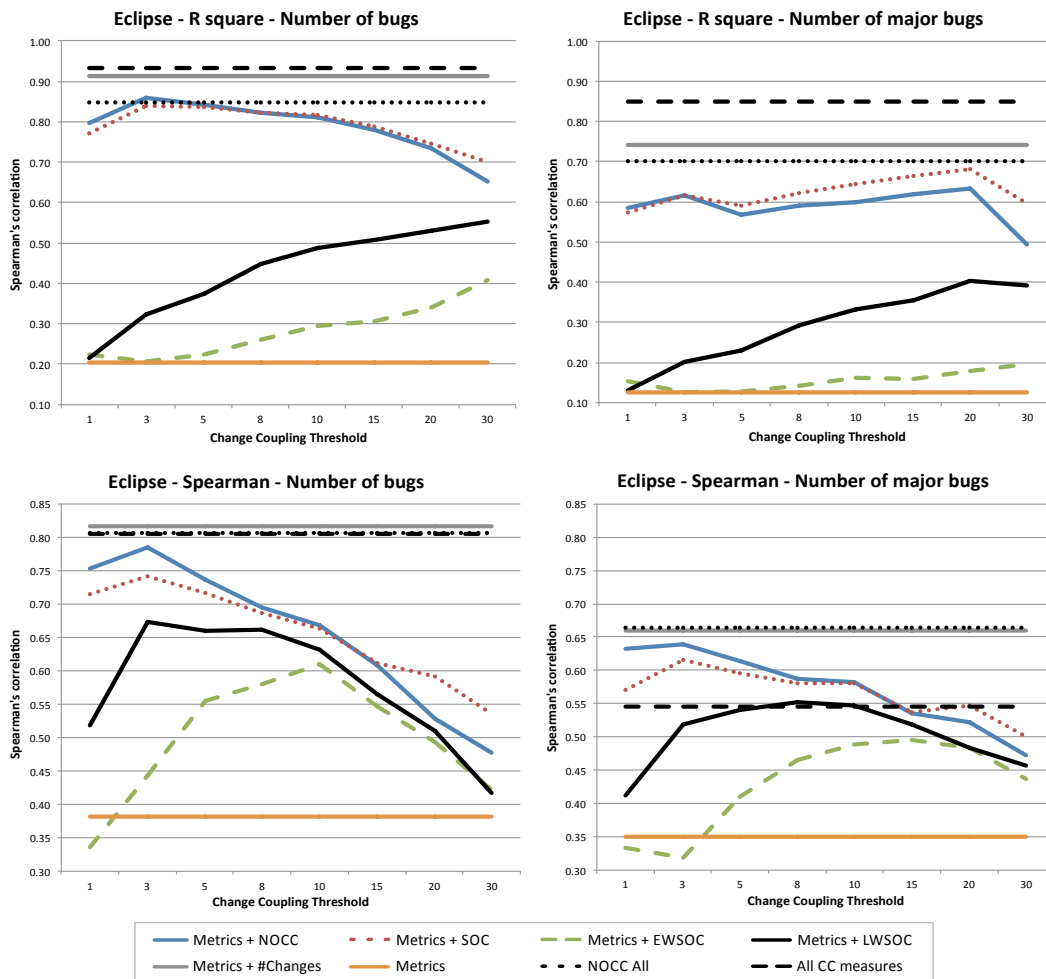


Figure 9.5. Results of the regression analysis for Eclipse

Predicting severe bugs is hard, as the performance of prediction models are, on average, lower than the ones for all the bugs. Models based on change coupling measures yield better results than models based on code metrics, and slightly better results than the one based on code metrics and number of changes.

The conclusions drawn for the correlation analysis are still valid for the regression. First, it is better not to forget the past, *i.e.*, change coupling measures that penalize past coupling relationships (*EWSOC* and *LWSOC*) have bad explanative and prediction power. Second, change proneness plays a role: The change coupling measures have different trends for different systems. This is because different systems have different average numbers of transactions and shared transactions per class. We do not show the regression results for ArgoUML and Mylyn, but the trends of *NOCC*, *SOC*, *EWSOC* and *LWSOC* are similar to the ones presented in Figure 9.3 for the correlation analysis.

9.5 Threats to Validity

Threats to construct validity. We already presented some of these threats in Section 3.2.3, when describing the limitations of populating Mevo models. Another threat concerns large commits: We discussed in Section 9.2 how to mitigate it.

Threats to statistical conclusion validity. In our experiments all the Spearman's correlation coefficients and all the regression models were significant at the 99% level.

Threats to external validity. In our experiments there are three threats belonging to this category: First, we analyzed only three software systems; second, they are all open-source; third, they are all developed in Java.

9.6 Summary

Change coupling has long been considered a significant issue. However, no empirical study of its correlation with actual software defects had been done until now. By exploiting the data residing in Mevo, we performed such a study on three large software systems and found that there was indeed a correlation between change coupling and defects: The correlation is higher than the one observed with source code metrics. Further, defects with a high severity seem to exhibit a correlation with change coupling that, in some instances, is higher than the one with the change rate of the components. We also enriched bug prediction models based on code metrics with change coupling information, and the results—in terms of explanative and predictive power—corroborate our previous findings.

In this chapter we empirically investigated the (negative) impact of change coupling on software quality, measured with number of defects. In the next chapter, we perform an analogous study on design flaws to determine their impact on software defects.

Chapter 10

On the Relationship Between Design Flaws and Software Defects

Over the last years, researchers proposed a variety of approaches to detect source code fragments that are hard to understand, change or maintain. Source code entities that have design flaws are good candidates, since flaws are known to have a negative impact on quality attributes [GHJV95]. As a last application of our approach, we want to empirically assess such negative impact. To this aim, we need to measure software quality and to identify design flaws in the source code. For the former, we apply the same technique employed in Chapter 9, *i.e.*, we quantify (negative) software quality with number of defects. To identify design flaws, simple source code metrics are insufficient, because they must be considered and analyzed in the context in which they appear. For this reason, *meaningful* metric combinations were devised as so-called *detection strategies* [Mar04] and put in the context of *design (dis)harmony* [LM06].

Researchers thoroughly analyzed design disharmonies: To find good metrics and thresholds for their classification [Mar04; LM06; SLT06], to propose correction strategies and refactorings [TSG04; LM06], to visualize them [WL08b], and to put them in relation to code evolvability [ML06] or change-proneness [KPG09]. Still, the relationship between design flaws and software defects was not investigated.

By analyzing the data residing in our Churrasco framework, we study this relationship, conducting an extensive experiment on six open-source software systems. First, we examine the frequency of design flaws in the systems. Then, we analyze the correlation of flaws with post-release defects. The fact that Mevo models multiple source code versions allows us to study also the evolution of flaws over time: In particular, we evaluate whether adding flaws to a software entity will induce bugs in the future.

We conducted our experiments not only analyzing each flaw, *per se*, but also extracting and comparing differences between flaws in all the mentioned situations.

Structure of the chapter. In Section 10.1 we introduce detection strategies, the technique we employ to identify design flaws in software systems. In Section 10.2 we describe our dataset and experimental setup. We present our set of experiments and discuss their results in Section 10.3. Before concluding in Section 10.5, we outline the threats to the validity of this study in Section 10.4.

10.1 Design Flaws and Detection Strategies

As opposed to object-oriented metrics [CK94], which are simple measures of size (e.g., lines of code, number of methods) or complexity (e.g., McCabe cyclomatic complexity) of software, *detection strategies* [Mar04] provide a formal method to identify design flaws in a given source code fragment, also referred to in the literature as “code smells” [FBB⁺99].

To recognize a number of design flaws, we transform informal design rules, guidelines, and heuristics [GHJV95; Rie96; FBB⁺99] into detection strategies, *i.e.*, metrics-based logical conditions that detect violations against design guidelines. We use the catalog of design flaws described by Lanza and Marinescu [LM06]. We illustrate how we translate informal design rules into a detection strategy to identify the design flaw called *Brain Method*, and refer the reader to [LM06] for details about the other detection strategies.

Example

The Brain Method design flaw refers to a method that tends to centralize the functionality of a class, in the same way a God Class [Rie96] centralizes the functionality of an entire (sub)system. It can be informally described by the following rules: (1) It is excessively large, (2) it has many conditional branches, computed using the McCabe’s cyclomatic complexity, (3) it has a deep nesting level, and (4) it uses many¹ variables. These rules can be transformed into the detection strategy depicted in Figure 10.1.

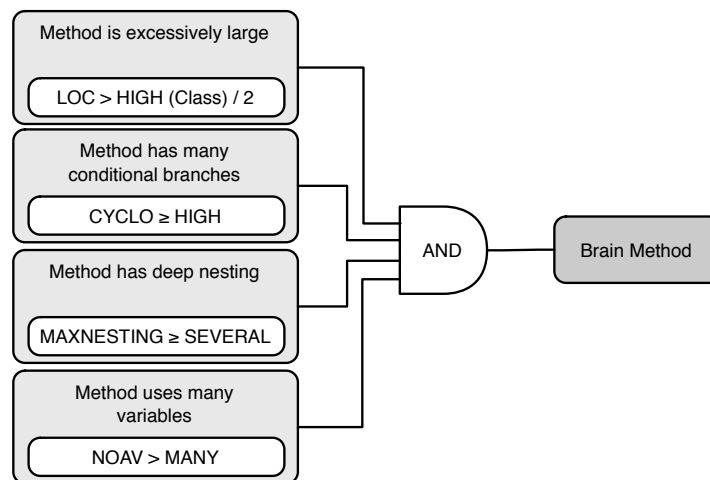


Figure 10.1. The Brain Method detection strategy

Filtering conditions are expressed in terms of metrics (the left part of the expressions) and related to thresholds (the right part of the expressions). Lanza and Marinescu computed the values of the thresholds by measuring 45 Java systems of different sizes and from different domains [LM06].

¹“Many” refers to a number higher than a human can keep in short-term memory [Pin97], *i.e.*, 6 - 9.

CYCLO—also known as McCabe’s Cyclomatic Complexity—is the number of linearly-independent paths through an operation. *MAXNESTING* represents the maximum nesting level of control structures within an operation. *NOAV* is the total number of variables directly accessed from the measured operation. Variables include parameters, local variables, instance variables and global variables. “*HIGH*” refers to a threshold for methods, while “*HIGH (Class)*” refers to a threshold for classes.

This detection strategy, functioning on any Java software system, produces a set of candidate methods exhibiting symptoms of the Brain Method design flaw. For example, in one version of Lucene we were able to detect 120 brain methods.

In addition to Brain Method, we consider the following method level design flaws:

1. *Feature Envy*: Methods more interested in the data of other classes than that of their own class [FBB⁺99], by accessing it directly or via accessors.
2. *Intensive Coupling*: Methods intensively coupled to other methods located in few other classes. The communication between the client method and (at least one of) its provider classes is excessively verbose.
3. *Dispersed Coupling* is complementary to the previous design flaw, and it refers to a method excessively tied to many other methods in the system dispersed among many classes. A single method communicates with an excessive number of classes, whereby the communication with each of the classes is not intense.
4. *Shotgun Surgery* denotes that a change in a single method implies many changes to many different methods and classes [FBB⁺99]. This design flaw deals with strong afferent (incoming) coupling, thus concerning the coupling strength and dispersion.

10.2 Experimental Setup

We perform our experiments on the six Java systems listed in Table 10.1. For each system, we create a Mevo model including versioning system information, bug data and bi-weekly source code snapshots as FAMIX models. Once created all the FAMIX models—one every two weeks—we employ detection strategies on them to identify design flaws. The result is a list of method level design flaws that each class contains. We conduct our study at the class level since classes are the cornerstone of the object-oriented paradigm, and developers perform maintenance and refactoring tasks mostly at this level.

Table 10.1. Software systems used for the experiments

System	Description
Lucene	High-performance, full-featured text search engine library.
Maven	Tool for build automation and management of Java projects.
Mina	Network application framework.
Eclipse CDT	C/C++ Integrated Development Environment (IDE) for Eclipse.
Eclipse PDE UI	Models, builders and editors to facilitate plug-in development in Eclipse.
Equinox	Plugin system for the Eclipse project.

For each system, Table 10.2 shows the period of time considered,² the number of versions included in the model, the size of the systems in terms of average number of classes, average number of design flaws and total number of bug references reported in the considered time period. Since the number of classes and the number of design flaws vary across system versions, we show average numbers over all the considered versions. Bug references indicate all the links to classes that a bug can have: When a single bug is linked with multiple classes, the number of bugs is one, while the number of bug references is equal to the number of linked classes.

Table 10.2. The data set used for the experiments

System	Apache			Eclipse		
	Lucene	Maven	Mina	Eclipse CDT	Eclipse PDE	Equinox
Time period	Jan 1, 2005 Oct 8, 2008	Jan 1, 2005 Feb 18, 2009	Jan 14, 2006 Dec 10, 2008	Jun 24, 2006 Feb 25, 2009	Jan 1, 2005 Sep 11, 2008	Jan 1, 2005 Jun 25, 2008
Last release	2.4.0	2.0.10	2.0.0-M4	5.0.2	3.4.1	3
Versions	99	108	76	70	97	91
Bug references (tot in history)	982	1,500	629	923	4,953	2,043
Classes (avg)	513.5	156.2	108.6	217.8	1,170.5	242.6
Brain method (avg)	106.9	24.8	1.6	6.9	29.1	35.5
Dispersed coupling (avg)	14.2	2.9	1.6	0.4	63.0	31.8
Feature envy (avg)	943.7	74.7	70.1	90.6	1,006.8	450.8
Intensive coupling (avg)	0.9	6.6	1.3	3.4	1.5	4.3
Shotgun surgery (avg)	123.7	20.7	19.6	31.5	117.5	36.7

Once we extracted the design flaw data for all the versions of a system (all the FAMIX models), for each flaw we build a design flaw matrix F , similar to the matrix created to compute the churn and the entropy of source code metrics (cf. Section 7.3.5). The design flaw matrix F , exemplified in Figure 10.2, has the following properties:

- Each column represents a (bi-weekly) version of the system.
- Each row represents a class.

²The end of the time period always corresponds to a major release of the software.

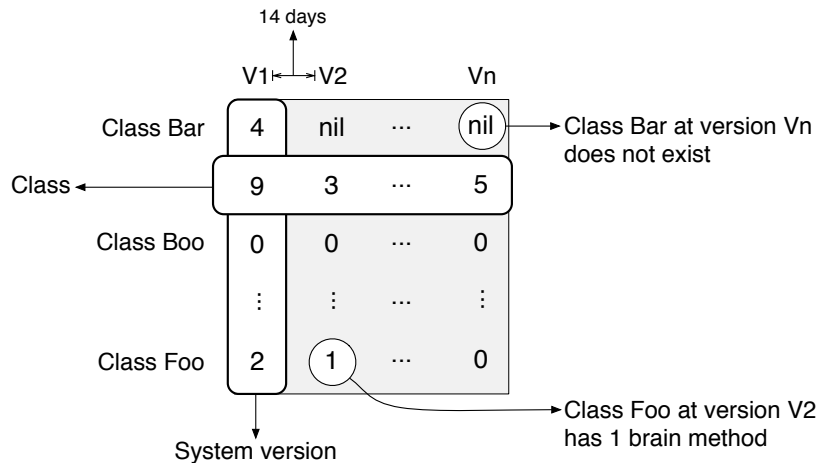


Figure 10.2. An example of design flaw matrix for brain methods

- The value of a cell at row r and column c is equal to the number of instances of the design flaw in the class represent by r at the version represent by c .
- Since some classes exist in some versions but not in others, the set of rows is composed of all the classes existing in at least one version. If a class at row r does not exist at the version in column c , we set the value of the cell c, r to nil.

