Software Systems as Cities

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
Richard Wettel

under the supervision of
Prof. Michele Lanza

September 2010

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Cesare Pautasso	Università della Svizzera Italiana, Switzerland
Prof. Rainer Koschke	University of Bremen, Germany
Prof. André van der Hoek	University of California, Irvine, USA

Dissertation accepted on 21 September 2010

Prof. 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.

Richard Wettel
Lugano, 21 September 2010

Abstract

Software understanding takes up a large share of the total cost of a software system. The high costs attributed to software understanding activities are caused by the size and complexity of software systems, by the continuous evolution that these systems are subject to, and by the lack of physical presence which makes software intangible. Reverse engineering helps practitioners deal with the intrinsic complexity of software, by providing a broad range of patterns and techniques. One of these techniques is software visualization, which makes software more tangible, by providing visible representations of software systems.

Interpreting a visualization is by no means trivial and requires knowledge about the visual language of the visualization. One means to ease the learning of a new visualization's language are metaphors, which allow the interpretation of new data representations by analogy. Possibly one of the most popular metaphors for software visualization is the city metaphor, which has been explored in the past by a number of researchers. However, in spite of the efforts, the value of this metaphor for reverse engineering has never been taken beyond anecdotical evidence.

In this dissertation, we demonstrate the value of the city metaphor for reverse engineering along two directions. On the one hand, we show that the metaphor is versatile enough to allow the representation of different facets of software. On the other hand, we show that the city metaphor enables the creation of software visualizations which efficiently and effectively support reverse engineering activities.

Our interpretation of the city metaphor at its core depicts the system as a city, the packages as districts, and the classes as buildings. The resulting "code city" visualization provides a structural overview of the software system, enriched with contextual data. To be able to perform analyses of real systems using our approach, we implemented a tool called CodeCity.

We demonstrate the versatility of the metaphor, by using it in three different analysis contexts, i.e., program comprehension, software evolution analysis, and software design quality assessment. For each of the contexts, we describe the visualization techniques we employ to encode the contextual data in the visualization and we illustrate the application by means of case studies. The insights gained in the three analysis contexts are complementary to each other, leading to an increasingly more complete "big picture" of the systems.

We then demonstrate how the visualizations built on top of our city metaphor effectively and efficiently support reverse engineering activities, by means of an extensive controlled experiment. The design of our experiment is based on a list of desiderata that we extracted from our survey of the current body of research. We conducted the experiment over a period of six months, in four sites located in three countries, with a heterogeneous sample of subjects composed of fair shares of both academics and industry practitioners. The main result of our experiment was that, overall, our approach outperforms the state-of-practice in supporting users solve reverse engineering tasks, in terms of both correctness and completion time.

Acknowledgements

I am grateful to Michele Lanza for being my advisor. Michele, it has been a privilege to work with you. Apart from being a mentor, you have been there for me in so many ways. I hope you will remain part of my life in the years to come.

I thank Matthias Hauswirth, Cesare Pautasso, Rainer Koschke, and André van der Hoek, for accepting to be in my dissertation committee. Your thorough reviews allowed me to improve this dissertation sensibly. It was great to have you all in Lugano for my defense.

I am lucky to have worked here with my awesome office mates: Romain Robbes, Mircea Lungu, Marco D'Ambros, Lile Hattori, Fernando Olivero, and Alberto Bacchelli. Romain, you're way too cool ^ ^ for words... Mircea, thanks for showing me the proverbial Romanian hospitality so far away from home. I enjoyed having you around. Marco, I will miss our daily exchange of looks, jokes, and nonsensical discussions. Lile, your strength and tenacity is the living proof that it's not a men's world we're living in. Fernando, it was great having an office mate who shared my passion for 3D "bisualization". Speaking of which, thank you for so graciously accepting all my jokes, even the bad ones. Alberto, thanks for all your unconditional support and for introducing me to the *bolognese* food. One rarely meets a person who is so smart, hard-working, reliable, and yet so humble. I will keep a dear memory of all of you and of the fun we had together inside and outside our office.

I thank Radu Marinescu, whose early guidance, passion for science, and friendship made all these possible.

I thank Tudor Gîrba for showing me new perspectives of my work during so many fruitful discussions. Your enthusiasm for research is contagious, Doru.

I thank Daniel Raţiu for offering to review my dissertation, in spite of his scarce spare time. Dani, your pragmatic feedback was extremely useful.

A special thanks goes to the numerous people involved in the experiment. To Radu Marinescu, Mircea Lungu, Alberto Bacchelli, Lile Hattori, and Romain Robbes for helping me out with the design of the experiment. To Oscar Nierstrasz, Serge Demeyer, Fabrizio Perin, Quinten Soetens, Alberto Bacchelli, and Sebastiano Cobianco for their help in organizing the experimental runs. Last, but not least, to all the subjects of our experiment: the developers in Bologna and Lugano, the Software Composition Group in Bern, and the Master students in Lugano and Antwerp. Your valuable contribution made my dissertation much stronger.

I would like to thank my mother for all the sacrifices she has made while raising me and my father for the priceless life lessons he has offered me.

To my better half, Simy. You are the meaning of my life.

Contents

Figures

Tables

Part I

Prologue

Chapter 1

Introduction

"*Software is hard*", remarked Donald Knuth[Knu02], one of the most respected researchers in computer science. Software has been defined as the set of computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system [IEE90]. Building well-crafted software is hampered both by external factors (i.e., changes in hardware and software, changing requirements, time pressure) and by internal characteristics related to the nature of software—an aspect we address later in more detail.

The consequences of these adverse conditions can sometimes be critical, e.g., the introduction of software defects, due to the time pressure. Finding ways to deal with such problems is the object of software engineering, which is defined as the application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [IEE90].

Due to the aforementioned external factors, namely a changing environment which triggers new requirements on the system, even the best design degrades over time, leading to a phenomenon aptly termed as "architectural drift" [Pin05], "design erosion" [vGB02], or "code decay" [EGK$^+$01]. This problem is addressed by reengineering [CCI90], which aims at improving the design of parts of the system to make it more capable of embracing future changes [Bec00].

A preliminary step towards modifying a software system is understanding the software system, which remains an open research challenge, in spite of the existing solutions. One solution is reverse engineering, which is the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction [CCI90].

A subdomain of software reverse engineering is software visualization, which has been defined by Price et al. as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [PBS93].

With our approach, we address the understanding of software systems, which represents a daunting and costly activity. Software maintenance claims a share estimated to 60%–90% of the total software costs [ZSG79, LS81, McK84, Erl00]. A significant part of the maintenance effort is spent on software understanding [Cor89].

Understanding a software system requires obtaining a mental model of the system, similar to the one of a system expert, who spends large amounts of time reading and writing its code. Building such a mental model is a tedious activity, due not only to the aforementioned external factors, but also to a number of properties of software, described next.

1.1 The Challenges of Software Understanding

Software is large and complex. The sheer size and complexity of software systems hinders even the comprehension of smaller parts of such a system in isolation, let alone the understanding of the system as a whole. There is a broad range of reverse engineering techniques and patterns that support software understanding, such as the ones proposed by Demeyer et al. [DDN02]: from lightweight techniques with wider focus (e.g., "skim the documentation", "read the code in one hour") to more heavyweight techniques with narrowed focus (e.g., "refactor to understand", "step through the execution"). One problem with these techniques is that they do not approach the system as a whole.

Software evolves. Software systems are subject to modifications over time. According to the second of Lehman's laws of software evolution [LB85], as systems evolve, they grow in complexity, and consequently more resources are needed to preserve and simplify their structure. However, change is not necessarily harmful; the XP (i.e., eXtreme Programming) community is built on the idea that change should not be resisted, but actually embraced [Bec00]. Moreover, the rich evolution data stored in versioning system repositories provides a valuable resource for both retrospective analyses and for trend prediction. As a consequence, the research community devised specific reverse-engineering techniques, such as "learning from the past" [DDN02]. However, from a software understanding perspective, dealing with even more data, due to the analysis of multiple versions of a system, further exacerbates the scalability problem.

Software is intangible. A less obvious impediment in the process of understanding software systems is the virtual nature of software—a man-made product aptly described by Ball and Eick as "intangible, having no physical shape or size" [BE96]. In his book on visual perception, Ware affirms that humans acquire more information through vision than through all the other senses combined [War04]. Consequently, software understanding is severely hindered by the lack of visual presence that characterizes software.

Software visualization is able to increase the perceived tangibility of software systems through intuitive visual representations. Over the last two decades, software visualization has earned a place among the most effective program comprehension techniques, widely used in the context of software maintenance, reverse engineering and re-engineering [Kos03].

1.2 Reverse Engineering with Software Visualization

The wide variety of software visualization approaches over the last two decades led to a plethora of visualizations, documented in several compendia [SDBP98, Zha03, GME05, Die07] and classifiable according to several taxonomies [Mye86, PBS93, RC93, MMC02].

Each visualization targets one or more of the many aspects of a software system and encodes information according to its own visual language. Performing an analysis of several aspects of a software system (e.g., design and evolution) would require conducting separate analyses for each targeted aspects, using a different visualization. Learning a new language for each new visualization may lead to a cognitive overload of its users, which would defeat the very purpose of visualization. A promising means to reduce the cognitive load are metaphors.