10.3 Experiments

Before studying the relationship between design flaws and software defects, we want to examine the frequencies of different flaws in different systems. In particular, we aim at answering the following question: *Are there design flaws that are more frequent in all the systems, or is each system different with respect to design flaws frequencies?*

Table 10.2 provides a first indication that there are patterns of design flaws frequencies in the analyzed systems. However, we cannot compare the numbers of flaws in Table 10.2, as systems have different number of classes. To better investigate our question, we compare the average number of flaws per class (which is also an average over all the versions).

Figure 10.3 shows the average number of design flaws per class, grouped by flaw: Feature envy is the most frequent in all the systems, followed by shotgun surgery, which is stable for all the systems. Intensive coupling is relatively low for all systems, while dispersed coupling and brain methods vary.

To analyze the relationship between design flaws and software defects we perform two sets of experiments:

1. Correlation analysis: We study the correlation between number of post-release defects per class and class-level design flaws.
2. Delta analysis: We investigate whether increments of flaws correlate with defects, within a given time window.

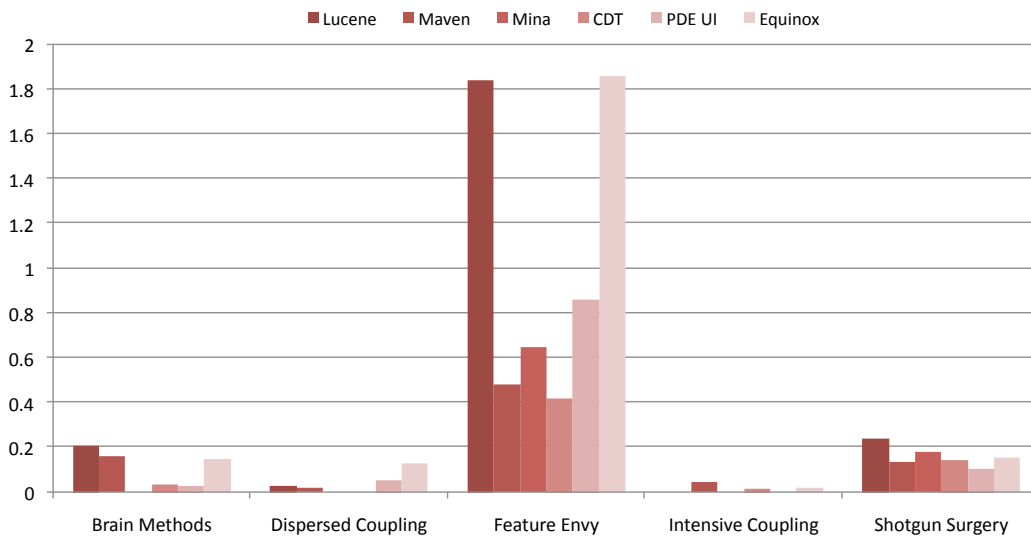


Figure 10.3. Average number of design flaws per class, grouped by flaw

10.3.1 Correlation Analysis

We found that some design flaws are more frequent than others. Now we want to study the correlation between number of design flaws and number of post-release defects at the class level. Our goal is to answer the following questions: *Do design flaws correlate with post-release defects? Does any flaw consistently correlate more than others in all systems?*

In our analysis we consider post-release defects, *i.e.*, defects reported after the considered version (release) of the system. For example, if we consider class `Foo` at version x , post-release defects are defects linked to `Foo` reported after x and within a certain time window. As Zimmermann *et al.*, we consider a period of six months for post-release defects [ZPZ07]. We consider post-release defects because we want to investigate whether the presence of design flaws generate bugs *in the future*.

To compute the correlation we need two lists: The first one contains the number of design flaws for each class, and the second one the number of post-release defects. The first list corresponds to a column of the matrix F , while the second is mapped to a column in an analogous matrix B . In B , rows represent classes, columns represent versions and the value of a cell at position c, r represents the number of post-release defects relative to the version c of the class r . To compute the number of post-release defects, we filter the extracted bug data according to the bug reporting dates.

To measure the correlation we could use either the Pearson's linear correlation coefficient, which should be used to linear relationships, or the Spearman's rank correlation coefficient, suitable for general associations [Tri06]. To choose which measure is more appropriate we studied the distribution of the data, which resulted to be highly skewed with respect to a normal distribution. In fact, most of the classes have very few or zero design problems and post-release defects. For this reason we opted for the Spearman's coefficient, as it is recommended with data that is skewed or that contains outliers [Tri06].

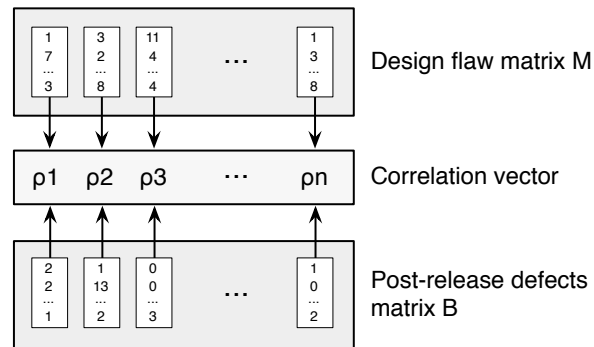


Figure 10.4. Computing Spearman's correlations over multiple versions of a system

Figure 10.4 shows how we compute the correlation over all the versions of a given system, producing a correlation vector that represents the correlation trend over time. We create the vector by computing the Spearman's coefficient on pairs of columns from the design flaws and bugs matrices (F and B).

Figure 10.5 shows, for each software system in our dataset, plots of the correlation vectors for all the considered design flaws. Interruptions in the lines mean that in the corresponding version of the system the Spearman's correlation coefficient was not significant. We see that every system is different and there is no flaw that is consistently more correlated with post-release defects across the systems. Moreover, not even within systems design flaws are consistently more or less correlated with respect to each other: They oscillate over different versions. To obtain a better grasp of how much flaws are correlated with defects, we only consider strong correlation, *i.e.*, the ones having a Spearman's coefficient above 0.4, which is the threshold for considering a correlation to be strong in fault prediction studies [ZN08].

Figure 10.6 shows the percentage of versions with a strong correlation, by design flaw and by software system. The percentages are computed from the correlation vectors as number of versions with a strong correlation (greater than 0.4) divided by total number of versions. We see that there are two types of systems: (1) Systems where no design flaw is strongly correlated with defects (PDE and Lucene) and (2) systems in which one design flaw is frequently strongly correlated with defects (Equinox, CDT, and Maven). Mina does not belong to any group, as some design flaws rarely have a strong correlation with post-release defects, but none of them is much more frequent than the others (as in Equinox, CDT, and Maven).

From the correlation analysis we draw the following conclusions:

- Design flaws correlate with post-release defects, but not strongly in all the analyzed systems.
- There is no design flaw that consistently correlates more than others in all the systems.
- In some systems there is no design flaw that strongly correlates with defects.
- Some systems are characterized by a particular design flaw, *i.e.*, one flaw is strongly correlated with defects much more frequently than all the others.



Figure 10.5. Spearman's correlations between number of design flaws and number of post-release defects over multiple versions of software systems

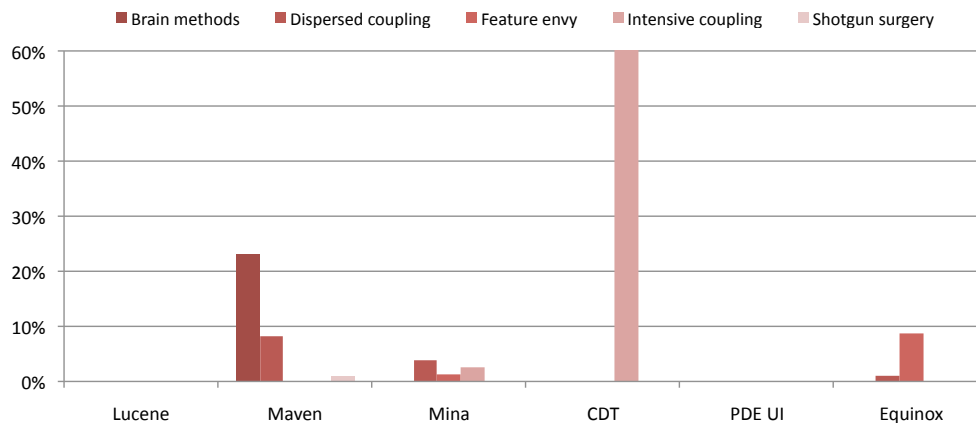


Figure 10.6. Percentages of systems' versions with a strong correlation (Spearman's $\rho > 0.4$) with post-release defects

10.3.2 Delta Analysis

In the second part of our experiments we want to investigate whether an addition of design flaws in a class generates bugs. We aim at answering the following questions: *Do design flaws additions correlate with software defects? Is there a design flaw for which additions consistently correlate more than addition of any other design flaw? What is the relationship between “flaws-defects” correlation and “flaws additions-defects” correlation?*

To study design flaws additions, we need to detect, extract and measure these addition events. We do so by analyzing each row in the design flaw matrix F , as depicted in Figure 10.7. Given a row, which includes the number of design flaws in all the considered versions of a class, we first compute the deltas between each consecutive pair of versions: A positive delta represents an addition of design flaw in the given class. Then, given the deltas row, we group all the sequences of positive values, where a sequence indicates a “longer” addition (see Figure 10.7).

To analyze the relationship between the detected sequences and software defects, we count the number of bugs (linked to the considered class) reported from the beginning to the end of the sequence plus a time window of 90 days.³ In case the sequence is composed of a single delta, the beginning of the sequence coincides with its end. We finally build two lists that we use to compute the correlation, where each element represents a design flaw addition (a sequence): The first list measures the total addition value, while the second one counts the number of defects reported during the addition period (the sequence) over a time window of 90 days.

Figure 10.8 shows, for each system and for each flaw, the Spearman's correlations between the two lists (the total addition values and the number of reported defects). For some flaws in some systems, not enough addition events were detected to obtain a significant correlation: In these cases we do not show the correlation. Looking at Figure 10.8 we conclude that:

- Additions of design flaws correlate with software defects, *i.e.*, introducing a design flaw in a class is likely to generate bugs that affect the class. However, this does not hold for all design flaws in all software systems.

³To be conservative, we use a time window which is half of the post-release defect time window.

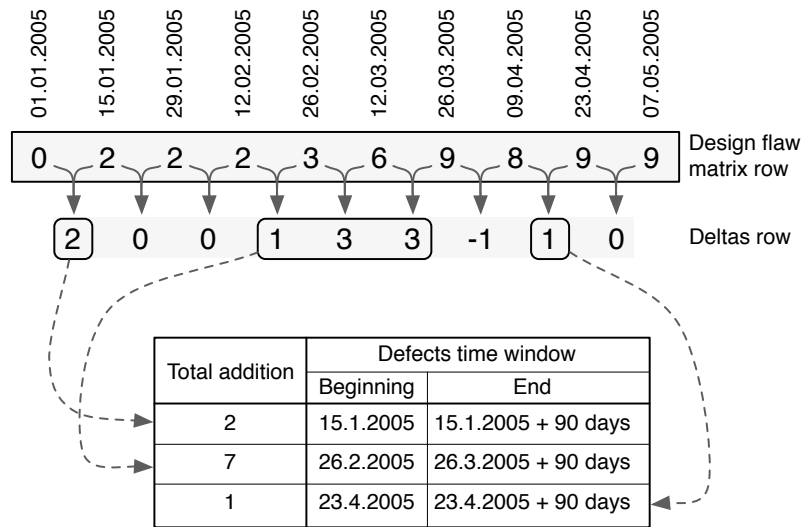


Figure 10.7. Extracting and measuring design flaw addition events

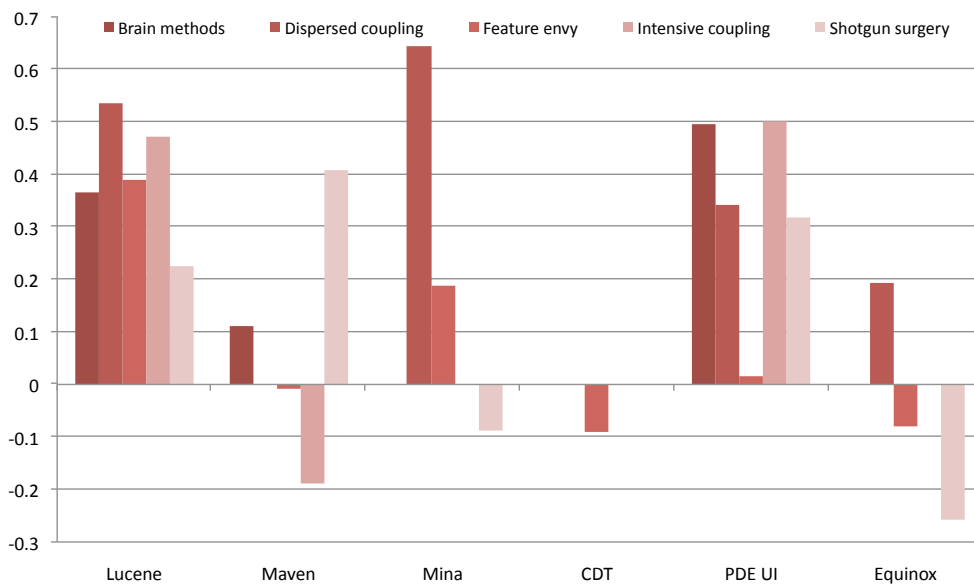


Figure 10.8. Spearman's correlations between additions of design flaws and number of generated defects

- There is no design flaw addition which consistently correlates more than others in all the systems.

Comparing correlations of defects with absolute numbers of design flaws and design flaw additions, we notice the following facts: In PDE and Lucene no design flaw is strongly correlated

with defects (see Figure 10.6), but at the same time, adding any design flaw in these systems is likely to introduce bugs, as the correlations between flaw additions and defects is relevant for all the design flaws (with the exception of feature envy in PDE). On the other hand, for systems characterized by a particular flaw, an addition of that flaw is not likely to introduce defects or, at least, not as much as other flaws. For example, in Maven brain method is far more frequently correlated with defects than other flaws; however, an addition of brain methods in Maven is not as likely to introduce defects as an addition of shotgun surgery. The same holds for Equinox with shotgun surgery and CDT with intensive coupling.

10.3.3 Wrapping Up

The goal of our experiments was to answer a number of questions concerning the frequency of design flaws in the analyzed systems, their correlations with software defects and whether their introduction to software entities is likely to generate bugs. The experiments showed that feature envy is the most frequent design flaw, but it is not the most correlated with software defects. Our correlation analysis revealed that none of the analyzed design flaws is more correlated with defects than others consistently across systems. Similarly, adding design flaws is likely to introduce defects in many (but not all) software systems, but no design flaw addition correlates with defects more than the others consistently across systems.

Moreover, we found out that some software systems are characterized by a specific design flaw “ f ” (different from system to system), in the sense that in these systems the flaw f is strongly correlated with defects much more frequently (across versions) than all the others. Interestingly, by performing deltas analysis, we discovered that an addition of f in these systems is not likely to introduce defects or, at least, not as much as other flaws that are less correlated with defects than f . A possible explanation of this finding is that, since in these systems the flaw f is very frequent, developers know how to deal with it without generating bugs. We also found systems in which no design flaw is strongly correlated with defects. Adding any flaw in such systems is likely to introduce defects.