1.3 Metaphor-Based Visualization

According to the conceptual metaphor theory developed in the field of cognitive linguistics [LJ80], a metaphor is a stable and systematic relationship between two conceptual domains, i.e., *source* and *target*. Projecting the structure of the source domain onto the target domain, enables one to express the target in terms of the source. By analogy, one is able to understand the novel in terms of the familiar or, in the case of software visualization, the invisible in terms of the visible. Research has shown that metaphors, far from being just figures of speech, are paramount to everyday communication and learning [LJ80, LN00].

To compensate the intangibility of software [BE96], an abundance of metaphors have populated the software engineering field. Simple metaphors, initially used to explain the abstract concepts, ended up being assimilated by the terminology, e.g., *tree* structure, data *stream*, data *flow*, class *inheritance*, software *tool*, *waterfall* model. Other metaphors illustrate complex processes, from a more philosophical perspective, e.g., the life cycle of a software system is *like a river* [Boo09].

There are metaphors that go beyond words and build on the idea that “a picture is worth a thousand words”. Visualization has the potential of reinforcing metaphors, making them more intuitive and memorable. Therefore, software visualization is a fertile ground for the study of software-related metaphors. Over the last decade, the availability of 3D graphics enabled the appearance of more realistic and easier to grasp visual metaphors, such as landscapes [BNDL04], cities [KM00, PBG03], or solar systems [GYB04].

We conducted our research in the context of the EvoSpaces[1] project, which aimed at exploiting multi-dimensional navigation spaces to visually explore object-oriented, evolving software systems [LGD09]. In the first phase of the project, Boccuzzo and Gall experimented with various metaphors taken from everyday life [BG07]. However, none of them appeared as promising as the city metaphor. Apart from a structural complexity which fits the inherent complexity of large-scale software, the city’s notions of locality and habitability [Gab96] help avoid disorientation in the 3D environment. For these reasons, we adopted the city as the central metaphor of the project.

Another reason why the city metaphor has been one of the most popular metaphors used for the visualization of software systems is the large number of similarities between the software and the urban domain.

1.4 Software and the City

Conceptually, both cities and software systems are complex, man-made entities. From a life cycle perspective, they are conceived during a *planning phase*, in which *architects* strive to fulfill the *requirements*, while maintaining a balance between *functionality* and *design*. Thereafter, cities and software systems are *built incrementally* and, once in place, they require perpetual *maintenance* to keep their value over time.

The influence of civil architecture on software engineering extends beyond terminology and reaches its best practices: The prominent design patterns by Gamma et al. [GHJV95] are rooted in the architectural pattern language proposed by Alexander et al. [AIS77] almost two decades before.

[1] `http://www.inf.usi.ch/projects/evospaces`

The city metaphor has been explored by other researchers before us [KM00, PBG03, MFM03, BNDL04, LSP05]. However, the existing approaches are plagued by one or more of the following problems: limited coverage of the various facets of software systems, poor tool support, and most of all, the scarcity of empirical evaluation.

Problem Statement:

> *The current software visualization approaches based on a city metaphor fail at demonstrating the value of this metaphor for reverse engineering.*

We claim that the city metaphor holds an intrinsic value in the context of software visualization for reverse engineering. On the one hand, the similarities between software systems and cities may ease the understanding of the metaphor. On the other hand, the consistent use of one central metaphor to illustrate several facets of software systems may reduce the cognitive load of the metaphor's adopters. However, the potential of the city metaphor has yet to be demonstrated. In this context, we formulate our thesis.

1.5 Our Approach

Thesis:

> *Depicting software systems as cities is a versatile metaphor which enables the creation of efficient software visualizations to support reverse engineering.*

We start our systematic exploration of the city metaphor with the design of the visual language, i.e., the mapping between the source domain (i.e., the city) and target domain (i.e., the software system). The software system is represented as a city, its packages as the city's districts, and its classes as the buildings. We enrich the city visualization with software metrics, i.e., the physical properties of the city artifacts (e.g., color, dimensions) reflect a set of measurable properties of the software artifacts.

To prove the claim of our thesis related to the *versatility* of the metaphor, we applied the metaphor in three distinct contexts, i.e., program comprehension, software evolution analysis, and design quality assessment. We first provided support for program comprehension based on static data relative to a single version of a software system. Then, we extended the approach to support software evolution analysis, by taking into account several versions of the same system. Finally, we added support for the assessment of the quality of a system's design, based on information extracted using detection strategies [Mar04a]. To illustrate our metaphor's versatility, we analyzed several open-source systems of various sizes for each of the three application contexts.

To prove the claim related to the *efficiency* of the software visualizations built on top of the city metaphor, we designed and conducted a controlled experiment aimed at discovering whether, under which circumstances, and to whom our approach is useful. With this experiment, we aimed at comparing our visualization approach, based on the city metaphor, to the state-of-the-practice in program comprehension, in terms of both correctness and completion time in supporting task solving.

To apply our approach on real software systems, we implemented a tool called CodeCity [WL08b], based on the design of our metaphor. We built CodeCity on top of the Moose reengineering framework [NDG05], which allows us to use FAMIX [DTD01], a language-independent meta-model for object-oriented software (e.g., Smalltalk, Java, C/C++, and C#).

1.6 Contributions

In the light of the described thesis, the contributions of this dissertation are:

1. **The definition of a versatile city metaphor for software visualization** [WL07b]. The building blocks of our city metaphor (i.e., domain mapping, concept mapping, property mapping) as well as the set of techniques we used to materialize this metaphor for software visualization (e.g., two novel layout algorithms, a comparison between different mapping strategies) are described and illustrated in Chapter 3.
2. **The application of the city metaphor to program comprehension** [WL07a]. We employed the different visualization techniques implemented in our metaphor to support the understanding of software systems, based on their most recent version. To validate this application, we performed an analysis on two case studies, presented in Chapter 3.
3. **The application of the city metaphor to software evolution** [WL08c]. To enable our approach to support evolution analyses, we extended our metaphor and devised three techniques aimed at exposing historical information at various granularity levels. We present the visualization techniques for software evolutions and the way we validated them by means of three case studies in Chapter 4.
4. **The application of the city metaphor to design quality assessment** [WL08d]. To enable our approach to support the analysis of software design, we devised a visualization technique called disharmony map, which allows one to get an overview of the design problems, i.e., detected violations of design guidelines. The technique and the four case studies we used to validate this application are presented in Chapter 5.
5. **The implementation of a tool which supports our city metaphor** [WL08b, WL08a, Wet08]. We implemented CodeCity, a scalable tool which supports all the three aforementioned applications, is publicly available and has been used in both academic research and industry. We present our tool support in Chapter 6.
6. **The empirical validation of our approach through a repeatable controlled experiment** [WLR10]. Starting from the shortcomings found in the literature, we designed a controlled experiment for the empirical validation of our approach and conducted it with participants from both academia and industry. The design, operation, and analysis of the data coming from this experiment are presented in detail in Chapter 7 and Chapter 8. Moreover, the entire experimental data set and everything that is required to make the experiment repeatable and transparent is presented in Appendix A.

1.7 Roadmap

The remainder of this dissertation is organized as follows.

Part I: Prologue

- In Chapter 2 we describe the software visualization research context from a historical perspective, by focusing on the main contributions in the field along the timeline of the last fifty years and observing the events that triggered the progress of this research field.

Part II: Approach deals with the first claim of our thesis, which refers to the versatility of the city metaphor.

- In Chapter 3, we describe in detail our city metaphor for software visualization and we discuss a number of visualization techniques aimed to support program analyses of single versions of software systems. We illustrate the application of our approach to program comprehension, by means of two case studies.
- In Chapter 4, we describe how we extended our city metaphor for software visualization to enable analyses of software system evolution, by means of three visualization techniques, i.e., age map, time traveling, and timeline. Then, we validate this application of the city metaphor by means of three case studies.
- In Chapter 5, we deal with the third application of the metaphor, i.e., design quality assessment. To this end, we describe a visualization technique called disharmony maps, which enriches our original city visualizations with design problem data. We illustrate this application of our city metaphor by means of four case studies.
- In Chapter 6 we describe CodeCity, the tool that supports our city metaphor and all three applications, i.e., program comprehension, software evolution analysis, and design quality assessment.

Part III: Evaluation deals with the second claim of our thesis, which refers to the efficiency of the visualizations built on top of the city metaphor.

- In Chapter 7 we discuss the design of our controlled experiment, based on a list of desirable features emerged from the study of the body of literature.
- In Chapter 8 we present both the operation and the analysis of the data. First, we describe how we conducted the experiment. Then, we present the analyses we performed on the data collected from our subjects and discuss the results.

Part IV: Epilogue

- In Chapter 9, we reflect over the results of the research we performed, in the context of our thesis. Then, we present the contributions and discuss the differences between expectations and realizations. Before concluding, we delineate several future work directions.

Part V: Appendix

- In Chapter A, we describe all the details concerning the repeatability of our controlled experiment, complementary to Part III. We first present the various questionnaires we used in our experiment for collecting personal data, task solutions, and feedback. Then we provide the oracle sets we used to grade the solutions of the participants to the experimental tasks. Finally, we present the data collected from our subjects.

Part V: Bibliography. The last part lists the bibliography used in this dissertation.

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

Chapter 2

A History of Software Visualization

Price et al. defined software visualization as 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 [PBS93].

According to Diehl, software visualization is concerned with visualizing the structure, behavior and evolution of software [Die07]. Structure visualizations focus on the static parts and relations of a system that can be extracted directly from the source code, without running the program. Behavior focuses on the dynamic information that can be extracted from the execution of the program. Evolution visualization focuses on the changes that a software system is subject to over its lifetime. In this context, our research focuses on both the structure and the evolution of software systems.

Today, software visualization is an established software engineering field, with dedicated venues and a flourishing community around it. However, getting to this state required five decades. In this chapter we present the major contributions to software visualization, in the context of progress triggering factors.

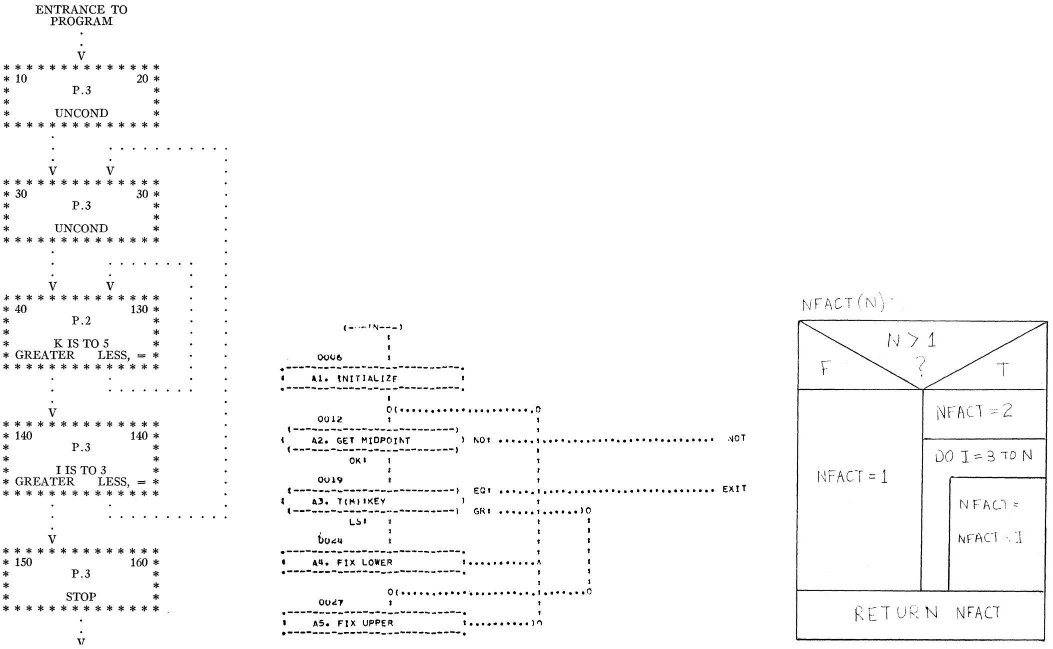
2.1 Foundations of Visualization

The interest in using vision to understand facts has started long time before software came to existence. The first data graphs have been published in 1786 by Playfair [Pla86], a Scottish engineer, considered the founder of graphical methods in statistics, who invented the line graphs, bar charts, and the pie charts.

However, the first theoretical foundation to the information visualization field were placed by Bertin in 1967 with his remarkable *semiology of graphics* [Ber67], in which the French cartographer describes a framework for designing diagrams.

2.2 Pre-1980s

The earliest form of software visualizations were diagrams. Regarded as helpful for program comprehension, diagrams were the object of considerable research effort. In 1959, Haibt created a system able to draw flowcharts (See Figure 2.1(a)) based on programs written in assembly language or Fortran [Hai59]. In 1963, Knuth developed a system which could produce flowcharts (See Figure 2.1(b)) as documentation artifacts [Knu63]. The more structured Nassi-Schneiderman diagrams (See Figure 2.1(c)) appeared in 1973, as an alternative to flowcharts [NS73].



(a) Haibt's flowcharts (b) Knuth's flowcharts (c) Nassi-Shneiderman

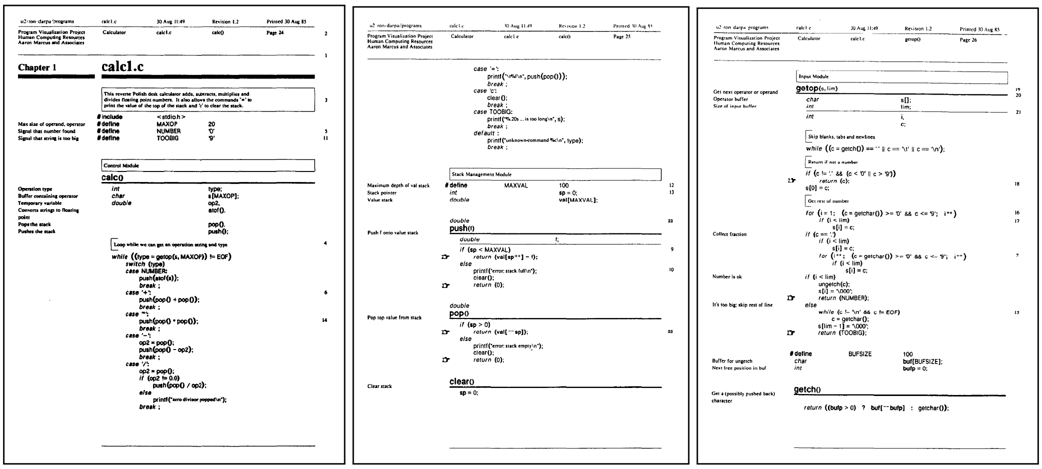
Figure 2.1. The first generation of diagrams

A rather primitive form of software visualization was source code presentation, which addressed the need for more readable representations of programs. The oldest type of source code presentation was pretty-printing, which employed the use of spacing, indentation, and layout to increase code readability. The first pretty-printers appeared in the 1970s and were dedicated to programs written in several programming languages, such as LISP, which included a built-in pretty printer, PL/I [CS70], or Pascal [HL77].

Another precursor of modern visualization was the animation of program behavior, used mainly for educational purposes. The two films of Knowlton from 1966, which demonstrated a low-level programming language developed at Bell Laboratories [Kno66a, Kno66b], represented the first documented use of animation techniques to illustrate program behavior.

2.3 The 1980s

The next step in pretty-printing employed the use of typeface, math notations, and headings, such as the one adopted by the Xerox Cedar community [Tei85]. In a similar vein, Knuth's WEB system [Knu84] combined source code and documentation in a single publication using a markup language, while the SEE Program Visualizer by Baecker and Marcus [BM86, BM89] could typeset a C program according to a style guide based on graphic design principles, and build a "program book" (See Figure 2.2) from a set of C programs.



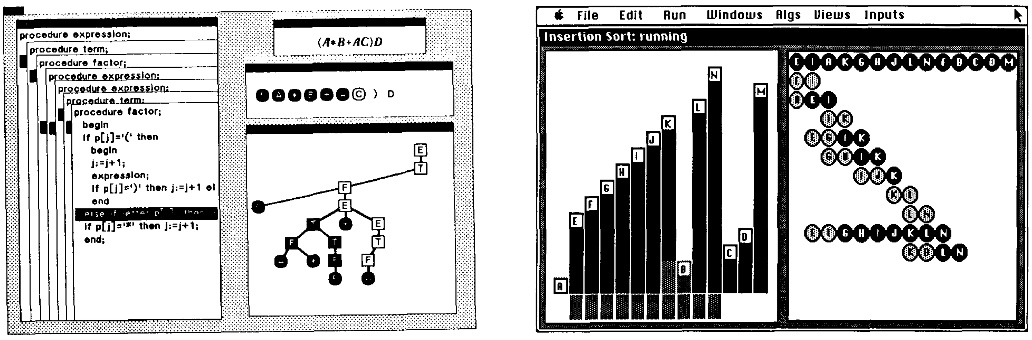
```
u2/ron-darpa/programs         calc1.c      30 Aug 11:49   Revision 1.2   Printed 30 Aug 85
Program Visualization Project Calculator   calc1.c        calc()         Page 24          2
Human Computing Resources
Aaron Marcus and Associates
                                                                                          1
Chapter 1                     calc1.c

                     This reverse Polish desk calculator adds, subtracts, multiplies and
                     divides floating point numbers. It also allows the commands '=' to
                     print the value of the top of the stack and 'c' to clear the stack.   3
                             #include    <stdio.h>
Max size of operand, operator #define     MAXOP          20
Signal that number found      #define     NUMBER         '0'                          5
Signal that string is too big #define     TOOBIG         '9'                          11

                              Control Module
                              calc()
Operation type                  int                      type;
Buffer containing operator      char                     s[MAXOP];
Temporary variable              double                   op2,
Converts strings to floating                             atof(),
point
Pops the stack                                           pop(),
Pushes the stack                                         push();
                                Loop while we can get an operation string and type   4
                                while ((type = getop(s, MAXOP)) != EOF)
                                    switch (type)
                                    case NUMBER:
                                        push(atof(s));
                                        break ;
                                    case '+':                                         6
                                        push(pop() + pop());
                                        break ;
                                    case '*':
                                        push(pop() * pop());                          14
                                        break ;
                                    case '-':
                                        op2 = pop();
                                        push(pop() - op2);
                                        break ;
                                    case '/':
                                        op2 = pop();
                                        if (op2 != 0.0)
                                            push(pop() / op2);
                                        else
                                            printf("zero divisor popped\n");
                                        break ;
```

```
u2/ron/darpa/programs         calc1.c      30 Aug 11:49   Revision 1.2   Printed 30 Aug 85
Program Visualization Project Calculator   calc1.c        calc()         Page 25
Human Computing Resources
Aaron Marcus and Associates

                                    case '=':
                                        printf("\t%f\n", push(pop()));
                                        break ;
                                    case 'c':
                                        clear();
                                        break ;
                                    case TOOBIG:
                                        printf("%.20s ... is too long\n", s);
                                        break ;
                                    default :
                                        printf("unknown-command %c\n", type);
                                        break ;

                              Stack Management Module
Maximum depth of val stack    #define     MAXVAL         100                          12
Stack pointer                 int                        sp = 0;                      13
Value stack                   double                     val[MAXVAL];

                              double                                                  22
Push f onto value stack       push(f)
                                double                   f;
                                if (sp < MAXVAL)                                      9
                           ☞        return (val[sp++] = f);
                                else
                                    printf("error: stack full\n");                    10
                                    clear();
                           ☞        return (0);

                              double
Pop top value from stack      pop()
                                if (sp > 0)
                           ☞        return (val[--sp]);                               23
                                else
                                    printf("error: stack empty\n");
                                    clear();
                           ☞        return (0);

Clear stack                   clear()
                                sp = 0;
```

```
u2/ron/darpa/programs         calc1.c      30 Aug 11:49   Revision 1.2   Printed 30 Aug 85
Program Visualization Project Calculator   calc1.c        getop()        Page 26
Human Computing Resources
Aaron Marcus and Associates

                              Input Module
Get next operator or operand  getop(s, lim)                                           19
Operator buffer                 char                     s[];                         20
Size of input buffer            int                      lim;
                                int                      i,                           21
                                                         c;
                                Skip blanks, tabs and newlines
                                while ((c = getch()) == ' ' || c == '\t' || c == '\n');
                                Return if not a number
                                if (c != '.' && (c < '0' || c > '9'))
                           ☞        return (c);                                       18
                                s[0] = c;
                                Get rest of number
                                for (i = 1;  (c = getchar()) >= '0' && c <= '9';  i++)   16
                                    if (i < lim)                                      17
                                        s[i] = c;
Collect fraction                if (c == '.')
                                    if (i < lim)
                                        s[i] = c;
                                    for (i++;  (c = getchar()) >= '0' && c <= '9';  i++)   7
                                        if (i < lim)
                                            s[i] = c;
Number is ok                    if (i < lim)
                                    ungetch(c);
                                    s[i] = '\000';
                           ☞        return (NUMBER);
It's too big; skip rest of line else
                                    while (c != '\n' && c != EOF)                     15
                                        c = getchar();
                                    s[lim - 1] = '\000';
                           ☞        return (TOOBIG);

                              #define     BUFSIZE        100
Buffer for ungetch              char                     buf[BUFSIZE];
Next free position in buf       int                      bufp = 0;

Get a (possibly pushed back)  getch()
character
                                return ((bufp > 0)  ?  buf[--bufp]  :  getchar());
```