10.4 Threats to Validity

Threats to construct validity. We already discussed some of these threats in Section 3.2.3. Another construct validity threat, concerning design flaws, is that we defined them using the detection strategies proposed by Marinescu [Mar04] to detect violations against design guidelines. These guidelines are based on thresholds statistically assessed on 45 Java systems. Although such a comprehensive number of software systems—from various projects and domains—was considered, they do not cover all the possible cases, and this can vary the effectiveness of the thresholds in identifying design flaws.

Threats to statistical conclusion validity. In our experiments we use the Spearman’s correlation coefficient to evaluate the relationship between design flaws and software defects, and all the correlations are significant at the 0.01 level.

Threats to external validity. In our experiments, there are two threats regarding to this category: First, we considered open-source software systems only. Differences between open-source and industrial development could change our results. However, as also reported by Lanza and

Marinescu [LM06], design flaws also appear in industrial software systems and can be effectively detected using the employed detection strategy. Second, we only considered software systems developed in the Java programming language. This affects the generalization of our findings. However, having the possibility to use the same parser for all the case studies ensures that all the code metrics are defined identically for each system.

10.5 Summary

Design flaws are known to have a negative impact on quality attributes of software systems, for example in their flexibility, or maintainability. In this chapter, we performed an extended analysis of the relationship between design flaws and software defects: We studied design flaws and software defects in six different software systems developed by independent development teams and emerging from the context of two unrelated communities (Apache and Eclipse).

Looking at the frequencies of design flaws in these projects, we discovered that *feature envy* is consistently the most recurring design flaw. The feature envy disharmony refers to methods that access more the data of other classes, than the data of the class containing it. It might be a sign that the method was misplaced, and that it should be moved to another class. The most significant aspect about feature envy is that it is a sign of an improper distribution of a system's intelligence.

Afterward, our analysis—spreading over the data from a minimum of two years in the history of the chosen systems—showed that design flaws do correlate with software defects. Also, no flaw consistently correlates more than others across all the different systems. Finally, we found that an increase in the number of design flaws is likely to generate bugs, and still, there is no design flaw addition that consistently correlates more than others across the totality of the systems.

To make our experiments extendible and reproducible by other researchers, our datasets, and in particular the design flaws and post-release defects matrices, are publicly available at: <http://www.inf.usi.ch/phd/dambros/flaws-defects/>.

Part IV
Epilogue

Chapter 11

Conclusion

In the light of the ubiquity and complexity of today's software systems, it is no wonder that software maintenance and evolution takes up the most part of the cost of a software system. Understanding the evolution of large systems is a challenging problem, due to the sheer size of data to be analyzed. In this context, researchers devised approaches that mine software repositories to analyze software evolution, with two main goals: inferring causes of problems in a system and predicting its future.

To be able to analyze a software system, one first needs to create a model of it, according to the goal to be achieved with the analysis, *e.g.*, predicting the location of future defects or detecting critical software components. By reviewing the state of the art in software evolution analysis, we learned that most approaches address a specific maintenance problem, and model software evolution considering only one evolutionary aspect. These approaches, and the infrastructures that implement them, cannot be adapted to tackle different maintenance problems.

In this dissertation, we introduced an approach that takes an integrated view of software evolution, combining evolutionary information about source code and software defects. Our thesis is that this integrated approach supports an extensible set of software maintenance activities. To this aim, we devised an integrated meta-model of evolving software systems, and we implemented it in an extensible framework.

To validate our thesis, we created and evaluated, on top of our approach, seven software evolution analysis techniques. We presented all the techniques, showing that they support various maintenance tasks, targeted at inferring the causes of problems and predicting the future of software systems.

11.1 Contributions

During the course of this dissertation, we made a series of contributions to the state of the art in modeling and analyzing software evolution. We summarize the major ones in the following.

11.1.1 Modeling and Supporting Software Evolution

Limitations of existing software evolution analysis approaches come from development tools and practices that, although unsatisfying for today's development, still represent de-facto standards. Therefore, to better understand these limitations, in Chapter 2 we looked at the history of software evolution, providing a historical perspective on our work. Subsequently, we surveyed software evolution analysis techniques and, based on their limitations, we identified four requirements for an integrated approach able to support various software maintenance tasks:

- R1. Integration of different evolutionary aspects.
- R2. Flexibility with respect to creating new techniques on top of the approach and extending the meta-model.
- R3. Modeling of software defects as first class entities.
- R4. Replicability of the analyses performed and availability of the data.

In Chapter 3, we introduced *Mevo*, an integrated meta-model of evolving software, and *Churrasco*, an extensible framework that implements Mevo and serves as a basis to create analysis techniques and tools. Mevo integrates versioning system, source code and defect information (R1), and models defects as first class entities taking their histories into account (R3). Churrasco provides a flexible meta-model support and enables the creation of analysis tools on top of it (R2). The framework also offers a web portal from which all the models used in our analyses and experiments are accessible, thus facilitating replicability (R4).

11.1.2 Analyzing Software Evolution to Infer Causes of Problems

In the second part of the dissertation, we presented three visualization techniques—built on top of Churrasco—aimed at detecting causes of problems in a software system. The techniques focus respectively on the evolution of source code, the evolution of software defects, and their co-evolution.

Analyzing change coupling information. In Chapter 4, we focused on the evolution of source code, and especially on a particular aspect of it, namely change coupling. Change coupling is the implicit dependency among software artifacts that frequently change together. We presented the *Evolution Radar*, a visual approach to study change coupling information—extracted from our Mevo meta-model—at different levels of abstraction. We applied the visualization to two large open-source software systems, and we conducted a user experiment with a developer of a Smalltalk system. Through these series of experiments, we illustrated how the Evolution Radar supports a number of maintenance activities, such as restructuring, re-documentation, change impact estimation, spotting design issues, identifying reengineering candidates and understanding module dependencies.

Analyzing the evolution of software defects. The focus of Chapter 5 was the evolution of software defects. We introduced the *System Radiography* and the *Bug Watch* views, two interactive visualizations to analyze bug repositories at different levels of granularity. The System Radiography supports the analysis of a bug database as a whole, showing how open bugs are distributed in a system and highlighting critical components, *i.e.*, the ones affected by the most severe bugs. The Bug Watch view targets the inspection of one or few system's components, facilitating the characterization of bugs—based on their life cycle—and the identification of critical ones. By applying the visualizations to the bug repository of Mozilla—consisting of more than a quarter million bugs—we provided anecdotal evidence of the usefulness of the approach.

Analyzing the co-evolution of source code and bugs. After studying the evolution of code and defects in isolation, in Chapter 6 we looked at their co-evolution. We proposed the *Discrete Time Figure*, a visualization technique to analyze such co-evolution at any level of granularity. Based on the visualization, we formally defined a catalog of co-evolutionary patterns that can be automatically detected in a system, such as day-fly, addition of features, bug fixing, *etc.* We employed Discrete Time Figures on three open-source software systems, detecting co-evolutionary patterns and characterizing the components of the systems.

11.1.3 Analyzing the Evolution of a Software System to Predict Its Future

In the third part of the dissertation, we presented four studies on software defects: Two of them dealt with how to improve defect prediction techniques, and the other two concerned the relationships of software defects with change coupling and design flaws. The presented studies either regarded directly predicting the future of a system (*e.g.*, defect prediction), or investigated relationships that—once understood—can support the prediction.

Predicting defects with the evolution of source code metrics. Defect prediction is a very active research field, experiencing a growing interest by the software engineering community. Defect prediction concerns the resource optimization problems: Knowing where future defects will be would allow a project manager to optimize the maintenance resources. When surveying defect prediction approaches in Chapter 2, we observed that a baseline to compare these approaches does not exist.

In Chapter 7, we introduced such a baseline in the form of a publicly available benchmark, composed of hundreds of versions of five software systems. Moreover, we proposed two novel approaches, based on the evolution of source code metrics extracted from our Mevo meta-model. We then evaluated these novel techniques on our benchmark, together with a selection of representative approaches from the literature. The results showed that our techniques are the best performing ones, as they gave consistently good results across all five systems.

Improving defect prediction with information extracted from e-mail archives. In Chapter 3, we argued that the Mevo meta-model is extensible, and that the Churrasco framework supports its extensibility. As a proof of concept, in Chapter 8 we extended Mevo to model e-mail information.

Based on the extended meta-model, we devised several metrics that measure the popularity of source code artifacts within discussions taking place in e-mail archives. Then, we investigated whether these popularity metrics correlate with defects, and whether they could be used to

improve existing bug prediction approaches. Our experiments on four open-source software systems provided positive answers for both investigations.

Analyzing the relationship between change coupling and software defects. Already when surveying approaches for change coupling analysis (cf. Chapter 2) and when presenting the Evolution Radar (cf. Chapter 4), we observed that change coupling has long been considered a significant issue. However, no empirical study of its correlation with a tangible effect of degraded software quality—as for example software defects—had been done until now. In Chapter 9, we conducted such a study, providing empirical evidence—on three open-source software systems—that such a correlation exists. Further, we showed that change coupling data can be used to improve existing defect prediction models based on change and source code metrics.

Analyzing the relationship between design flaws and software defects. Design flaws are known to have a negative impact on software quality attributes, such as flexibility or maintainability. However, it is not clear what the actual impact of design flaws on measurable effects of low quality—such as software defects—is, and whether some flaws are more defect prone than others. In Chapter 10, we analyzed the relationship between a catalog of design flaws and software defects on six open-source software systems. We also studied the evolution of the flaws over multiple versions of the systems, to investigate whether adding design flaws is likely to generate defects. Our experiments showed that the presence and the addition of design flaws significantly correlate with software defects. However, there was no empirical evidence of one flaw being more defect prone than the others.

11.1.4 Tools

Software evolution analysis is unavoidably tied to tools, since they are necessary to cope with the sheer amount of data that characterizes large and long-lived software systems. As a technical contribution, we implemented a number of tools to support the research presented in this dissertation.

Churrasco is an extensible framework that implements the Mevo meta-model and, through a dedicated data interface, serves as a basis for the other analysis tools. Churrasco features a web interface to create models of software systems, hiding the data retrieval and processing tasks from the users. The tool also supports collaborative software evolution visualization and analysis (discussed in Appendix A).

The Evolution Radar visualizes change coupling information at different levels of abstraction. We implemented two versions of the Evolution Radar: as a stand-alone tool, and as an IDE enhancement.

Bug's Life is a visualization tool that supports the analysis of software bugs in the large, visualizing an entire bug repository, and in the small, rendering individual bugs.

BugCrawler visualizes the co-evolution of source code and bugs at various granularity levels. The tool features also a query engine that detects a catalog of co-evolutionary patterns in the visualized software entities.

Pendolino is a scriptable data analysis tool that computes and exports a variety of metrics and properties about Mevo models (*e.g.*, code metrics, design flaws, change coupling measures, *etc.*) in a format compatible with Matlab and other statistical tools.

11.2 Limitations and Future Work

During the work on this dissertation, we encountered promising future research directions. Some of them are ideas on how to overcome limitations of our approach. In the following, we outline possible future work, discussing also—when appropriate—the shortcomings that originate them.

Extending the meta-model. The Mevo meta-model combines versioning system, source code and defect information. In our dissertation we extended it to model also e-mail data extracted from mailing list archives. However, there are also other kinds of artifacts that contain information concerning the evolution of software systems, such as design documents, developers' discussions taking place in chats, debugging information recorded by IDEs [KM08], fine-grained versioning information [Rob08], built system data [Ada09], test case reports. We envision enriching our meta-model with some of these pieces of data. The challenge in this context will be how to reliably link them to source code entities.

Improving the quality of the data. In Section 3.2.3 we discussed the limitations of our approach to populate models of evolving software systems. These limitations include: (1) Not all links between software defects and source code artifacts are detected; (2) we do not handle renaming events in the history of software artifacts, *i.e.*, a renaming is seen as an addition and a deletion; (3) the data residing in Bugzilla repositories contain some noise [AADP⁺08]; (4) we deal with all SCM transactions in the same way, but some of them commit one conceptual change, while others groups several changes. We plan to devise new heuristics and algorithms to mitigate the impact of these shortcomings, as for example the approach proposed by Bachmann and Bernstein to improve the linking between SCM transactions and a bug reports [BB09].

Replicating the experiments on a nearly-ideal dataset. As mentioned above, in our approach not all links between bugs and source code artifacts are detected, *i.e.*, the links are only a fraction of all existing links. Bird *et al.* argued that this subset is not a fair representation of the full population, thus inducing a bias that might threaten the results obtained using the subset [BBA⁺09]. Later, Nguyen *et al.* investigated whether the bias exists also in a nearly-ideal dataset (the IBM Jazz software project), where the links between bugs and code are enforced by strict development guidelines, and not inferred by subsequent analysis. Based on their experiments, the authors conjectured that the bias is likely to be a symptom of the underlying development process instead of being due to the linking technique [NAH10]. We want to replicate the experiments presented in the third part of the dissertation on the IBM Jazz software project, to inspect whether the bias exists in our dataset and, in the positive case, assess its impact on our results.

Generalizing our approach. We applied our approach to open-source software projects only. However, development practices in industrial settings may differ and conduct to different compartments in the developers, and thus to different results. Moreover, all the experiments pre-

sented in the third part of the dissertation (predicting the future) were performed on Java systems. The language might have an impact on the results, as communities using different languages can have different developer cultures. On the other hand, these choices have also some benefits. Using open-source systems allowed us to share our datasets, thus facilitating the replication of our experiments. Using the same language enabled the usage of the same parser for all the case studies, avoiding threats due to behavior differences in parsing, a known issue for reverse engineering tools [KSS02]. Nevertheless, to corroborate our findings, we plan to apply our analysis techniques on industrial systems as well as systems written in other object-oriented languages.

Conducting user studies. In most of the cases, we were the only users of our tools. Since these tools are designed to support analysts, project managers and developers in performing software maintenance tasks, we should involve such practitioners in assessing the tools usefulness and usability. We want to conduct a series of user studies to carry out such an assessment.

Analyzing other evolutionary relationships. In Chapters 4 and 9 we studied change coupling, the implicit and evolutionary dependency of software artifacts that frequently change together. We define another implicit and evolutionary relationship, the one between two or more software artifacts that in their histories were affected, either at the same time or in different time periods, by the same bug. We name this kind of relationship “*bug sharing*”: We plan to analyze it and compare it with change coupling.

Predicting defects with co-evolutionary patterns. In Chapter 6 we formally defined a catalog of co-evolutionary patterns that characterize the co-evolution of source code and bugs. Since the patterns can be automatically detected, we want to investigate whether they can be used to improve existing defect prediction techniques.

Exploiting author information. The Mevo meta-model includes author information, such as who performs a commit, creates a bug report, is in charge of fixing a bug, is in charge of verifying a fix for a bug, *etc.* We plan to explore how such information can be exploited, since with our approach we did so only to a limited extent.

Integrating the visualization tools. Our visualization tools—the Evolution Radar (cf. Chapter 4), Bug’s Life (cf. Chapter 5) and BugCrawler (cf. Chapter 6)—although being implemented on top of the same framework (Churrasco), are independent. We want to integrate them to allow one navigating among the various views the tools offer, as for example “jumping” from a Bug Watch view of a bug to a Discrete Time Figure of the component that bug affects.