Figure 2.2. Excerpt from a program book produced with the SEE Program Visualizer

Probably the most researched software visualization direction of the 1980s was program behavior, used mainly for educational purposes. The most popular early example of program behavior animation was Baecker's film from 1981 "Sorting Out Sorting", which provided a visual aid for understanding some of the most important sorting algorithms, such as shell-sort, bubble-sort, or quick-sort [Bae98]. The most prominent visualization systems of this period were Balsa [BS84], a system for animating algorithms used as an educational aid, and its direct successor, Balsa-II [Bro88], both illustrated in Figure 2.3.



```
procedure expression;
 procedure term;
  procedure factor;
   procedure expression;
    procedure expression;
     procedure term;
      procedure factor;
        begin
        if p[j]='(' then
          begin
          j:=j+1;
          expression;
          if p[j]=')' then j:=j+1 el
          end
        else if letter p[j] then
        if p[j]='*' then j:=j+1;
        end;
```

(A*B+AC)D
A
F
C
) D
E
T
F
E
T
F
E
T
F
File Edit Run Windows Algs Views Inputs
Insertion Sort: running
A
E
F
G
H
I
J
K
B
L
N
C
D
M
E I A K G H J L N F B C D M
K L N

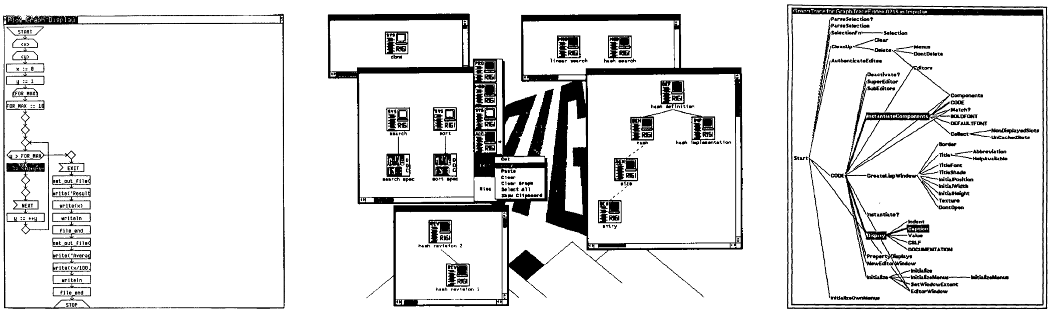
Figure 2.3. The Balsa (left) and Balsa-II (right) algorithm visualization systems

In 1983, Myers published his work at Xerox Palo Alto Research Center on Incense, a system for the visualization of data structures, such as records and pointers [Mye83].

The availability of personal computers equipped with displays enabled researchers to create interactive software visualization systems for a wide audience. Reiss envisioned the use of multiple views including simple visualizations (See Figure 2.4(a)) in an early IDE (i.e., Integrated Development Environment) from 1984, called Pecan [Rei84].

In 1986, Müller et al. presented Rigi, a software visualization system aimed at visualizing the structure of software systems in terms of components and relationships [Mül86, MK88], which stood the test of time and ended up becoming one of the most popular early software visualization systems (See Figure 2.4(b)).

The growing number of object-oriented programming languages emerging in the mainstream raised the interest in understanding not only the structure, but also the behavior of systems written according to this paradigm. Kleyn et al. proposed GraphTrace [KG88], a tool using concurrently animated views to visualize dynamic information, illustrated in Figure 2.4(c).



(a) Pecan (b) Rigi (c) GraphTrace

Figure 2.4. Examples of prominent systems from the 1980s

The various types of emerging visualizations raised the need to categorize and organize them. A first attempt in this direction was Myers's taxonomy of program visualization systems [Mye90]. Myers classified program visualization systems along two axes. The first axis determines what part of a program is visualized (i.e., code, data, or algorithms), while the second shows whether the displayed information is static or dynamic.

A few years after, Price et al. proposed a more detailed taxonomy [PBS93], according to which software visualization is categorized along six dimensions: scope, content, form, method, interaction, and effectiveness. The relative orthogonality of these taxonomies reveals the difficulty of categorizing software visualization.

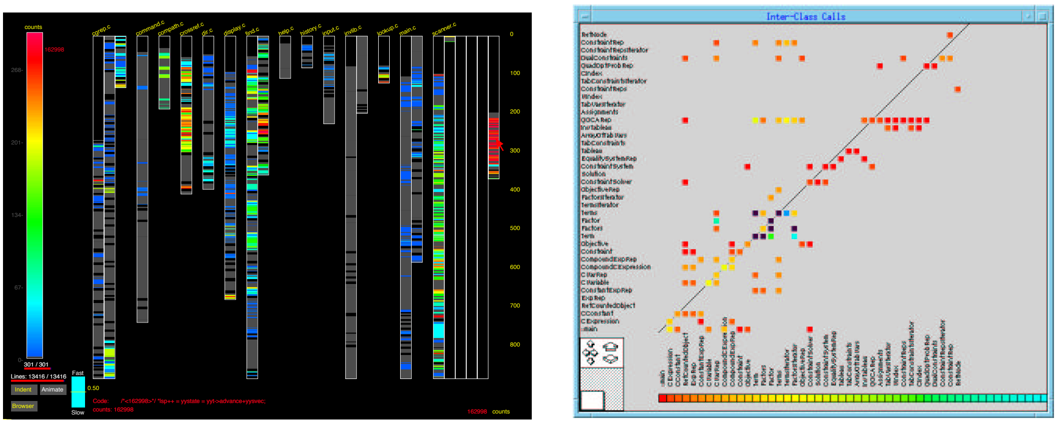
2.4 The 1990s

With time, more and more researchers gained interest in software visualization and started to investigate this novel research field. As a consequence, several seminal works in software visualization date are born in this period.

In 1992 Eick et al. presented SeeSoft [ESEE92], a software visualization tool able to provide fine-grained (i.e., down to the level of line of code) visualizations of software systems.

Later, Ball and Eick [BE96] successfully applied SeeSoft in several contexts, each with a different perspective on software, e.g., static properties, performance profiles (See Figure 2.5(a)), or version histories (i.e., evolution).

In 1993, in the context of dynamic program visualization, De Pauw et al. introduced Jinsight (See Figure 2.5(b)), a tool with novel views of the behavior of object-oriented systems [PHKV93].