Visualizing fine grained change coupling data. The Evolution Radar—presented in Chapter 4—visualizes change coupling information at the file and module granularity. We want to extend the tool to visualize finer grained change coupling [ZWDZ05; YMNCC04; BZ06], *i.e.*, at the method level. The challenge in this context will be how to render huge amounts of data, while keeping the visualization intelligible and useful.

11.3 Closing Words

In this dissertation, we showed that integrating evolutionary information about source code and software defects leads to novel techniques for both software evolution analysis and supporting maintenance tasks. The evolution of software, however, is not only defects revolving around the changing code: It is a *holistic* process with a variety of facets, which leaves traces in distinct repositories. New repositories open different perspectives on the evolution but, as we did with e-mail data, they must always be *integrated* with the source code—the “place” where software is changed.

Our thesis, while enlightening the central role of such integration, is only a first step towards capturing software evolution as a holistic phenomenon.

Part V
Appendices

Appendix A

Supporting Collaborative Software Evolution Analysis

In our dissertation, we presented a software evolution analysis framework called *Churrasco*. We discussed how Churrasco supports a number of analysis techniques, aimed at inferring causes of problems in software systems and at predicting their future. In this appendix, we present another feature of Churrasco: the collaboration support. The framework provides a set of collaborative visual analyses and supports collaboration by allowing users to annotate the analyzed data.

The need of collaboration in software development is receiving more and more attention. Tools that support collaboration, such as Jazz for Eclipse [Fro07], were only recently introduced, but hint at a larger current trend. Just as software development teams are geographically distributed, consultants and analysts are too. Specialists in different domains of expertise should be allowed to collaborate without the need of being physically present together. Because of these reasons, we argue that software evolution analysis should be a collaborative activity. As a consequence, *software evolution analysis tools should support collaboration*, by allowing different users, with different expertise, from different locations, to collaboratively analyze a system.

Moreover, we argue that software evolution analysis tools should possess the following characteristics, related to collaboration:

- *Accessibility*. Researchers developed a plethora of evolution analysis tools and environments. One commonality among many prototypes is their limited usability, *i.e.*, often only the developers themselves know how to use them, thus hindering the development and/or cross-fertilization of novel analysis techniques. There are some notable exceptions, such as Moose [DGN05], which were used by a large number of researchers over the years. Researchers also investigated ways to exchange information about software systems [KZK⁺06; TDD00], approaches which however are seldom followed up because of lack of time or manpower. We argue that software evolution tools should be easily accessible: In a collaborative settings, where the participants are likely to have different hardware configurations running different operating systems, the analysis tool should be usable without any strings attached.

The Churrasco framework provides an easily accessible web interface both to import software systems (create models) and to analyze them by means of web-based visualizations.

- *Incremental storage of results.* Results of analyses and findings on software systems produced by tools are often written into files and/or manually crafted reports, and are therefore of limited use. A different approach would be to incrementally and consistently store analysis results back into the analyzed models: This would allow researchers to develop novel analyses that exploit from the results of a previous analysis (cross-fertilization of ideas/results). It would also permit asynchronous collaboration, in which the participants do not have to perform the analysis at the same time. Finally, it would serve as a basis to create a benchmark for analyses targeting the same problem, or to combine techniques targeting different problems.

Churrasco stores the findings into a central database to create an incrementally enriched body of knowledge about a system, which can be exploited by subsequent users.

Structure of the appendix. In Section A.1 we discuss how Churrasco supports collaboration, detailing the web-based visualizations offered by the tool. We then provide an example of a collaborative session and describe two collaboration experiments performed with Churrasco (cf. Section A.2). In Section A.3 we discuss pros and cons of Churrasco, and examine tool building issues. We survey related work in Section A.4, and conclude in Section A.5.

A.1 Churrasco's Collaboration Support

In Section 3.4.1 we presented Churrasco and its architecture, focusing on the components to import software projects, process the data and instantiate Mevo models. Here we discuss in detail Churrasco's components that support collaboration, which were only briefly introduced in Section 3.4.1. Such components are:

- *The Visualization module* supports software evolution analysis by creating and exporting interactive web-based visualizations.
- *The Annotation module* supports collaborative analysis by enriching any entity in the system with annotations. It communicates with the web visualizations to depict the annotations within the visualizations.

A.1.1 The Visualization Module

The visualization module offers the following interactive visualizations that support software evolution analysis: The Evolution Radar, the System Complexity and the Correlation view.

The Evolution Radar supports software evolution analysis by depicting change coupling information. We comprehensively presented this visualization technique in Chapter 4.

The System Complexity View supports the understanding of object-oriented systems, by enriching a simple two-dimensional depiction of classes and inheritance relationships with software metrics (see left part of Figure A.1). By default, the size of the nodes is proportional to the number of attributes (width) and methods (height), while the color represents the number of lines of code. This mapping can be changed from the Churrasco's web interface, by assigning any software metric, from a rich catalog, to the width, height and color of the nodes. The goal of the visualization technique is to provide clues on the complexity and structure of a system.

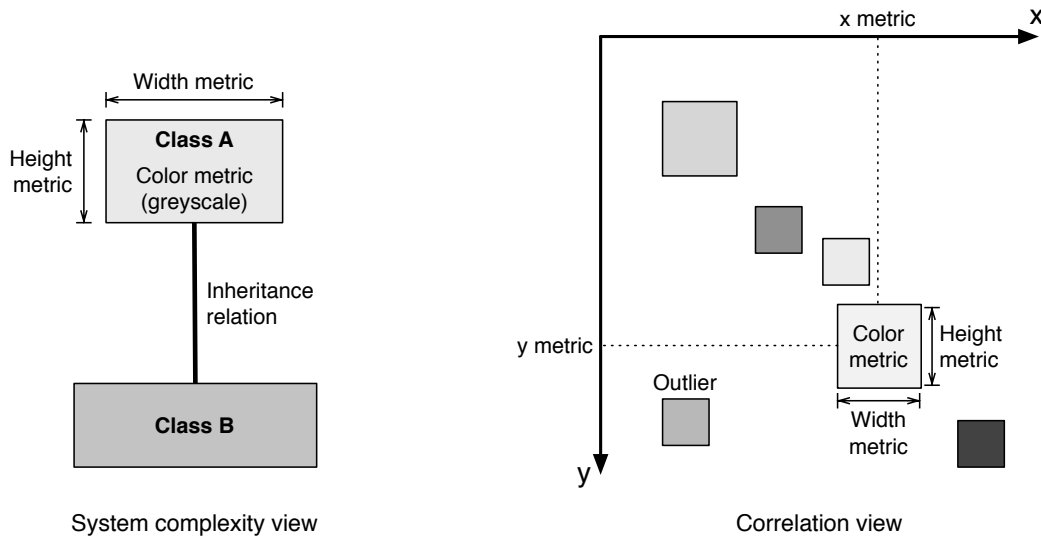


Figure A.1. System Complexity and Correlation View principles

The Correlation View shows all the classes of a software system in a two-dimensional space, using a scatterplot layout and mapping up to five software metrics on them: on the vertical and horizontal positions, on the size and on the color (see right part of Figure A.1). The default mapping is the following: The nodes' coordinates represent the number of attributes (x) and methods (y), the color represents the number of lines of code, while the size of the nodes is fixed. As for the System Complexity view, the mapping can be changed at any time using Churrasco's web interface. The Correlation view is useful to understand the correlation between different metrics in a software system and to detect outliers, *i.e.*, entities having metric values completely different with respect to the majority of the entities in the system.

In the collaboration experiments discussed later, we provide examples of Evolution Radar and System Complexity visualizations, but not of Correlation view. We give an example of this view here, depicted in Figure A.2. Nodes represent classes of the ArgoUML software system, where the x position is proportional to the number of lines of code, the y is proportional to the number of post-release bugs and the color maps the number of methods. Such a choice of metrics mapping can be useful to understand whether larger classes (higher number of lines of code) generate more bugs. This correlation does not hold in the case of ArgoUML (see Figure A.2). Moreover, we spot some outliers in the view: The one marked as "A", which has an outstanding number of bugs, and the ones marked as "B", with an outstanding number of lines of code.

The Correlation view and the System Complexity visualization are created using the Mondrian framework [MGL06] (residing in Moose) and the Episode framework [Pri07] (residing in Churrasco's visualization module). To make the visualizations interactive within the web portal, Episode attaches Ajax callbacks to the figures.

Figure A.3 shows an example of a System Complexity visualization rendered in the Churrasco web portal. The main panel is the view where all the figures are rendered as SVG graphics. The figures are interactive: Clicking on one of them will highlight the figure (red boundary),



Figure A.2. A Correlation view applied to the ArgoUML software system. Nodes represent classes, nodes' position represents number of lines of code (x) and number of post-release bugs (y), and nodes' color maps the number of methods.

generate a context menu and show the figure details (the name, type and metric values) in the figure information panel on the left. Under the information panel Churrasco provides three other panels useful to configure and interact with the visualization:

1. The metrics mapping configurator, which allows the user to customize the view by changing the metrics mapping.
2. The package selector, which allows the user to select, and then visualize, multiple packages or the entire system.
3. The regular expression matcher, with which the user can select entities in the visualization according to a regular expression.

A.1.2 The Annotation Module

The idea behind Churrasco's annotation module is that each model entity can be enriched with annotations to (1) store findings and results incrementally into the model and to (2) let different users collaborate in the analysis of a system in parallel. Annotations can be attached to *any* visualized model entity, and each entity can have several annotations. An annotation is composed of the author who wrote it, the creation timestamp and the text. To support persistent annotations, we extended the Mevo meta-model: Adding annotations results in enriching Mevo models. Persistence is provided by the Meta-base component (see Appendix B).

When the user selects the menu action "Show annotations" an additional panel is rendered at the top left corner of the web page (above the recent annotation panel). The panel shows all the annotations for the selected entity and allows the user to delete (only) his/her annotations. Selecting the "Add annotation" menu item will result in displaying another panel (again in the top left corner) that allows the user to write and add new annotations to the selected entity. Since the annotations are stored in a centralized database (enriching Mevo models), any new annotation is immediately visible to all the people using Churrasco, thus allowing different users to collaborate in the analysis. Churrasco features three other panels aimed at supporting collaboration:

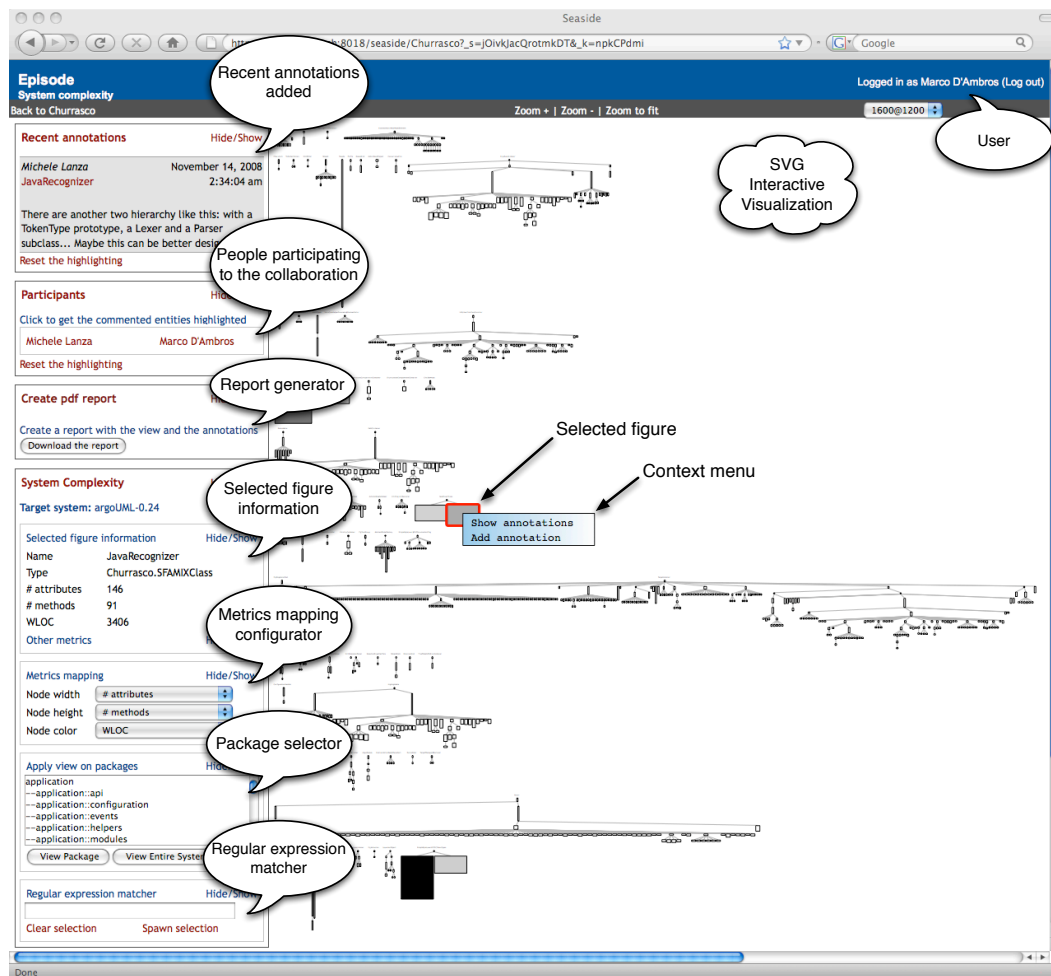


Figure A.3. A screenshot of the Churrasco web portal showing a System Complexity visualization of ArgouML

1. The “Recent annotations” panel displays the most recent annotations inserted, together with the name of the annotated entity, and—by clicking on it—the user can highlight the corresponding figure in the visualization.
2. The “Participants” panel lists all the people who annotated the visualizations, *i.e.*, people collaborating in the analysis. When one of these names is selected, all the figures annotated by the corresponding person are highlighted in the view, to see which part of the system that person is working on.
3. The “Create pdf report” panel generates a pdf document containing the visualization and all the annotations referring to the visualized entities. Figure A.4 shows a modified excerpt (modified to fit in the page) of such a report: In the visualization part, the entities having at least one annotation are highlighted in red, and the corresponding annotations are listed, together with the author and date information.

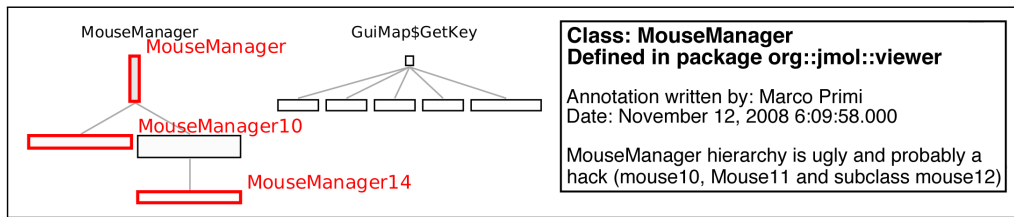


Figure A.4. An excerpt of a report generated by Churrascode. The entities having one or more annotations are highlighted in red, and the corresponding annotations are provided.

A.2 Collaboration in Action

We show how Churrascode supports collaborative software evolution analysis through one simple example scenario, presented next, and two collaboration experiments with respectively 8 and 4 participants.

A.2.1 Analyzing ArgoUML

We use the following simple scenario to exemplify Churrascode's usage for collaborative analysis: Marco and Michele, working on different machines in different locations, study the evolution of ArgoUML, a UML modeling tool composed of about 2000 Java classes. The users first create a Mevo model by indicating the URL of the ArgoUML SVN and Bugzilla repositories in the importer page of Churrascode. Once the model is created and stored in the Churrascode's database, they start the analysis with a System Complexity view of the system. Each user renders the view in his web browser, and attaches annotations to interesting figures in the visualizations. The annotations are immediately visible to the other user on the left side of the browser window (in the annotations panel).