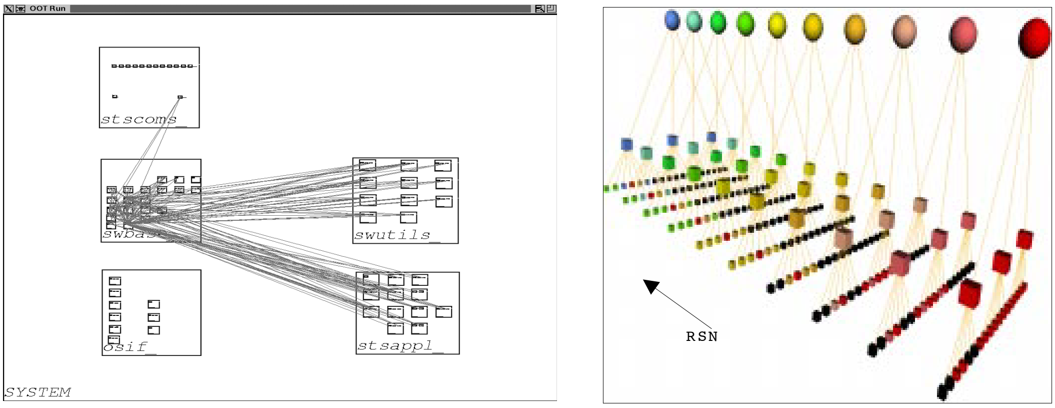


(a) Program execution in SeeSoft

(b) Inter-class call matrix in Jinsight

Figure 2.5. Examples of dynamic visualization

The first contributors to software evolution visualization were Holt and Pak in 1996, who used their tool called GASE (See Figure 2.6(a)) to visualize architectural data for eleven versions of an industrial software system [HP96]. Three years later, Gall et al. used both conventional 2D and novel 3D (See Figure 2.6(b)) visualizations to analyze the evolution of another industrial system, as captured by its release history [GJR99].



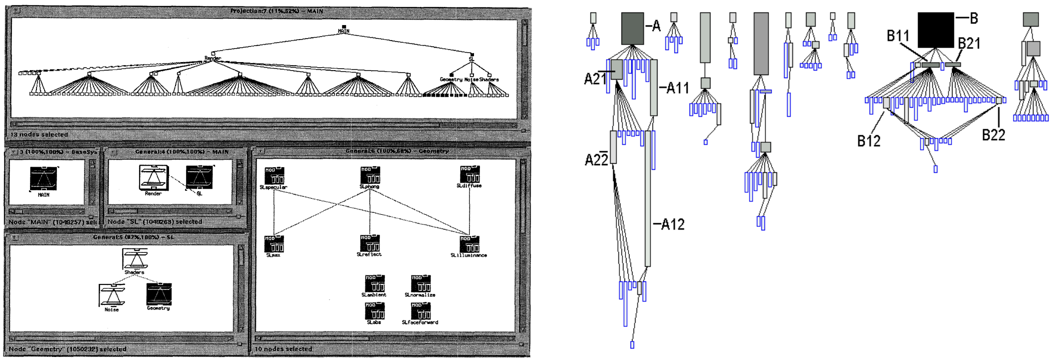
(a) Changing dependencies in GASE

(b) 3D Visualization of release histories

Figure 2.6. Early visualizations of software evolution

In 1995, Storey et al. presented SHriMP [SM95], a tool for the visualization of software structures (See Figure 2.7(a)), obtained by augmenting the Rigi environment with multi-perspective views and fisheye [Fur86] techniques.

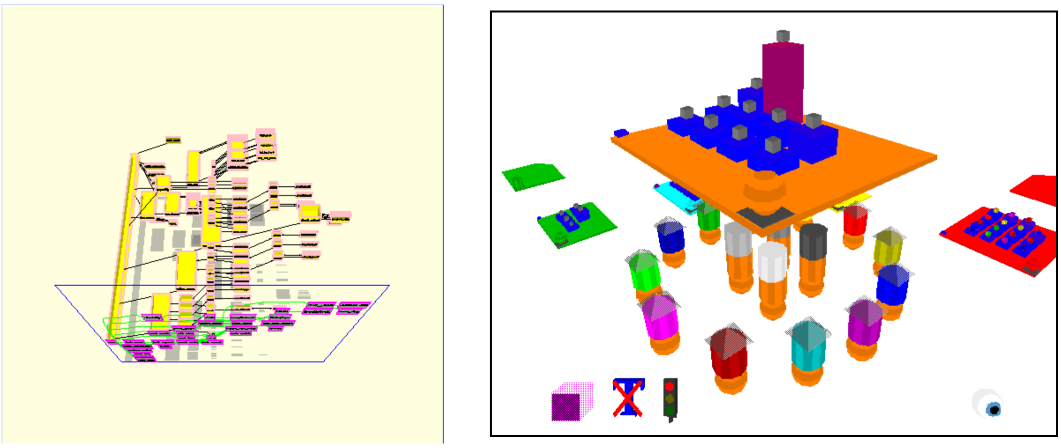
In 1999, Lanza proposed a lightweight visualization of software systems called polymetric views (See Figure 2.7(b)), implemented in the CodeCrawler tool [Lan99, DDL99]. The polymetric views, which employed simple, yet effective visualizations enriched with software metrics, were able to reveal outliers and patterns in object-oriented software systems.



(a) Fisheye views of nested graphs in SHriMP (b) Polymetric views in CodeCrawler

Figure 2.7. Examples of static visualization of system structure

This period was favorable also for experimenting with novel research directions for visualization, such as 3D (i.e., three-dimensional) visualization and Virtual Reality (VR). In 1995, Reiss presented a configurable engine for building 3D visualization of programs (See Figure 2.8(a)), called PLUM [Rei95]. In 1998, Young and Munro explored representations of software for program comprehension in VR [YM98], such as the one illustrated in Figure 2.8(b).

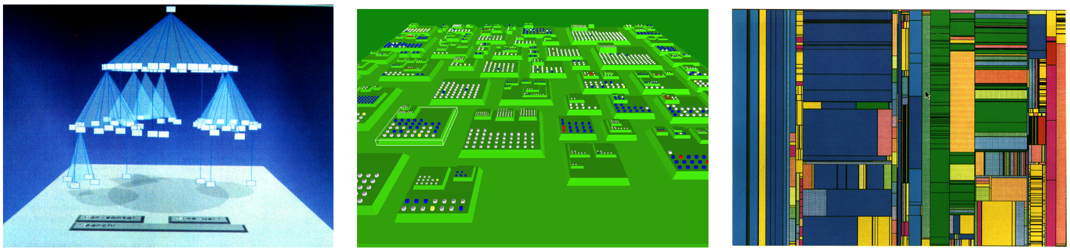


(a) PLUM (b) Software visualized in a VR environment

Figure 2.8. Early examples of 3D software visualization

The abundance and diversity of contributions are evidence of the momentum gained by the software visualization field during the 1990s.

The broader field of information visualization has been a source of inspiration for software visualization research. First, the excellent work of Edward Tufte on the use of visual means to efficiently present information [Tuf90, Tuf97, Tuf01] has influenced the entire field of information visualization, including software visualization. Furthermore, several information visualization techniques—such as the Cone Trees (See Figure 2.9(a)) of Robertson et al. [RMC91], the Information Pyramids (See Figure 2.9(b)) of Andrews et al. [AWP97], or the tree-maps (See Figure 2.9(c)) devised by Shneiderman [Shn92]—have inspired software visualization techniques.



(a) Cone Trees (b) Information Pyramids (c) Tree-map

Figure 2.9. Information visualization techniques that inspired software visualization

The Birth of UML. Although not a visualization approach, the Unified Modeling Language (UML) is closely related to software visualization. UML is a diagram-based language used to describe both static aspects (e.g., class diagrams, package diagrams, or component diagrams) and dynamic aspects (e.g., sequence diagrams, timing diagrams) of software. UML, as part of the Unified Software Development Process [JBR99], is the current de-facto industry standard for modeling.

The most popular use of UML was to visually support the description of the famous design patterns, i.e., general reusable solutions to commonly occurring problems in software design [GHJV95]. Figure 2.10 shows the class diagram corresponding to the *Observer* design pattern.

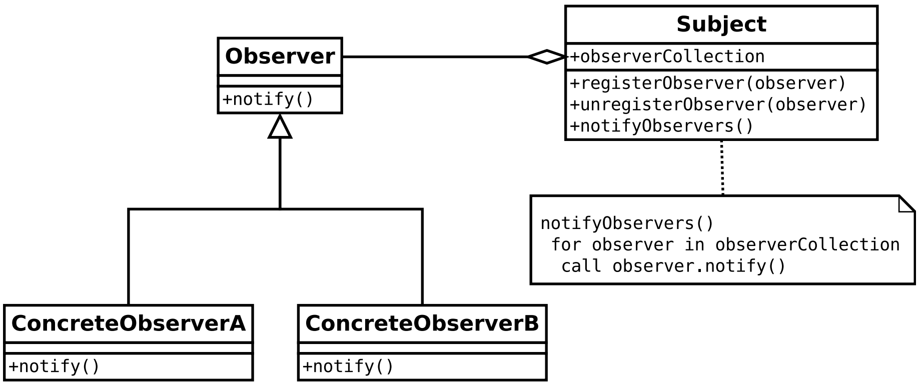


Figure 2.10. A UML class diagram

2.5 The 21st Century

By the beginning of the 21st century, information visualization was a research field of its own, with many dedicated venues, as opposed to software visualization, which was perceived as an esoteric means to addressing program comprehension or reverse engineering problems.

In 2001, an International Seminar on Software Visualization was organized in Dagstuhl to bring together the most significant practitioners and researchers working in the area of software visualization. Fifty researchers from all around the world discussed the state of the art in software visualization and identified the challenges for the future of the field [Die02]. More importantly, they decided to initiate a series of international venues on software visualization.

In the following years, two new venues dedicated to software visualization have been inaugurated. First, the IEEE International Workshop on Visualizing Software for Understanding and Analysis (i.e., VISSOFT) started in 2002, followed by the ACM Symposium on Software Visualization (i.e., SoftVis) in 2003. The momentum gained by the software visualization field in this period is reflected in the subsequent explosion of software visualization approaches.