While Michele is analyzing the entire system, Marco focuses on the Model package, which contains several classes characterized by large number of methods and many lines of code. The entities annotated by Marco in the fine-grained view are then visible to Michele in the coarse-grained System Complexity. Marco has the advantage of a more focused view, while Michele sees the entire context. Figure A.5 shows Marco's view on the left, while Michele's one is depicted on the right. Marco selected the FacadeMDRImpl class (highlighted in red in Marco's view), and is reading Michele's comments about that class (highlighted in blue in Michele's view). The following are two examples of collaboration:

1. Marco, focusing on the Model package, annotates that the class FacadeMDRImpl shows symptoms of bad design: It has 350 methods, 3400 lines of code, only 3 attributes, and it is the only implementor of the Facade interface. Michele adds a second annotation that Marco's observation holds also with respect to the entire system, and that FacadeMDRImpl is the class with the highest number of methods in the entire system.
2. Marco sees that several classes in the Factory hierarchy implement the Factory interface, and also inherit from classes belonging to the AbstractModelFactory hierarchy. This is not visible in Michele's view (where Factory and AbstractModelFactory are highlighted in blue), who discovers that fact by highlighting the entities annotated by Marco and by reading his annotations.

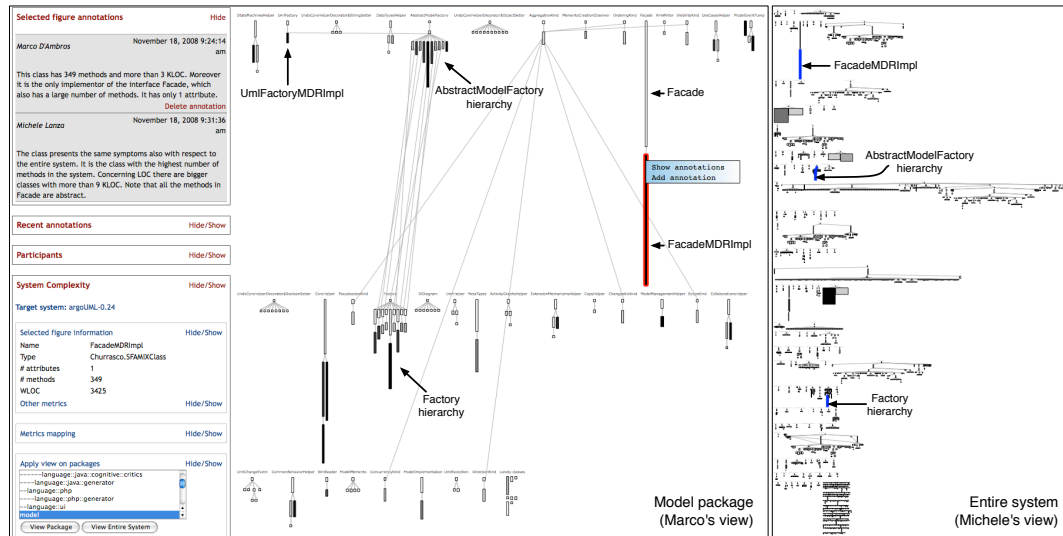


Figure A.5. The web portal of Churrasco visualizing System Complexities of the Model package of ArgoUML on the left, and the entire ArgoUML system on the right

Both the participants now want to find out whether these design problems have always been present in the system. They analyze the system's history in terms of its change coupling using the Evolution Radar. This visualization is time dependent, *i.e.*, different radar views are used to represent different time intervals. Figure A.6 shows, on the left, an Evolution Radar visualization corresponding to the time interval October 2004 – October 2005 and, on the right, the radar corresponding to October 2005 – October 2006. They both represent the dependencies of the Diagram module (displayed as a cyan circle in the center) with all the other modules of ArgoUML, by rendering individual classes.

Marco is looking at the time interval 2004 – 2005 (left part of Figure A.6). He selects the class `UMLFactoryMDRImpl` (marked in red), belonging to the Model module, because it is the closest to the center (*i.e.*, highest coupling with the Diagram module in the center) and because it is large (the size maps the number of changes in the corresponding time interval). Marco attaches to the class the annotation that it is potentially harmful, given the high coupling with a different module (Diagram), with respect to the one the class belongs to (Model).

In the meantime, Michele is looking at the time interval 2005 – 2006 (right part of Figure A.6). He highlights the classes annotated by Marco and sees the `UMLFactoryMDRImpl` class. In Michele's radar, the class is not coupled at all with the Diagram module, *i.e.*, it is at the boundary of the view (marked in red). Therefore, Michele adds an annotation to the class, saying that it is probably not harmful, since the coupling decreased over time. After reading this comment, Marco goes back to the System Complexity view, to see the structural properties of the class in the system. The `UMLFactoryMDRImpl` class (marked in the left part of Figure A.5) has 22 methods, 9 attributes and 600 lines of code. It implements the interfaces `AbstractUmlModelFactoryMDR` and `UMLFactory`. After seeing the class in the System Complexity, Marco adds another annotation saying that the class is not harmful after all.

These pieces of information can then be used by other users in the future. For example,

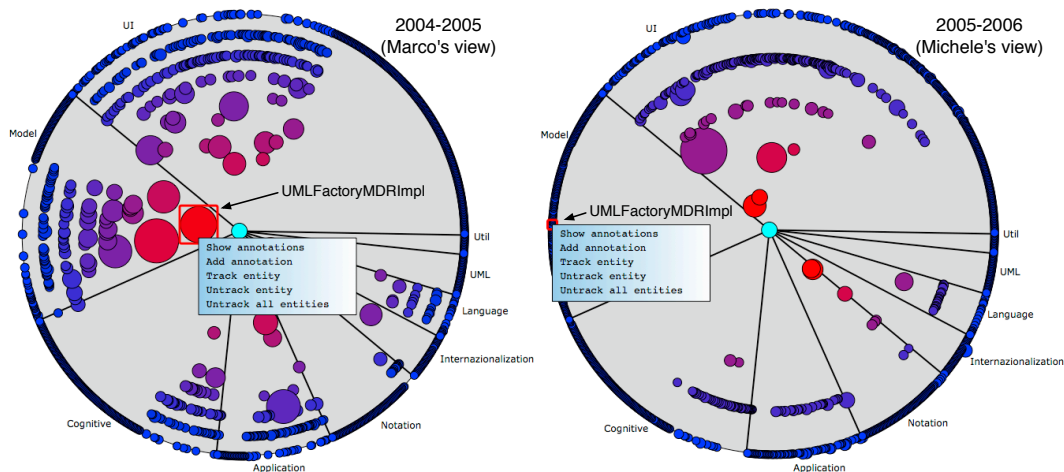


Figure A.6. Evolution Radars of ArgoUML

suppose that Romain wants to join the analysis with Marco and Michele, or to start from their results. He can first see on which entities the previous users worked on, by highlighting them, and then read the corresponding annotations to obtain the previously acquired knowledge about the system.

This simple scenario shows how:

1. The knowledge about a system, gained in software evolution analysis activities, can be incrementally built.
2. Different users, from different locations, can collaborate.
3. Different visualization techniques can be combined to improve the analysis.

A.2.2 First Collaboration Experiment: Analyzing Jmol

The previous example showed that Churrasco supports collaborative analysis. However, the example is hardly a collaborative experiment, because (1) there were only two participants (2) who were the developers of the tool, (3) possessing prior knowledge about the analyzed software system. Therefore, we decided to perform a collaboration experiment—in a more realistic settings—with the following goals: (1) Evaluate whether Churrasco is a good means to support collaboration in software evolution analysis, (2) test the usability of the tool, and (3) test the scalability of the tool with respect to the number of participants.

We performed the experiment in the context of a university course on software design and evolution. The experiment lasted three hours: During the first 30 minutes we explained the concept of the tool and how to use it, in the following two hours (with a 15 minutes break in the middle) the students performed the actual experiment, and in the last 15 minutes they filled in a questionnaire about the experiment and the tool. The participants were: five master students, two doctoral students working in the software evolution domain and one professor. The master students were lectured on reverse engineering topics before the experiment.

The task consisted in using the System Complexity and Correlation views and looking at the source code to (1) discover classes on which one would focus reengineering efforts (explaining why), and (2) discover classes with a big change impact and explain why. The target system chosen for the experiment was Jmol,¹ a 3D viewer for chemical structures, consisting of ca. 900 Java classes. Among the participants only one possessed some knowledge about the system.

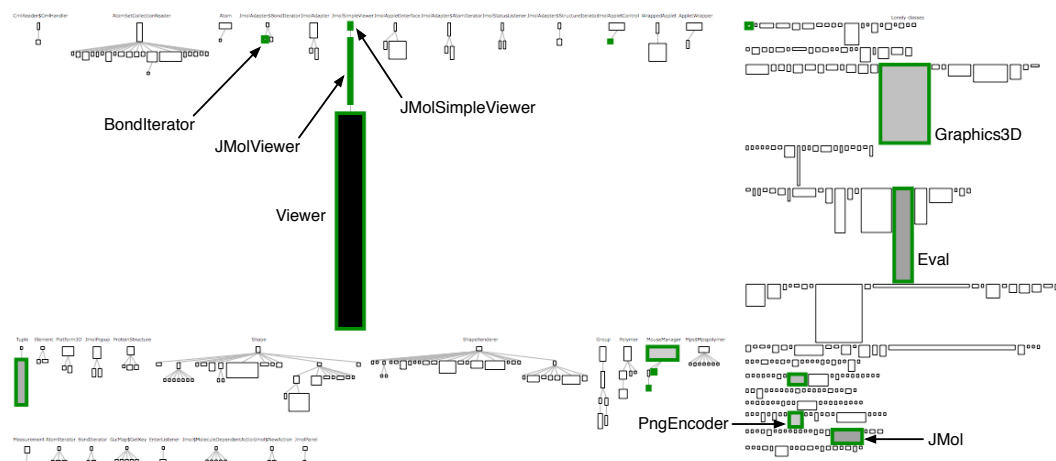


Figure A.7. A System Complexity of Jmol. The color denotes the amount of annotations made by the users. The highlighted classes (green boundaries) are annotated classes.

Figure A.7 shows a System Complexity of Jmol in which nodes' size maps number of attributes (width) and methods (height) and nodes' color represents the amount of annotations they received, *i.e.*, number of annotations weighted with their length. We see that the most annotated class is Viewer, the one with the highest number of methods (465). However, we can also see that not only the big classes (with respect to methods and/or attributes) were commented, but also very small classes.

Table A.1 lists a subset of the annotations made by the users during the experiment. In the assigned time, the participants annotated 15 different classes for a total of 31 annotations, distributed among the different participants, *i.e.*, everybody actively participated in the collaboration. The average number of annotations per author was 3.87, with a minimum of 2 and a maximum of 13.

The annotations were also used to discuss about certain properties of the analyzed classes. In most of the cases, the discussion consisted in combining different pieces of knowledge about the class (local properties as number of methods with properties of the hierarchy with dependency, *etc.*).

At the end of the experiment all participants but one filled in a survey about the tool and the collaboration experience. Table A.2 reports the results of the survey. In the cases where the sum of the answers is not seven, a participant did not indicate an answer.

Although not a full-fledged experiment, it provided us with information about our initial goals: The survey shows that the participants found the tool easy to use, collaboration important in reverse engineering and Churrasco as a good means to support collaboration (for all the

¹Jmol is available at: <http://jmol.sourceforge.net>

Table A.1. A subset of the annotations made on Jmol. NOA stands for number of attributes and NOM for number of methods.

Class	NOA	NOM	Annotations
JmolSimple-Viewer	0	8	"This is a strange hierarchy. There is only one subclass per superclass (all with many method and few attributes)."
JMolViewer	0	135	"Strange: 134 abstract methods, only 1 concrete, only 1 subclass."
Viewer	54	465	"This class seems to be the "thing" in the system, at least in terms of functionality", "Strong dependency with Eval.", "High fan out (25) and many LOC (>1k).", "High number of access to foreign data."
Eval	34	198	"This class should probably be broken down.", "Very strong dependency with Viewer."
JMol	60	25	"This class has the largest fan out (78). Probably part of the core of the system.", "High coupling, low cohesion", "13 protected methods and no child!"
PngEncoder	23	26	"17 protected attributes, completely useless since there's no child!"
BondIterator	5	5	"There are ca. 6 classes with Iterator logic. The implementation is strange. I would expect them to be in some hierarchy."
Graphics3D	101	166	"This can be probably broken down. It's an implementation of a 3d engine."

Table A.2. A subset of the results from the questionnaire, using a Likert Scale [Lik32] (SA = strongly agree, A = agree, N = Neutral, D = disagree, SD = strongly disagree)

Statement	SD	D	N	A	SA
Churrasco is easy to use			1	3	2
System Complexity view is useful				2	5
Correlation view is useful			1	1	5
Churrasco is a good means to collaborate					7
Collaboration is important in reverse engineering			1	5	1

participants the experiment was the first reverse engineering collaboration experience). Another result is that they found the provided visualizations useful to achieve the given tasks. Churrasco scaled well with eight people accessing the same model, on the web portal, at the same time, without any performance issue.

A final comment given by the users during an informal conversation after the experiment, is that they had fun in the collaborative session: They especially liked to wait for annotations from other people, on the entity they already commented, or to see what was going on in the system and which classes were annotated, to also personally look at them.

A.2.3 Second Collaboration Experiment: Collaborative Restructuring

The purpose of the first experiment was to perform a preliminary evaluation of Churrasco's usability and whether users would find Churrasco a good means to collaborate. In our second experiment we wanted to simulate a more structured form of collaboration, where users do not have all equal roles.

We performed the experiment in the context of a university course on software engineering,

with a setup very similar to the one of the previous experiment. This time, the participants were four bachelor students with little knowledge about reverse engineering. The experiment took place during the last week of a project in which all students had developed a web application in Smalltalk during six weeks. During the last week of the project, the students could not add new features to the system, but they could only restructure / refactor it to improve its design and code quality.

The task that the students had in the experiment was to identify which parts of the system should be refactored, using the System complexity and Correlation views in Churrasco. The students had different roles in the collaboration: One acted as a leader, responsible to analyze the system, by selecting classes that he thought were candidates for refactoring; the other students would check in detail whether the classes in question needed to be refactored or not.

Using the annotations, the leader could also ask questions that the followers then answered. Typical questions were: “What is the responsibility of this class?”, “Can we remove this class?”, “These hierarchies seem to be duplicated, can we merge them?”, “Why is this class in this hierarchy? Should it not be a subclass of that class?” *etc.*

The target software system was composed of 166 classes and 983 methods, for a total of ca. 5,000 lines of Smalltalk code.

Table A.3. A subset of the annotations made on the Smalltalk web application

Class ElementModel
What is the difference between Element Model and Element? Are both hierarchies replicated?
One is the model that manages the functionalities of the element, the other one manages the displaying of the element (it is a proxy pattern).
One is for the layout behavior while the other one is for the widget behavior.
Class WBLBorderLayoutModel
This layout seems to have more behavior than the others, even though it has the same number of attributes. Maybe it is doing too much. Should it be a composite layout?
It has a lot of complex operations that being detached can raise the complexity much more. As you say, it has functionalities that can be distributed in more than one class.
The layout is complex. Dividing it into several classes will require too much time and effort.