Software Evolution Visualization. Before the beginning of the 21^{st} century, researching the evolution of software systems was a tempting, yet unfeasible perspective, due to the lack of data. However, with the spread of the open-source movement, a growing number of version control repositories containing entire development histories of software systems, became publicly accessible. As a consequence, many researchers turned the focus of their work towards software evolution. The following contributions illustrate this new direction in software visualization.

In 2001, using the Evolution Matrix polymetric view to analyze the evolution of classes, Lanza identified class evolution patterns [Lan01].

The Revision Towers [TM02] proposed by Taylor and Munro in 2002 showed change information at the file level, as extracted from the log file of the versioning repository.

The Evolution Spectrographs of Wu et al. presented the evolution of the system's components from the perspective of a single property [WHH04].

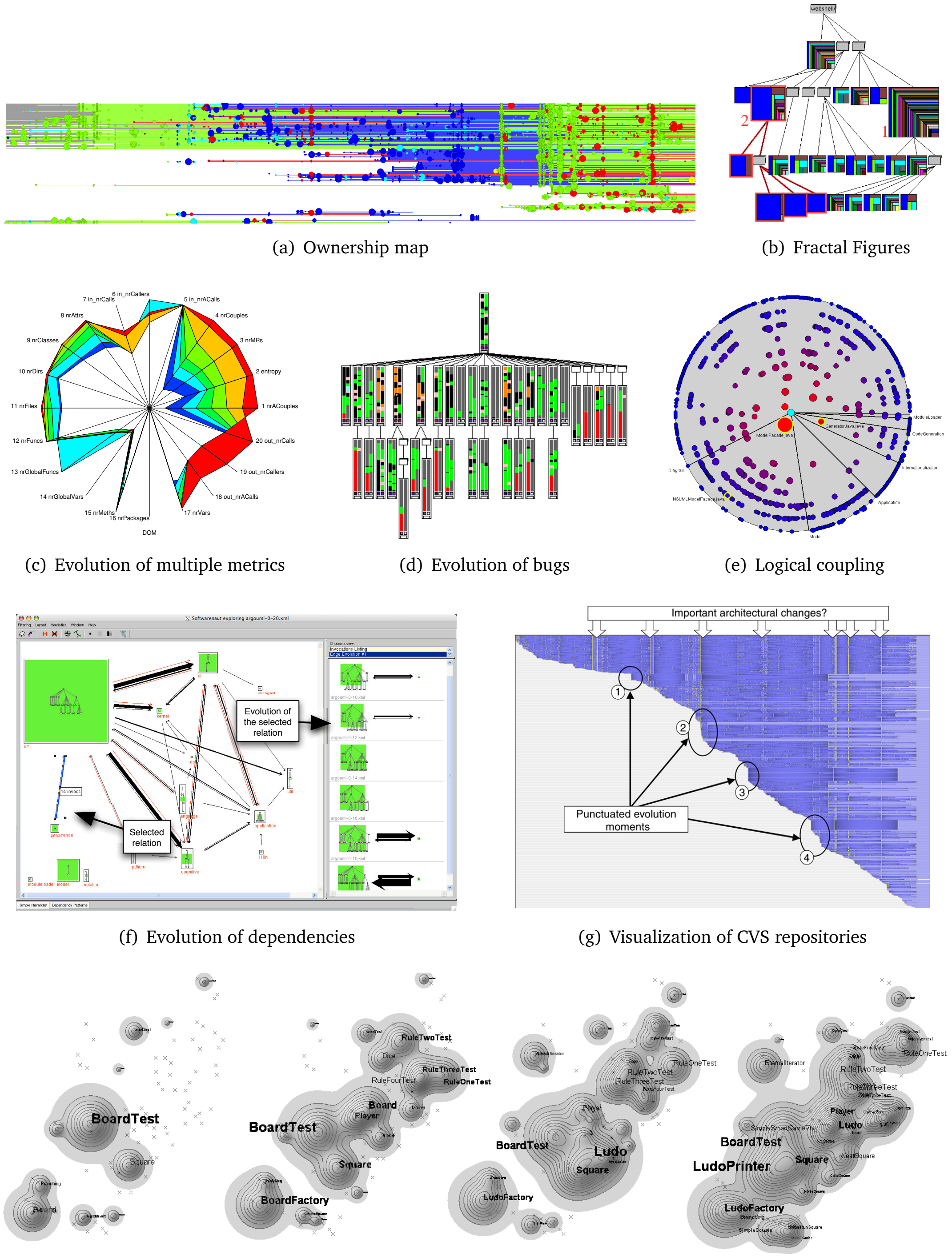
Fischer and Gall et al. visualized the evolution of features in large-scale software [FG04]. The visualizations of Pinzger et al. (See Figure 2.11(c)) implemented in RelVis encapsulated multiple evolution metrics [PGFL05]. Ratzinger et al. described EvoLens, a technique aimed at easing navigation in the immense exploration space created by the inclusion of time [RFG05].

Gîrba et al. studied the evolution of class hierarchies using the Class Hierarchy History Complexity polymetric view [GLD05]. Two interesting approaches focusing on developers were the study of the evolution of code ownership by Gîrba et al. [GKSD05], using a visualization called Ownership Maps (See Figure 2.11(a)) and the Fractal Figures (See Figure 2.11(b)) of D'Ambros et al., which focused on the development effort [DLG05].

D'Ambros and Lanza visualized the evolution of software bugs (See Figure 2.11(d)) with BugsCrawler [DL06b] and the evolution of logical coupling (See Figure 2.11(e)) between classes (i.e., classes that change together) with Evolution Radar [DL06a]. Lungu and Lanza visualized the evolution of inter-module relationships (See Figure 2.11(f)) using Softwarenaut [LL07].

Telea and Auber proposed a technique called Code Flows [TA08], which allowed tracking fine-grained changes of source code, i.e., at the level of the line of code. Voinea et al. provided overviews of system evolution extracted from CVS repositories (See Figure 2.11(g)), using the dedicated tools CVSscan and CVSgrab [VLT07].

Inspired by cartography, Kuhn et al. visualized the evolution of an application's vocabulary (See Figure 2.11(h)) using a tool called Software Cartographer [KLN08].



(a) Ownership map

(b) Fractal Figures

(c) Evolution of multiple metrics

(d) Evolution of bugs

(e) Logical coupling

(f) Evolution of dependencies

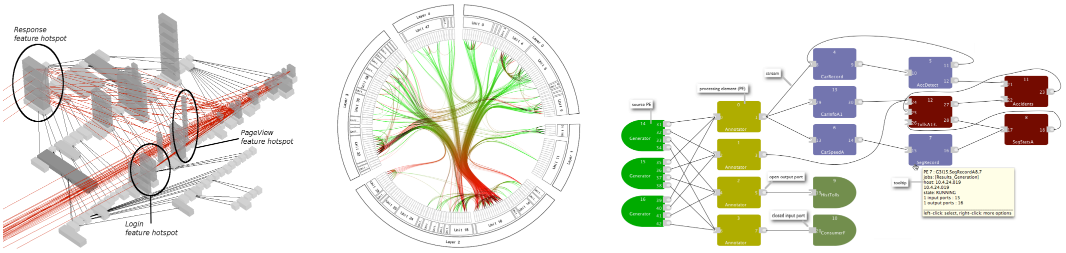
(g) Visualization of CVS repositories

(h) Evolving software map

Figure 2.11. Visualizations of software evolution

Noticing the richness and abundance of visualizations directed at evolution, Diehl divided software visualization approaches in three categories [Die07], by adding visualization of the evolution of software systems to the static and dynamic dynamic program visualization directions previously established by Price et al. [PBS93].

Dynamic Software Visualization. The biggest challenge in visualizing dynamic data is scalability, because of the large amounts of data that need to be manipulated in this context. However, due to the increasing computational power available today, dynamic visualization is starting to become feasible. Greevy et al. proposed a 3D dynamic visualization tool called TraceCrawler (See Figure 2.12(a) for the analysis of feature interactions [GLW06]. Another trace analysis tool which provided extremely appealing visualizations was ExTraVis (See Figure 2.12(b)) of Holten et al. [CHZ$^+$07]. De Pauw et al. described a tool called Streamsight (See Figure 2.12(c)), aimed at visualizing large-scale streaming applications [PAA08].

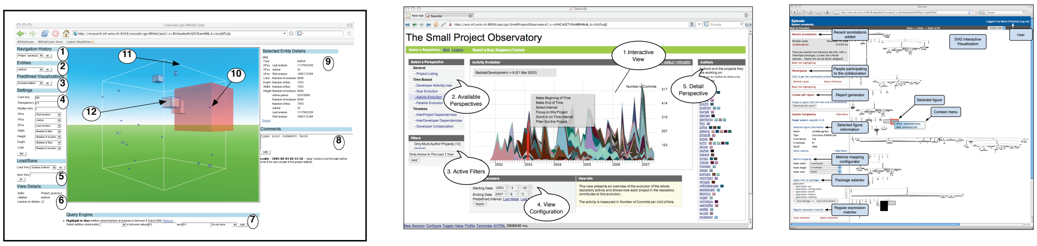


(a) TraceCrawler (b) ExtraVis (c) Streamsight

Figure 2.12. Modern dynamic visualizations

Software Visualization on the Web. Over the last decade, the Internet has become a basic commodity in many countries. To increase accessibility to their technology and reduce the inherent problems related to the installation and configuration of conventional tools, some researchers moved their software visualizations to the web, in the form of web applications. Mesnage and Lanza developed White Coats [ML05], a 3D web application for the visualization of software evolution, based on the data stored in versioning system repositories (See Figure 2.13(a)). Anslow et al. explored the web as a medium for 3D software visualization and animation [AMNB07]. Lungu et al. described the Small Project Observatory (See Figure 2.13(b)), an application whose holistic and focused views provides support for reverse engineering of software ecosystems [LLGH07, LLGR10]. D'Ambros and Lanza implemented Churrasco, a tool supporting the collaborative visual analysis of evolving software systems (See Figure 2.13(c)), through a methodology that combines software visualization and concurrent annotations [DL10].

Configurable Software Visualization. Due to the increasing popularity of software visualization and to the availability of rich data, a number of researchers invested in building highly-configurable visualization engines, to allow fast prototyping of new visualizations or the adaptation of existing visualizations to new types of data. Two examples of such engines were GSEE by Favre [Fav01] and Mondrian [MGL06], a visualization framework by Meyer et al. which allowed the interactive building of software visualizations.


1
2
3
4
5
6
7
8
9
10
11
12
The Small Project Observatory
1. Interactive View
2. Available Perspectives
3. Active Filters
4. View Configuration
5. Detail Perspective
Recent annotations added
People participating to the collaboration
Report generator
Selected figure information
Metrics mapping configurator
Package selector
Regular expression matcher
SVG Interactive Visualization
User
Selected figure
Context menu

(a) White Coats (b) The Small Project Repository (c) Churrasco

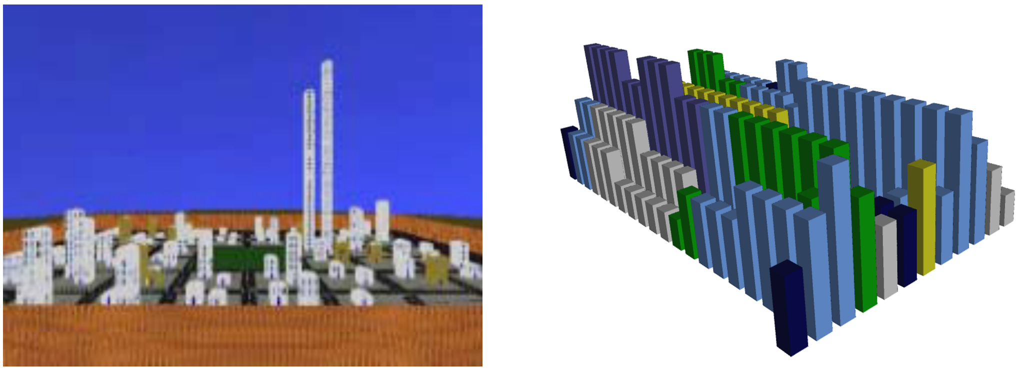
Figure 2.13. Software visualizations as web applications

Software Visualization Integration. Many researchers moved their visualizations closer to practitioners, by integrating them in popular IDEs. Examples of such integrations in Eclipse include Creole [LMSW03], an Eclipse plugin by Lintern et al. for SHriMP views [SM95] and X-Ray [Mal07], a plugin by Malnati for integrating polymetric views [Lan03] in Eclipse.

3D Metaphor-Based Visualization. The advances in hardware over the last decade led to a proliferation of three-dimensional solutions in software visualization. The instances of a first "generation" of 3D software visualization approaches, documented in an early survey [SB99] were mostly 3D extensions of popular 2D graph-based representations, which did not benefit of the real potential of 3D. However, the approaches representing the current state of the art [TC09] stepped away from metaphors inspired by 2D visualization and proposed novel metaphors, more suited for an immersive experience, such as the ones presented next.

In 2000, Knight et al. described and implemented a 3D visual metaphor for virtual reality called Software World [KM00], according to which a system was represented as the world, source files as cities, classes as districts, and methods as buildings (See Figure 2.14(a)).

In 2001, Maletic et al. [MLMD01] presented Imsovision, a system for visualizing object-oriented software in a Virtual Reality environment. Later, based on this experience, Marcus et al. proposed a new 3D representations for software visualization revolving around the concept of poly cylinders [MFM03] and implemented in sv3D (See Figure 2.14(b)).

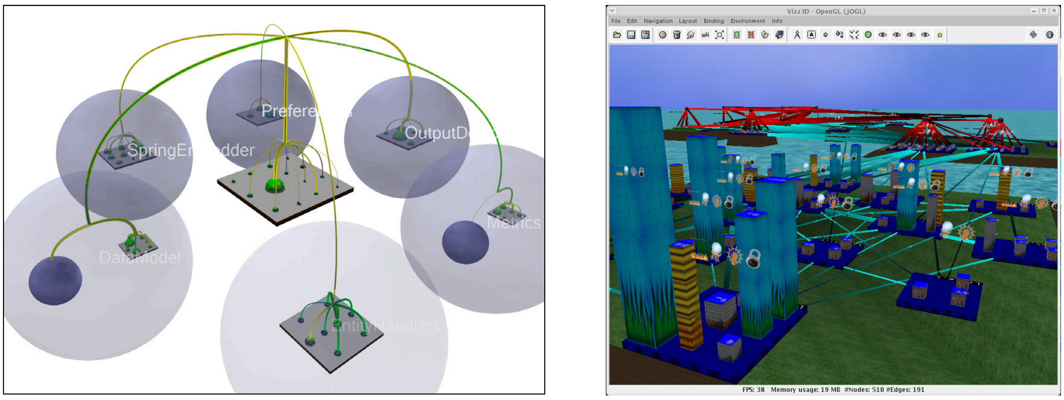


(a) Software World: a city representing a source file (b) sv3D: a poly cylinder representing a source file

Figure 2.14. Software visualizations based on 3D metaphors (1/3)

In 2004 Balzer proposed the Software Landscapes visualization [BNDL04], based on the landscape metaphor (See Figure 2.15(a)) whose familiarity should facilitate intuitive navigation and comprehension.

In 2003, Panas et al. envisioned and described a draft for a 3D city metaphor that would illustrate both static and dynamic aspects of software [PBG03]. Two years later, the authors followed up on the idea with an architecture for a framework that would allow the implementation of various 3D metaphors and a prototype called Vizz3D [PLL05]. After two more years, aware of the importance of system overviews, Panas et al. proposed a unified single-view visualization (See Figure 2.15(b)) based on their city metaphor [PEQ+07].


Prefere
SpringE
dder
OutputD
Metrics
DataModel
EntityHandlers
Vizz3D - OpenGL (JOGL)
File Edit Navigation Layout Binding Environment Info
FPS: 38 Memory usage: 19 MB #Nodes: 510 #Edges: 191

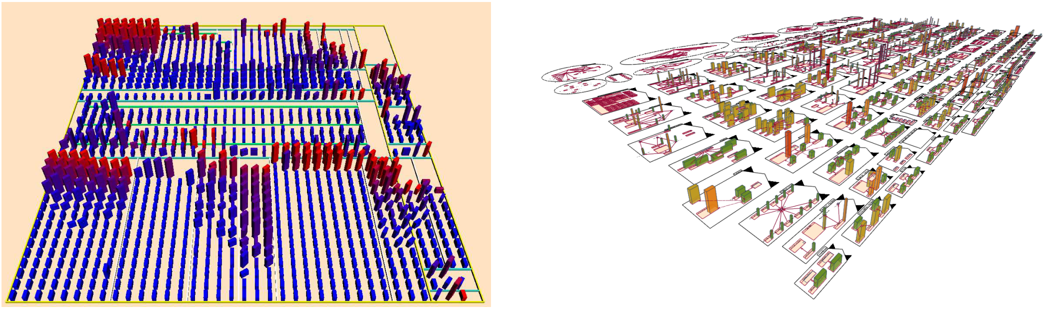
(a) Software Landscapes: a landscape metaphor

(b) Vizz3D: a city metaphor

Figure 2.15. Software visualizations based on 3D metaphors (2/3)

A third approach using a landscape-like metaphor (See Figure 2.16(a)) was the work of Langelier et al., whose implementation called Verso represented classes as 3D boxes with metrics mapped on their visual properties and employed two space efficient layouts to represent the structure of software systems [LSP05].

Finally, Lange and Chaudron enriched UML diagrams with software metrics [LC07] and obtained a set of views, including a 3D view based on a city metaphor, called UML-city (See Figure 2.16(b)).



(a) Verso: 3D boxes

(b) UML City: UML diagram enriched with metrics

Figure 2.16. Software visualizations based on 3D metaphors (3/3)

2.5.1 The Quest for Evidence

The last decade has revealed an increasing interest in the search for evidence of the efficiency and effectiveness of software engineering approaches. Wohlin et al. [WRH$^+$00] addressed the issue of experimentation in software engineering. Kitchenham et al. [KPP$^+$02] drafted a set of preliminary guidelines for empirical research in software engineering. Two years later, Kitchenham et al. proposed the adoption of evidence-based methodology in software engineering, inspired by its successful application to medicine [KDJ04].

The growing demand for empirical validation in software engineering has inevitably reached the software visualization field. As a side effect, the lack of empirical validation has become the "ultimate" cause invoked—often blindly—by reviewers when rejecting visualization approaches from major software engineering venues. Conversely, the mere inclusion of a controlled experiment has the magic effect of rendering "respectable" a software visualization paper, in particular in the presence of "statistical significance".