Table A.3 shows a subset of the annotations made by the users during the experiment, the ones written for a couple of classes. These two groups of annotations exemplify how the collaborative session was performed: The leader was asking questions about the design of classes and hierarchies, and the other students were answering these questions.

During the experiment, the participants annotated 11 different classes for a total of 27 annotations, 9 written by the leader and 18 by the other students. Since the participants knew the system, they were faster in writing annotations with respect to the participants of the first collaboration experiment.

As in the previous experiment, at the end of the collaborative session the participants filled in a survey about the tool and the collaboration experience. Table A.4 presents a subset of the results. The survey shows that the participants found that collaboration helped them in understanding the system, and the used methodology (with the leader) was useful to structure the collaborative effort. Moreover, the use of annotations eased the task of selecting potential candidates for refactoring. Another result is that the students found that the collaborative support provided by Churrasco has an added value.

Table A.4. A subset of the results from the questionnaire, using a Likert Scale (SA=strongly agree, A=agree, N=Neutral, D=disagree, SD=strongly disagree)

Statement	SD	D	N	A	SA
Churrasco is a good means to collaborate				1	3
Collaboration helped me in understanding the system				3	1
The proposed methodology (leaders) helps in structuring the collaborative effort				3	1
Reading other users' annotations eases the given tasks				2	2
	1-2	3-4	5-6	7-8	9-10
Quantify (1-10) the added value of the collaborative support provided by the tool				2	2

A.3 Discussion

The main benefits of Churrasco are its accessibility and flexibility. The features of the framework can be accessed through a web browser: (1) The importers to create and populate Mevo models, (2) the System Complexity and Correlation views, to support the understanding of a system's structure and (3) the Evolution Radar visualization, to study the evolution of the system's modules in terms of change coupling.

The visualizations are interactive, and they allow users to inspect the entities represented by the figures, to apply new visualizations on-the-fly from the context menus, and to navigate back and forth among different views. The framework can be extended, with respect to the Mevo meta-model (as done for example in Chapter 8) and with respect to the visualizations.

A.3.1 Tool Building Issues

We decided to develop Churrasco as a web application, because it eased implementing it as a collaborative platform. However, developing a web-based tool that supports scalable and interactive visualizations raised a number of issues, related to interacting, updating, and debugging.

Interacting. Supporting interaction through a web browser is still a non-trivial task, and even supposedly simple features, such as context menus, must be implemented from scratch. In our Churrasco tool, we implemented the context menus as SVG composite figures, with callbacks attached, which are rendered on top of the SVG visualization. Moreover, it is hard to guarantee a responsive user interface, since every web application introduces a latency due to the transport of information.

Updating. The standard way of rendering a web visualization is that every time something changes in the page, the whole page is refreshed to show the updated version. In the context menu example, whenever the user selects a figure, the page changes because a new figure appears, and therefore the page needs to be refreshed to show the menu. This introduces latencies that make the web application unusable, when it comes to rendering very large SVG files. For this reason, we implemented many actions that do not require a complete re-rendering of a page using Ajax requests. Examples of such actions are: rendering of context menus, highlighting figures, displaying figure information, displaying and adding annotations.

Debugging. A barrier to develop web applications is the lack of support for debugging. Even if there are some applications such as Firebug,² providing HTML inspection, Javascript debugging and DOM exploration, the debugging support is not comparable with the one given in mainstream integrated development environments such as Eclipse.

A.3.2 Wrapping Up

All in all, while building Churrasco, we learned that creating a web application that supports interactive visualizations implies a number of technological challenges. On the other hand, web applications introduce a number of novel ways to interact with systems—as for example collaborative software analysis—that will open up new research directions.

A.4 Related Work

We are not aware of software visualization tools which support collaboration. However, a number of approaches support web-based software evolution visualization and analysis.

Sarma *et al.* introduced Tesseract [SMWH09], a Flash-based tool that provides interactive visualizations of relationships between files, developers, bugs, and e-mails. Tesseract features four cross-linked displays that show: (1) the project activity over time with respect to the number of commits and number of communications; (2) a graph of change coupling dependencies among files, (3) a graph of communication dependencies among developers and (4) the defects affecting the considered files. The main difference between Churrasco and Tesseract resides in the problem they address: While Churrasco is aimed at understanding a system's evolution, the goal of Tesseract is to support the analysis of the socio-technical relations between code, developers, and issues.

Another web-based visualization tool is the Java applet version of Shrimp³ [SM95]: It displays architectural diagrams using nested graphs where graph nodes embed source code fragments. The tool is fully interactive, providing animated panning, zooming, and fisheye-view actions. While Shrimp supports the exploration of software architecture, Churrasco focuses on software evolution analysis.

Beyer and Hassan proposed the Evolution Storyboards [BH06], a visualization technique that offers dynamic views. The storyboards, rendered as SVG files (thus visible in a web browser), depict the history of a project using a sequence of panels, each representing a particular time period in the life of a software project. This visualization is only partially interactive, *i.e.*, it only shows the names of the entities represented by the SVG figures. In contrast, the views offered in the Churrasco web portal are fully interactive, providing context menus, spawning and navigation capabilities.

Lungu *et al.* presented a web-based approach to visualize entire software repositories [LLGH07]. Their technique, validated on Smalltalk repositories, focuses on understanding the structure of the organization behind the repositories, by studying the interaction among the developers. They also provide views to see the evolution of the repositories over time. Both the approaches are fully interactive and web-based, but while Lungu's approach focuses on the entire repository evolution with coarse-grained views, Churrasco targets single projects with fine-grained visualizations.

²Firebug is available at <http://getfirebug.com>

³Available at <http://www.thechiselgroup.com/shrimp>

Mancoridis *et al.* introduced REportal, a web portal for the reverse engineering of software systems [MSC⁺01]. REportal allows users to upload their code (Java or C++) and then to browse, analyze and query it. These services are implemented by reverse engineering tools developed by the authors over the years. REportal supports software analysis through browsing and querying, whereas Churrasco supports the analysis by means of interactive visualizations.

Nentwich *et al.* presented BOX, a portable, distributed and interoperable approach to browse UML models [NEFZ00]. BOX translates a UML model represented in XMI into VML (Vector Markup Language), which can be directly displayed in a web browser. BOX enables software engineers to access and review UML models, without the need to purchase licenses of tools that produced the models. While BOX is focused on design documents, such as UML diagrams, in Churrasco we focus on the history and structure of software systems.

Finnigan *et al.* developed the Software Bookshelf, a web-based paradigm for the presentation and navigation of information representing large software systems [FHK⁺97]. The Software Bookshelf integrates various parsing and analysis tools in the backend, providing a means to capture, organize, and manage information about software systems. Differently from Churrasco, the goal of the Software Bookshelf is to support the re-documentation and migration of legacy systems.

A major difference between all the mentioned approaches and Churrasco is that these techniques support single user software evolution analysis, while Churrasco supports collaborative analysis.

A.5 Summary

The need of collaboration is receiving an increasing importance in software development. We argue that collaboration has also an important role in software evolution analysis. In this appendix, we presented how our Churrasco framework supports collaborative software analysis, by means of interactive visualizations and persistent annotations. Two major features of Churrasco, related to collaboration, are:

- *Accessibility.* The tool is fully web-based, *i.e.*, the entire analysis of a software system—from the initial model creation to the final study—can be performed from a web browser, without having to install or configure any tool.
- *Modeling of results.* Churrasco relies on a centralized database and supports annotations. Thus, the knowledge of the system, gained during the analysis, can be incrementally stored on the model of the system itself.

We showed, through a couple of collaboration experiments with respectively eight and four participants, that Churrasco is a good means to support collaborative software evolution analysis.

Appendix B

The Meta-Base

The *Meta-base* [DLP07b] is the core module of the Churrasco framework [DL08b], which provides flexibility and persistence to *any* meta-model in general, and to our Mevo meta-model in particular. As Churrasco, the Meta-base is developed in Smalltalk.

The tool takes as input a meta-model described in EMOF and outputs a descriptor, which defines the mapping between the object instances of the meta-model, *i.e.*, the model, and tables in the database. EMOF (Essential Meta Object Facilities) is a subset of MOF¹, a meta-meta-model used to describe meta-models. The Meta-base uses a Smalltalk implementation of EMOF called *Meta* and it ensures persistence with the object-relational module GLORP [Kni00] (Generic Lightweight Object-Relational Persistence).

The Meta-base provides flexibility by dynamically and automatically adapting to any provided meta-model: To do so, it generates descriptors of the mapping between the database and the meta-model. This allows us to dynamically both modify and extend our meta-model.

The tool can be used to exchange models (or only parts of them) through a database, thus supporting interoperability.

B.1 Object Persistence

The Meta-base relies on GLORP² for object persistence. GLORP is a powerful object-relational mapping layer for Smalltalk. It allows us to define the mapping between Smalltalk objects and tables/rows in a relational database (DB from now on). Once this mapping is defined, objects can be read from and written to the DB in a completely transparent way, without having to write any SQL statement.

Example

We want to define the mapping for simplified versions of FAMIXClass and FAMIXMethod. FAMIXClass has a name, belongs to a package and has a collection of methods, while FAMIXMethod has just a name.

¹MOF and EMOF are standards defined by the OMG (Object Management Group) for Model Driven Engineering. For more details about MOF and EMOF consult the specification at: <http://www.omg.org/mof/>

²Available at <http://www.glorp.org>

To define the mapping, we need to create a new class, *i.e.*, `FamixDescriptorSystem`, inheriting from `Glorp.DescriptorSystem`. In this class we add methods (1) to define the structure of the database table corresponding to the FAMIX class and method (see Listing B.1) and (2) to define the mapping between the tables and the classes (see Listing B.2).

```

1 tableForFAMIXClass: aTable
2   aTable createFieldNamed: 'Id' type: platform serial.
3   aTable createFieldNamed: 'Name' type: (platform varChar: 50).
4   aTable createFieldNamed: 'PackagedIn' type: (platform integer).
5
6 tableForFAMIXMethod: aTable
7   aTable createFieldNamed: 'Id' type: platform serial.
8   aTable createFieldNamed: 'Name' type: (platform varChar: 50).
9   aTable createFieldNamed: 'BelongsTo' type: (platform integer).

```

Listing B.1. The code snippet to specify the structure of the database tables corresponding to `FAMIXClass` and `FAMIXMethod`

```

1 descriptorForFAMIXClass: aDescriptor
2   | t |
3   t := self tableNamed: 'Class'.
4   tMethod := self tableNamed: 'Method'.
5   tAttribute := self tableNamed: 'Attribute'.
6   tPackage := self tableNamed: 'Package'.
7   aDescriptor table: t.
8   "direct mappings"
9   aDescriptor addMapping: (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
10  aDescriptor addMapping: (DirectMapping from: #name to: (t fieldNamed: 'Name')).
11  "one-to-one mapping"
12  (aDescriptor newMapping: OneToOneMapping)
13    attributeName: #packagedIn;
14    referenceClass: FAMIXPackage;
15    mappingCriteria:
16      (Join from: (t fieldNamed: 'PackagedIn') to: (tPackage fieldNamed: 'Id')).
17  "one-to-many mapping"
18  (aDescriptor newMapping: OneToManyMapping)
19    attributeName: #methods;
20    referenceClass: FAMIXMethod;
21    join: (Join from: (t fieldNamed: 'Id') to: (tMethod fieldNamed: 'BelongsTo')).
22  ^aDescriptor
23
24 descriptorForFAMIXMethod: aDescriptor
25   | t |
26   t := self tableNamed: 'Method'.
27   aDescriptor table: t.
28   "direct mappings"
29   aDescriptor addMapping: (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
30   aDescriptor addMapping: (DirectMapping from: #name to: (t fieldNamed: 'Name')).
31   ^aDescriptor

```

Listing B.2. The code snippet to define the mappings between the classes `FAMIXClass` and `FAMIXMethod` and the database tables/rows

Listing B.2 includes three kinds of mapping: direct, one-to-one and one-to-many.

Direct Mapping (Listing B.2 row 9, 10, 29 and 30). It expresses simple relationships between instance variables and table columns. It is used when the “type”³ of the instance variable is directly supported by the DB, for example for Integer, Text, Date, Timestamp, etc.

One-to-one Mapping (Listing B.2 row 12–16). It describes the relationship between FAMIXClass and FAMIXPackage. This mapping, declared in the FAMIXClass descriptor, defines the following properties:

- The attribute name, *i.e.*, the name of the instance variable getter.
- The reference class: It specifies the class of the objects, and therefore the corresponding table in the DB. In the considered case the class is FAMIXPackage, which corresponds to the Package table (not shown for brevity).
- The join expression: It defines which columns of the two tables are linked. GLORP uses this information to create the appropriate SQL join query to fetch the data from the DB and create the objects.

One-to-many Mapping (Listing B.2 row 18–21). It expresses that a FAMIXClass can have several FAMIXMethods. The structure of the mapping is similar to the one-to-one mapping, with two differences. First, the attribute name refers to a collection of objects instead of a single object. All these objects have to be instances of the class “referenceClass” (FAMIXMethod). Second, the data will be written in the table corresponding to the reference class (FAMIXMethod), instead of the current class (FAMIXClass). This is because each row in the FAMIXMethod table refers to a single row in the FAMIXClass table (the container class), whereas each row in the FAMIXClass table can refer to multiple rows in the FAMIXMethod table.

A last type of mapping, not used in the code snippet, is *many-to-many*. It expresses the most generic relationship by means of a link table. If two classes have this kind of relationship, the relationship itself is stored in a separated link table in the DB.

What about Inheritance?

We want to add to our simplified FAMIX meta-model a superclass of FAMIXClass, namely FAMIXAbstractNamedEntity, which has a name as instance variable. When adding this superclass, we also remove the *name* instance variable from the FAMIXClass class, since it is inherited from FAMIXAbstractNamedEntity.

GLORP provides two techniques to manage inheritance: filtered—exploited in the Meta-base—and horizontal. In the filtered inheritance all the classes are represented in a single table, with a discriminator field for which subclass they are. The table has the union of all possible fields for all classes. In the horizontal inheritance each concrete class is represented in its own table. Each table duplicates the fields that are in common between the concrete classes. Figure B.1 shows the two approaches for our examples.

With horizontal inheritance (Figure B.1(a)) the *name* column is duplicated in both tables; With filtered inheritance (Figure B.1(b)) the *PackagedIn* value is nil for the AbstractNamedEntity EntityA and there is the “Class” identifier column.

³To use GLORP we have to assume that an instance variable is always of the same class, called type.

AbstractNamedEntity	
Id	Name
1	EntityA

Class		
Id	Name	PackagedIn
1	ClassA	PackageA

AbstractNamedEntityAndClass			
Id	Name	PackagedIn	Class
1	EntityA	-	FAMIXAbstractNamedEntity
2	ClassA	PackageA	FAMIXClass

(a) Horizontal Inheritance

(b) Filtered Inheritance

Figure B.1. Types of inheritance in GLORP

Reading & Writing

Once we defined the mapping between the tables and the objects, *i.e.*, we completed the descriptor class, reading and writing objects is straightforward. The code snippet depicted in Listing B.3 reads all the FAMIXClass objects from a DB, modifies them and stores them back in the DB.

```

1 famixClasses := session readManyOf: FAMIXClass.
2 "the FamixClass objects are modified"
3 session registerAll: famixClasses.

```

Listing B.3. Reading and writing objects from/to the DB

The variable “*session*” is an object storing the connection with the DB. It is also possible to retrieve only the objects satisfying a given condition, as shown below:

```

1 famixClasses := session readManyOf: FAMIXClass
2   where: [:each | each isAbstract].

```

When we read a FAMIXClass from the DB, we retrieve—on demand—all the classes which have a relationship with it (in our example FAMIXMethod and FAMIXPackage). This means that the message “readManyOf:” sent to the session object retrieves FAMIXClasses only, not FAMIXMethods and FAMIXPackages. If we send the getter message “methods” or “packagedIn” to a FAMIXClass object, the collection of FAMIXMethod objects or the FAMIXPackage object are dynamically read from the DB.

B.2 Generating Descriptors with the Meta-base

The Meta-base takes as input a meta-model described in Meta and outputs a GLOP class descriptor, which defines the mapping between the object instances of the meta-model, *i.e.*, the model, and the database. Figure B.2 shows how the Meta-base works.

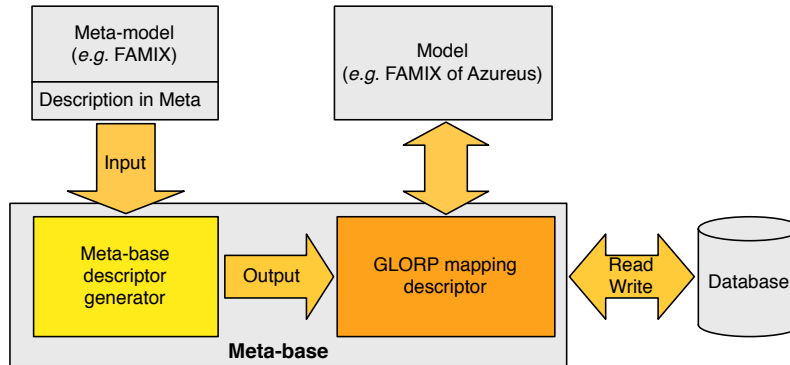


Figure B.2. Generating and using GLOP descriptors with the Meta-base

Using the Meta-base is straightforward. Suppose we have a meta-model described in Meta (a Smalltalk implementation of EMOF): To create the GLOP class descriptor with the Meta-base, we can use the following code snippet:

```

1 classes := OrderedCollection with: ClassA with: ClassB ...
2 ^ClassDBDescriptorGenerator uniqueInstance
3   createClassDescriptorForClasses: classes
4   named: 'Descriptor' in: aPackage.

```

This code generates the descriptor class named “Descriptor,” located in the “aPackage” package. The Meta-base uses the generated descriptor to define the mapping between the meta-model and the database. To get a connection with the database—which adheres to the mapping—we use the code below:

```

1 db := MetaDBBridge uniqueInstance.
2 db descriptorClass: Descriptor.
3 db login: ((Login new)
4   username: 'user'; password: 'pass';
5   connectString: 'databaseServerLocation';
6   database: PostgreSQLPlatform new; yourself).

```

Once we created the the database connection, we can generate the tables on the database (if the database is empty) with:

```

1 db createTables

```

and read/write objects of the model with:

```

1 someClasses := db session readManyOf: ClassA.
2 someClasses addAll: (db session readManyOf: ClassB).
3 "someClasses are modified"
4 db session registerAll: someClasses.

```

B.3 Example

We present a simple example which shows how the Meta-base supports inheritance and all types of relationships (direct, one-to-one, one-to-many and many-to-many).

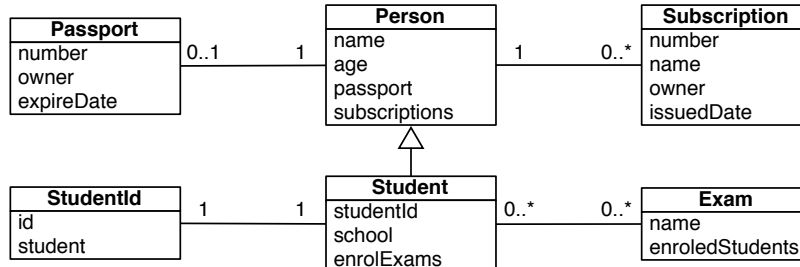


Figure B.3. The UML class diagram of our example

Figure B.3 shows the UML class diagram of an example meta-model, while the code snippet below shows part of its Smalltalk EMOF description (the classes Person and Passport).

```

1 Person>>metamodelAge
2   ^ (EMOF.Property name: #age type: Number)
3 Person>>metamodelName
4   ^ (EMOF.Property name: #name type: String)
5 Person>>metamodelPassport
6   ^ (EMOF.Property name: #passport
7     opposite: #owner type: Passport)
8 Person>>metamodelSubscription
9   ^ (EMOF.Property name: #subscription opposite: #owner
10  type: Subscription multiplicity: #many )
11 Passport>>metamodelExpireDate
12  ^ (EMOF.Property name: #expireDate type: Date)
13 Passport>>metamodelNumber
14  ^ (EMOF.Property name: #number type: Number)
15 Passport>>metamodelOwner
16  ^ (EMOF.Property name: #owner
17  opposite: #passport type: Person)
  
```

Once we defined the meta-description—as shown in the code snippet—we can generate the GLORP descriptor, read and write object instances of the meta-model from and to the database as previously described. Figure B.4 shows a screenshot of some database tables automatically generated and populated.

Person						
dbid	school	studentid	passport	age	name	class
1	UZH	2	4	22	Anna Cazzulani	Student
2	USI	3	5	20	Giorgio Tiepoli	Student
3	NULL	NULL	1	27	Peppe Castiglia	Person
4	NULL	NULL	6	28	Michele Vaaanzo	Person
5	NULL	NULL	2	31	Jhonny Bravo	Person
6	Politecnico	1	3	18	Carmelo Varicella	Student

PersonExamLink			StudentId			Exam	
dbid	personid	examid	dbid	id	student	dbid	name
1	2	2	1	1	6	1	Calculus
2	1	3	2	3	1	2	Algebra
3	6	4	3	2	2	3	Greek
4	2	4				4	Physics

Figure B.4. Examples of generated and populated database tables

B.4 Summary

The Meta-base provides flexibility and persistence to any meta-model described according to the EMOF specification. It supports interoperability, as models—or just model entities—can be exchanged through a database.

The Meta-base takes as input a meta-model description and automatically generates the object persistence descriptor, *i.e.*, the mapping between the objects (instances of the meta-model) and the generated database tables. The Meta-base manages direct, one-to-one, one-to-many and many-to-many relationships among meta-model entities. It also supports inheritance between meta-model classes by means of filtered inheritance.

Bibliography

- [AADP⁺08] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2008)*, pages 304–318. ACM, 2008.
- [ABGM99] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus. Using version control data to evaluate the impact of software tools. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 324–333. ACM, 1999.
- [ACC⁺02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
- [Ada09] Bram Adams. Co-evolution of source code and the build system. In *PhD Symposium at the 25th IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 461–464. IEEE CS, 2009.
- [AHM06] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 361–370. ACM Press, 2006.
- [APGP05] Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, and Martin Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. *Electronic Notes in Theoretical Computer Science*, 127(3):87–99, 2005.
- [APM04] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE 2004)*, pages 31–40. IEEE CS, 2004.
- [Art88] Lowell Jay Arthur. *Software Evolution: The Software Maintenance Challenge*. John Wiley and Sons, 1988.
- [AV09] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 298–308. IEEE CS Press, 2009.

- [BAB⁺93] David M. Butler, James C. Almond, R. Daniel Bergeron, Ken W. Brodlie, and Robert B. Haber. Visualization reference models. In *Proceedings of the 4th Conference on Visualization (VIS 1993)*, pages 337–342. IEEE Computer Society, 1993.
- [BAHS97] Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk. In *Proceedings of the ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.
- [BB09] Adrian Bachmann and Abraham Bernstein. Data retrieval, processing and linking for software process data analysis. Technical report, University of Zurich, Department of Informatics, 2009.
- [BBA⁺09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, pages 121–130. ACM, 2009.
- [BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [BDL10] Alberto Bacchelli, Marco D’Ambros, and Michele Lanza. Are popular classes more defect prone? In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, pages 59–73. ARCoSS LNCS Springer, 2010.
- [BDLR09] Alberto Bacchelli, Marco D’Ambros, Michele Lanza, and Romain Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of the 16th IEEE Working Conference on Reverse Engineering (WCRE 2009)*, pages 205–214. IEEE CS Press, 2009.
- [BDS⁺00] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. Scrum: An extension pattern language for hyperproductive software development, 2000.
- [BDW99] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
- [BE96] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.
- [BEJWKG05] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 177–186. ACM, 2005.

- [BEP07] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 11–18. IEEE CS Press, September 2007.
- [BGA06] Salah Bouktif, Yann-Gael Gueheneuc, and Giuliano Antoniol. Extracting change-patterns from CVS repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 221–230. IEEE Computer Society, 2006.
- [BGD⁺06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 137–143. ACM, 2006.
- [BGD⁺07] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, pages 6–13. IEEE Computer Society, 2007.
- [BH06] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 199–210. IEEE Computer Society, 2006.
- [BJS⁺08] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE 2008)*, pages 308–318. ACM, November 2008.
- [Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: Improving cooperation between developers and users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work (CSCW 2010)*, pages 301–310. ACM, 2010.
- [BPZK08] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 337–345. IEEE CS, September 2008.
- [BR00] Keith H. Bennett and Vaclav T. Rajlich. *The Future of Software Engineering*, chapter Software Maintenance and Evolution: A Roadmap, pages 75–87. ACM Press, 2000.
- [BS98] Aaron B. Binkley and Stephen R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 452–455. IEEE Computer Society, 1998.

- [BZ06] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 2006)*, pages 221–230. IEEE Computer Society, 2006.
- [CAT07] Fanny Chevalier, David Auber, and Alexandru Telea. Structural analysis and visualization of c++ code evolution using syntax trees. In *Proceedings of the Ninth International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 90–97. ACM, 2007.
- [CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [CKN⁺03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis 2003)*, pages 77–86. ACM, 2003.
- [CLL99] Peter Coad, Jeff de Luca, and Eric Lefebvre. *Java Modeling Color with Uml: Enterprise Components and Process with Cdrom*. Prentice Hall PTR, 1999.
- [Coc01] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001.
- [Cor89] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.
- [CZH⁺08] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.
- [CZVRvD09] Bas Cornelissen, Andy Zaidman, Bart Van Rompaey, and Arie van Deursen. Trace visualization for program comprehension: A controlled experiment. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC 2009)*, pages 100–109. IEEE Computer Society, 2009.
- [Dav95] Alan Mark Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.
- [DB08] Annette J. Dobson and Adrian Barnett. *An Introduction to Generalized Linear Models, Third Edition*. Chapman & Hall/CRC Texts in Statistical Science, 2008.
- [DBL10] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 23–31. IEEE CS Press, 2010.
- [DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006)*, pages 203–212. IEEE Computer Society, 2006.
- [DGLP08] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. Analyzing software repositories to understand software evolution. In *Software Evolution*, chapter 3, pages 37–67. Springer, 2008.

- [DGN05] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: An agile reengineering environment. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 99–102. ACM, 2005. Tool demo.
- [DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.
- [DL06a] Marco D’Ambros and Michele Lanza. Applying the evolution radar to postgresql. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 177–178. ACM, 2006.
- [DL06b] Marco D’Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 189–198. IEEE CS Press, 2006.
- [DL06c] Marco D’Ambros and Michele Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 227–236. IEEE CS Press, 2006.
- [DL07] Marco D’Ambros and Michele Lanza. Bugcrawler: Visualizing evolving software systems. In *Proceedings of the 11th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 333–334. IEEE CS Press, 2007. Tool demo.
- [DL08a] Marco D’Ambros and Michele Lanza. Churrasco: Supporting collaborative software evolution analysis. In *Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.
- [DL08b] Marco D’Ambros and Michele Lanza. A flexible framework to support collaborative software evolution analysis. In *Proceedings of the 12th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 3–12. IEEE CS Press, 2008.
- [DL09] Marco D’Ambros and Michele Lanza. Visual software evolution reconstruction. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(3):217–232, May 2009.
- [DL10] Marco D’Ambros and Michele Lanza. Distributed and collaborative software evolution analysis with Churrasco. *Journal of Science of Computer Programming*, 75(4):276–287, 2010.
- [DLL06] Marco D’Ambros, Michele Lanza, and Mircea Lungu. The Evolution Radar: Visualizing integrated logical coupling information. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 26–32. ACM, 2006.

- [DLL09] Marco D’Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the Evolution Radar. *IEEE Transactions on Software Engineering*, 35(5):720 – 735, 2009.
- [DLLR09] Marco D’Ambros, Mircea Lungu, Michele Lanza, and Romain Robbes. Promises and perils of porting software visualization tools to the web. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution (WSE 2009)*, pages 109–118. IEEE CS Press, 2009.
- [DLLR10] Marco D’Ambros, Michele Lanza, Mircea Lungu, and Romain Robbes. On porting software visualization tools to the web. *International Journal on Software Tools for Technology Transfer*, 2010. To appear.
- [DLP07a] Marco D’Ambros, Michele Lanza, and Martin Pinzger. “A Bug’s Life” — Visualizing a bug database. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007)*, pages 113–120. IEEE CS Press, 2007.
- [DLP07b] Marco D’Ambros, Michele Lanza, and Martin Pinzger. The Metabase: Generating object persistency using meta descriptions. In *Proceedings of the 1st Workshop on FAMIX and Moose in Reengineering (FAMOOSR 2007)*, 2007.
- [DLR05] Stéphane Ducasse, Michele Lanza, and Romain Robbes. Multi-level method understanding using microprints. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 33–38. IEEE Computer Society, 2005.
- [DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.
- [DLR09] Marco D’Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proceedings of the 16th IEEE Working Conference on Reverse Engineering (WCRE 2009)*, pages 135–144. IEEE CS Press, 2009.
- [DLR10] Marco D’Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–40. IEEE CS Press, 2010.
- [DPS⁺07] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *Proceedings of the 23th IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 94–103. IEEE CS Press, 2007.
- [DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.
- [ELH⁺05] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering

- research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, 2005.
- [EMM01] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001.
- [Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
- [ESS92] Stephen G. Eick, Joseph L. Steffen, and Eric E. Jr. Sumner. SeeSoft – A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.
- [EZS⁺08] Marc Eaddy, Thomas Zimmermann, Kaitin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transaction on Software Engineering*, 34(4):497–515, 2008.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [FD04] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 387–396. IEEE Computer Society, 2004.
- [FG04] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, 2004.
- [FG06] Michael Fischer and Harald C. Gall. Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 179–188. IEEE Computer Society, 2006.
- [FH83] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48*, 1983.
- [FHK⁺97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
- [FO00] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.
- [FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM 2003)*, pages 23–32. IEEE Computer Society Press, 2003.

- [Fro07] Randall Frost. Jazz and the eclipse way of collaboration. *IEEE Software*, 24(6):114–117, 2007.
- [Gô5] Tudor Gîrba. *Modeling history to understand software evolution*. PhD thesis, University of Berne, Berne, 2005.
- [GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.
- [GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday’s Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 40–49. IEEE Computer Society, 2004.
- [GFS05] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
- [GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the 14th IEEE International Conference on Software Maintenance (ICSM 1998)*, pages 190–198. IEEE Computer Society Press, 1998.
- [GHJ04] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 336–341. ACM Press, 2004.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Gil77] Tom Gilb. *Software Metrics*. Winthrop, 1977.
- [Gil81] Tom Gilb. Evolutionary development. *SIGSOFT Software Engineering Notes*, 6(2):17–17, 1981.
- [GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23. IEEE Computer Society Press, 2003.
- [GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM 1999)*, pages 99–108. IEEE CS Press, 1999.
- [GKMS00] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2):653–661, 2000.

- [GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE CS Press, 2005.
- [GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of the 9th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 2–11. IEEE CS Press, 2005.
- [GLW06] Orla Greevy, Michele Lanza, and Christoph Wyseier. Visualizing live software systems in 3D. In *Proceedings of the 3rd ACM Symposium on Software Visualization (SoftVis 2006)*, pages 47–56. ACM, 2006.
- [GM98] Todd L. Graves and Audris Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Symposium on Software Metrics (METRICS 1998)*, pages 267–273. IEEE Computer Society, 1998.
- [GW05] Carsten Gorg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, pages 205–214. IEEE Computer Society, 2005.
- [Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 78–88. IEEE Computer Society, 2009.
- [HB08] Reid Holmes and Andrew Begel. Deep Intellisense: A tool for rehydrating evaporated information. In *Proceedings of the 5th Working Conference on Mining Software Repositories*, pages 23–26. ACM, 2008.
- [HEDK06] Christine A. Halverson, Jason B. Ellis, Catalina Danis, and Wendy A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW 2006)*, pages 39–48. ACM Press, 2006.
- [HH05] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 263–272. IEEE Computer Society, 2005.
- [HHM04] Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. MSR 2004: International workshop on mining software repositories. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 770–771. IEEE Computer Society, 2004.
- [HJK⁺07] Abram Hindle, Zhen Ming Jiang, Walid Koneilat, Michael W. Godfrey, and Richard C. Holt. YARN: Animating software evolution. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007)*, pages 129–136. IEEE CS Press, 2007.
- [HLG08] Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey, editors. *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), 2008, Proceedings*. ACM, 2008.

- [HMHJ05] Ahmed E. Hassan, Audris Mockus, Richard C. Holt, and Philip M. Johnson. Guest editor's introduction: Special issue on mining software repositories. *IEEE Transactions on Software Engineering*, 31:426–428, 2005.
- [Hol06] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.
- [HW07] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43, New York, NY, USA, 2007. ACM.
- [Jac03] E. J. Jackson. *A Users Guide to Principal Components*. John Wiley & Sons Inc., 2003.
- [Jaz02] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.
- [JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 467–477. ACM, 2002.
- [JKZ09] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE 2009)*, pages 111–120. ACM, 2009.
- [JPZ08] Sascha Just, Rahul Premraj, and Thomas Zimmermann. Towards the next generation of bug tracking systems. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2008)*, pages 82–85. IEEE Computer Society, 2008.
- [KAG⁺96] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE 1996)*, pages 364–371. IEEE Computer Society, 1996.
- [KBT07] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining software repositories with iSPARQL and a software evolution ontology. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, pages 10–17. IEEE Computer Society, 2007.
- [KC98] R. Kazman and S. J. Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse (ICSR 1998)*, pages 290–299. IEEE Computer Society, 1998.
- [KLN08] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 209–218. IEEE Computer Society, 2008.

- [KM05] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 159–168. ACM, 2005.
- [KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2006/FSE-14)*, pages 1–11. ACM, 2006.
- [KM08] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 301–310. ACM, 2008.
- [KM10] Holger M. Kienle and Hausi A. Müller. Rigi—an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75:247–263, 2010.
- [KMN08] Jens Knodel, Dirk Muthig, and Matthias Naab. An experiment on the role of graphical elements in architecture visualization. *Empirical Software Engineering*, 13(6):693–726, 2008.
- [KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 279–294. IEEE Computer Society, 2006.
- [KN06] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 58–64. ACM, 2006.
- [Kni00] Alan Knight. GLORP: Generic lightweight object-relational persistence. In *Addendum to the Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, pages 173–174. ACM Press, 2000.
- [KPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009)*, pages 75–84, 2009.
- [KSS02] Ralf Kollmann, Petri Selonen, and Eleni Stroulia. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 22–32. IEEE Computer Society, 2002.
- [KZK⁺06] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Girba, Martin Pinzger, James Whitehead, and Andreas Zeller. TA-RE: An exchange language for mining software repositories. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 22–25. ACM, 2006.

- [KZWZ07] Sunghun Kim, Thomas Zimmermann, James Whitehead, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, pages 489–498. IEEE CS, 2007.
- [Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE 2001)*, pages 37–42. ACM Press, 2001.
- [Lan03] Michele Lanza. *Object-Oriented reverse engineering — Coarse-grained, fine-grained, and evolutionary software visualization*. PhD thesis, University of Berne, 2003.
- [LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.
- [LD03] Michele Lanza and Stéphane Ducasse. Polymetric views — A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.
- [LDGP05] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. Codecrawler — An information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 672–673. ACM Press, 2005. Tool demo.
- [Leh80a] Meir M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.
- [Leh80b] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.
- [LFGT09] Andrea De Lucia, Fausto Fasano, Claudia Grieco, and Genny Tortora. Recovering design rationale from email repositories. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 543–546. IEEE CS Press, 2009.
- [LHS05] Paul Luo Li, Jim Herbsleb, and Mary Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: A case study of OpenBSD. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, pages 32–41. IEEE Computer Society, 2005.
- [Lik32] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.
- [LL06] Mircea Lungu and Michele Lanza. Softwarentaut: Exploring hierarchical system decompositions. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 349–350. IEEE CS Press, 2006. Tool demo.
- [LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *Proceedings of the 11th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 91–100. IEEE CS Press, 2007.

- [LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 183–192. IEEE CS Press, 2006.
- [LLGH07] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Reinout Heeck. Reverse engineering super-repositories. In *Proceedings of the 14th IEEE Working Conference on Reverse Engineering (WCRE 2007)*, pages 120–129. IEEE CS Press, 2007.
- [LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
- [LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: Experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis 2003)*, pages 47–56. ACM, 2003.
- [MÖ9] Mika V. Mäntylä. *Software evolvability – Empirically discovered evolvability issues and human evaluations*. PhD thesis, Helsinki University of Technology, Helsinki, 2009.
- [Mĭ0] Mika V. Mäntylä. Empirical software evolvability – Code smells and human evaluations. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*. IEEE CS Press, 2010.
- [Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.
- [Mar02] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.
- [Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 350–359. IEEE CS Press, 2004.
- [McC95] Jim McCarthy. *Dynamics of software development*. Microsoft Press, 1995.
- [Men10] Thilo Mende. Replication of defect prediction studies: Problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering (PROMISE 2010)*, pages 1–10. ACM, 2010.
- [MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.
- [MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis 2003)*, pages 27–36. ACM, 2003.
- [MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *Proceedings of the 3rd International ACM Symposium on Software Visualization (Softvis 2006)*, pages 135–144. ACM Press, 2006.

- [Mil76] H.D. Mills. Software development. *IEEE Transactions on Software Engineering*, 2:265–273, 1976.
- [MJ82] Daniel D. McCracken and Michael A. Jackson. Life cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7(2):29–32, 1982.
- [MK88] Hausi. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 1988)*, pages 80–86. IEEE Computer Society Press, 1988.
- [MK10] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Proceeding of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 109–118. IEEE Computer Society, 2010.
- [ML06] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.
- [MPF08] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.
- [MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 181–190. ACM, 2008.
- [MSC⁺01] Spiros Mancoridis, Timothy S. Souder, Yih-Farn Chen, Emden R. Gansner, and Jeffrey L. Korn. REportal: A web-based portal site for reverse engineering. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 221–230. IEEE Computer Society, 2001.
- [NAH10] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE 2010)*, 2010. To appear.
- [NB05a] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 580–586. ACM, 2005.
- [NB05b] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 284–292. ACM, 2005.
- [NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 452–461. ACM, May 2006.
- [NEFZ00] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Andrea Zisman. BOX: Browsing objects in XML. *Software Practice and Experience*, 30(15):1661–1676, 2000.

- [Nie04] Oscar Nierstrasz. Putting change at the center of the software process. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2004.
- [NR69] P. Naur and B. Randell. *Software Engineering*. NATO, Scientific Affairs Division, Brussels, 1969.
- [NZHZ07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 529–540. ACM, 2007.
- [OA96] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.
- [OL90] Paul W. Oman and Ted G. Lewis. *Milestones in Software Evolution*. IEEE Computer Society Pres, 1990.
- [OS01] Liam O’Brien and Christoph Stoermer. Architecture reconstruction case study. Technical report, CMU/SEI-2001-TR-026, 2001.
- [OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Software Engineering Notes*, 27(4):55–64, 2002.
- [OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 86–96. ACM, 2004.
- [OWB05] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
- [OWB07] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Automating algorithms for the identification of fault-prone files. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 219–227. ACM, 2007.
- [Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 279–287. IEEE Computer Society Press, 1994.
- [PBD08] David Pattison, Christian Bird, and Premkumar Devanbu. Talk and work: A preliminary report. In *Proceedings of the 5th International Working Conference on Mining Software Repositories (MSR 2008)*, pages 113–116. ACM, 2008.
- [PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

- [PC86] David Lorge Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986.
- [PGF05] Martin Pinzger, Harald Gall, and Michael Fischer. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196, 2005.
- [PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2nd ACM Symposium on Software Visualization (SoftVis 2005)*, pages 67–75. ACM, 2005.
- [Pin97] Steven Pinker. *How the Mind Works*. W. W. Norton, 1997.
- [PNM08] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE-16)*, pages 2–12. ACM, 2008.
- [PP05] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511 – 526, 2005.
- [Pri07] Marco Primi. The episode framework - Exporting visualization tools to the web. Bachelor's thesis, University of Lugano, 2007.
- [RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
- [RDGM04] Daniel Rațiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 223–232. IEEE Computer Society, 2004.
- [RdMF02] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175, 2002.
- [RFG05] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. In *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR 2005)*, pages 1–5. ACM Press, 2005.
- [RH07] Peter C. Rigby and Ahmed E. Hassan. What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, pages 23–30. IEEE Computer Society, 2007.
- [Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.
- [RL05] Romain Robbes and Michele Lanza. Versioning systems for evolution research. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 155–164. IEEE CS Press, 2005.

- [Rob08] Romain Robbes. *Of change and software*. PhD thesis, University of Lugano, Switzerland, 2008.
- [Rob10] Gregorio Robles. Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings. In *Proceedings of the 7th International Working Conference on Mining Software Repositories (MSR 2010)*, pages 171–180. IEEE CS Press, 2010.
- [Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.
- [Roy70] Walker W. Royce. Managing the development of large software systems: Concepts and techniques. *Proceedings of the IEEE WESTCON*, pages 1–9, 1970. Reprinted in *Proceedings of the 9th International Conference on Software Engineering (ICSE 1987)*, pages 328–338. IEEE CS Press, 1987.
- [SDBE98] John Stasko, John Domingue, Marc Brown, and Blaine Price (Eds.). *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.
- [SK03] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
- [SLT06] Mazeiar Salehie, Shimin Li, and Ladan Tahvildari. A metric-based heuristic framework to detect object-oriented design flaws. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, pages 159–168. IEEE Computer Society, 2006.
- [SM95] M.-A. D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 11th IEEE International Conference on Software Maintenance (ICSM 1995)*, pages 275–284. IEEE Computer Society, 1995.
- [SMWH09] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 23–33. IEEE Computer Society, 2009.
- [Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.
- [SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [Sta84] Thomas A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.
- [Sta99] Jennifer Stapleton. DSDM: Dynamic systems development method. *International Conference on Technology of Object-Oriented Languages*, 0:406, 1999.
- [Sto00] Team development with visualworks. cincom technical white paper, 2000. Cincom Technical Whitepaper.

- [SW08] Mark Sherriff and Laurie Williams. Empirical software change impact analysis using singular value decomposition. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 268–277. IEEE Computer Society, 2008.
- [SZ00] John Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. *IEEE Symposium on Information Visualization*, 0:57–65, 2000.
- [SZZ06] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE 2006)*, pages 18–27. ACM, 2006.
- [TA08] Alexandru Telea and David Auber. Code Flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, 2008.
- [TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX: Exchange experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, 2000.
- [TG02] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 127–136. IEEE Computer Society, 2002.
- [Tic82] Walter F. Tichy. Design, implementation, and evaluation of a Revision Control System. In *Proceedings of the 6th International Conference on Software Engineering (ICSE 1982)*, pages 58–67. IEEE Computer Society Press, 1982.
- [TLTC05] M. Termeer, C. F. J. Lange, A. Telea, and M. R. V. Chaudron. Visual exploration of combined architectural and metric information. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VIS-SOFT 2005)*, pages 1–6. IEEE Computer Society, 2005.
- [TM02] Christopher Taylor and Malcolm Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis (VIS-SOFT 2002)*, pages 43–50. IEEE Computer Society, 2002.
- [Tri06] Mario Triola. *Elementary Statistics*. Addison-Wesley, 2006.
- [TSG04] Adrian Trifu, Olaf Seng, and Thomas Genssler. Automated design flaw correction in object-oriented systems. In *Proceedings of the 8th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 174–183. IEEE Computer Society, 2004.
- [uM04] Davor Čubranić and Gail C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 92–97, 2004.
- [uMSB05] Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

- [VRD04] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 328–337. IEEE Computer Society Press, 2004.
- [VRRD06] Filip Van Rysselberghe, Matthias Rieger, and Serge Demeyer. Detecting move operations in versioning information. In *Proceedings of the 10th IEEE European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 271–278. IEEE Computer Society, 2006.
- [VT06a] Lucian Voinea and Alexandru Telea. How do changes in buggy Mozilla files propagate? In *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis 2006)*, pages 147–148. ACM, 2006.
- [VT06b] Lucian Voinea and Alexandru Telea. Multiscale and multivariate visualizations of software evolution. In *Proceedings of the 2006 ACM Symposium on Software Visualization (SoftVis 2006)*, pages 115–124. ACM, 2006.
- [VT06c] Lucian Voinea and Alexandru Telea. An open framework for CVS repository querying, analysis and visualization. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 33–39. ACM, 2006.
- [VT07] Lucian Voinea and Alexandru Telea. Visual data mining and analysis of software repositories. *Computers & Graphics*, 31(3):410–428, 2007.
- [VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis 2005)*, pages 47–56. ACM, 2005.
- [WHH04] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89. IEEE CS Press, 2004.
- [WL07a] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007)*, pages 231–240. IEEE CS Press, 2007.
- [WL07b] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007)*, pages 92–99. IEEE CS Press, 2007.
- [WL08a] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th ACM/IEEE International Conference on Software Engineering (ICSE Companion 2008)*, pages 921–922. ACM, 2008. Tool demo.
- [WL08b] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM International Symposium on Software Visualization (Softvis 2008)*, pages 155–164. ACM Press, 2008.
- [WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, pages 1–8. IEEE Computer Society, 2007.

- [WSDN09] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 1–11. IEEE Computer Society, 2009.
- [WZX⁺08] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 461–470. ACM, 2008.
- [YCM78] S. S. Yau, J. S. Colofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of the 2nd IEEE International Conference on Computer Software and Applications (COMPSAC 1978)*, pages 60–65. IEEE Computer Society Press, 1978.
- [YMNCC04] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):573–586, 2004.
- [ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 531–540, 2008.
- [ZNG⁺09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, New York, NY, USA, 2009. ACM.
- [ZPSB09] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. Improving bug tracking systems. In *Companion to the 31th International Conference on Software Engineering (ICSE Companion 2009)*, pages 247–250. IEEE Computer Society, 2009.
- [ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (PROMISE 2007)*, pages 9–15. IEEE Computer Society, 2007.
- [ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press.
- [ZWDZ05] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.