What makes software visualization so difficult to validate empirically, compared to other software engineering fields? Many software engineering fields have a clearly defined problem and working frame (i.e., can be easily abstracted as black boxes), their operation does not require human intervention (i.e., they are automated) and their outcome is deterministic. In contrast, software visualization approaches typically lack a clearly defined problem and working frame (i.e., due to the exploratory nature of the activities they support), do require human intervention in operation (i.e., they are semi-automated), and their outcome depends on the observational and visual analytical human skills. Software visualization alone rarely solves problems; instead, it supports human problem-solving processes. Measuring its contribution, i.e., the efficiency and effectiveness of software visualization, is quite a challenge.

A symptom of this difficulty is the complete lack of benchmarks for software visualization, as remarked by Maletic et al. [MM03]. While several fields of software engineering have dedicated benchmarks (e.g., distributed systems) that allow researchers to compare the performance of a new approach with that of the state of the art, there are no benchmarks for software visualization. The lack of benchmarks leads to an increased amount of effort required to set up such an experiment and to an increased number of threats to validity, due to the lack of standardization.

In spite of the adverse conditions, a number of empirical validations of software visualization approaches have seen light over the last three years. A successful empirical validation provides an advantage to any software visualization approach. Therefore, most of the authors of these experiments aimed at validating their own visual approaches, such as the interactive views for UML models by Lange et al. [LC07], the dynamic object process graphs by Quante [Qua08], or the trace visualizations by Cornelissen [CZRvD09].

Other researchers looked for general insights on visualization by evaluating tools developed by other researchers. For instance, Sensalire et al. presented a series of evaluation experiments involving the use of several existing tools with the goal of building a model of desirable features of a visualization tool [SOT09].

Undoubtedly, empirical validation has reached software visualization. It will be interesting to see its long term effect on the future of this field.

2.6 Summary

In this chapter, we presented a historical perspective of software visualization. It all started five decades ago with diagrams, pretty printers, and movies that illustrated program behavior. In the 1980s, the focus moved towards creating advanced pretty printers, but also algorithm animation and a few early static visualization approaches. In the 1990s, the points of interest for software visualization research were distilled to three distinct directions, triggered by the aspect of the software system being visualized: static information, dynamic information, and system evolution. However, the turning point towards the thrive of software visualization was in 2002, when the most prominent researchers in the field decided to initiate a series of venues dedicated to software visualization.

These are exciting times for software visualization research. On the one hand, the advances in hardware open new horizons barely imaginable two decades ago. One the other hand, the invasion of software visualization approaches—some of them rather futile—raises skepticism and calls for methods for evaluating their usefulness.

The first claims of our thesis is that depicting software systems as cities is a versatile metaphor for software visualization. In Part II we present our approach, based on the city metaphor, and three application contexts we use to demonstrate the versatility of the metaphor.

Part II

Approach

Preview

After looking at the history of software visualization and the challenges that lie in front of us, we start our exploration of the city metaphor. This part describes our city metaphor and the three application contexts we employ to demonstrate the versatility of the metaphor.

In Chapter 3, we look at the ideas underlying our city metaphor and how they influenced the design of the metaphor. After describing the metaphor, we apply it in the context of program comprehension, on two case studies. The insights we learn are a good indication of the usefulness of the metaphor, while the questions remained unanswered drive our research towards the second application context.

In Chapter 4, we extend our approach to enable the visualization of software evolution, by creating three complementary techniques. Using three case studies, we illustrate how we apply our approach in the context of software evolution. We obtain the answers to a number of questions that we could not address in the context of program comprehension. Moreover, we ask the developers of the systems to assist us with fragments of system history they witnessed.

In Chapter 5, we present our third application context of our approach, i.e., design quality assessment. We describe a technique called disharmony map, which relies on actual design problem data acquired using an existing technique called detection strategies. We use this approach on several case studies to observe software systems in terms of their deficiencies.

In Chapter 6, we present the tool support we developed to enable us to apply our approach on real systems and discuss how we dealt with a number of challenges related to tool support, such as performance and scalability, configurability, usability, and availability.

Chapter 3

A City Metaphor for Program Comprehension

3.1 Introduction

There's no place like home. Developers spend an important part of their time constructing complex software systems, an activity termed as programming, which is according to Weinberg "a kind of writing" [Wei98]. The more familiar we are with a program, the easier it is to understand the impact of any modification we may want to perform, i.e., familiarity has an important influence on program comprehension strategies [SFM99].

Familiarity is strongly related to *habitability*, which is what makes a place livable, like home. Gabriel [Gab96] states that "habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in life to understand its construction and intentions [...]". While Gabriel's position stems from the point of view of language design, we focus on the creation of habitable visual representations of software systems, as support for program comprehension and reverse engineering activities.

We argue that habitability is an important, yet neglected concept in software visualization. The existing approaches fail at conveying a sense of habitability to the viewer. Inherently, 2D visualizations lack the realism required for making the viewer feel immersed in the environment, as opposed to 3D visualization. However, most of the existing 3D visualization do not exploit the *locality* of their exploration space, allowing objects to be freely moved. In such environments, viewers are vulnerable to disorientation, one of the main arguments against 3D visualizations.

We chose a *city* metaphor [WL07b] for our software visualization approach, due to the many similarities between software and civil engineering, which enable a straightforward and intuitive mapping between the two domains. Moreover, a city is an intrinsically complex construct and can only be incrementally explored, in the same way the understanding of a complex system increases step by step.

Before describing our vision of the city metaphor, we briefly describe the underlying metamodel of our approach, which enables us to visualize software systems written in different programming languages, such as Java, C++, C#, or Smalltalk.

3.2 Modeling Software Systems

To model the software systems we aim to visualize, we rely on FAMIX [DTD01], a language-independent meta-model, whose core is synthesized in Figure 3.1. Since we focus on object-oriented systems, the entities that we need to understand in this context are packages, classes, methods, and attributes, while the relationships among these entities are inheritance definitions, invocations, and accesses.

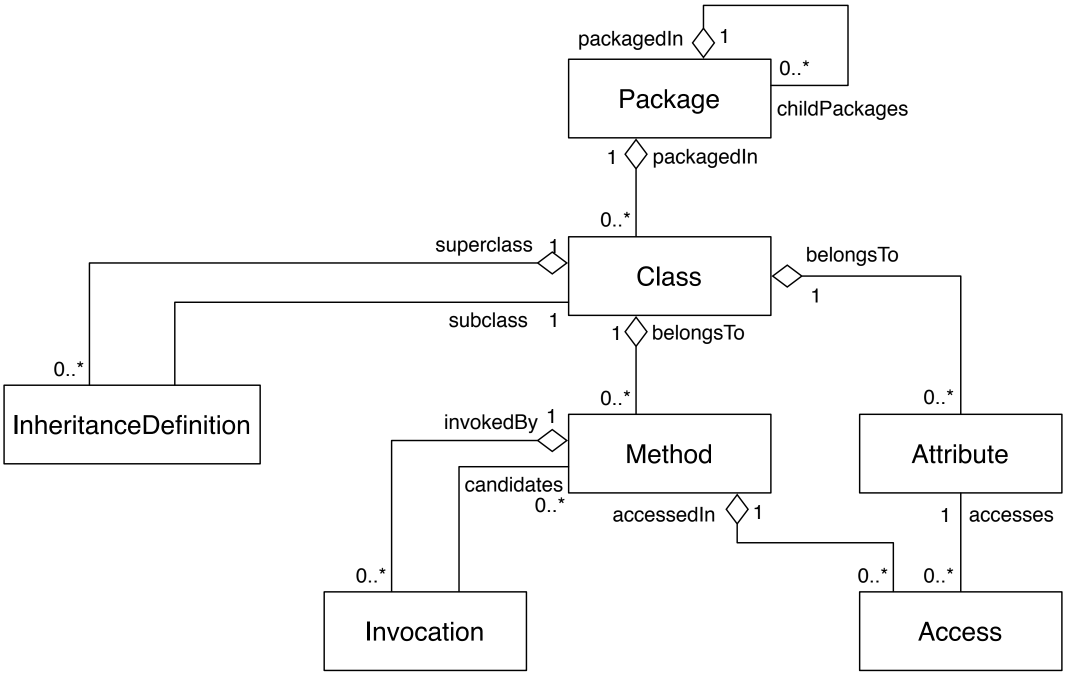


Figure 3.1. The core of the FAMIX meta-model

3.3 The City Metaphor

We propose a 3D visualization approach based on a city metaphor, i.e., we visually represent software systems as cities. Since a city, with its downtown and suburbs, is a familiar concept with a clear notion of orientation, our city representations enable the users to employ their natural wayfinding skills, reducing thus the risk of disorientation often associated with 3D visualization.

Moreover, our code cities revolve around the notion of locality, which enables users to incrementally build familiarity with the represented software systems. By choosing the metaphor, we defined the domain mapping: "Software systems as cities" sets software as the target domain and city as the source domain. The next step in defining our metaphor is concept mapping.

3.3.1 Concept Mapping

In our *code cities*, classes are visualized as buildings and packages as districts, as illustrated in Figure 3.2. This choice is rooted in the fact that classes are the cornerstone of the object-oriented paradigm and, together with the packages they reside in, the primary orientation point for developers.

The position of a city artifact depends on the "location" of the depicted software entity in the grand scheme of the software system it belongs to. Since many modern object-oriented programming languages provide the package mechanism as a solution for organizing classes, we chose to reflect the package hierarchy in the city's districts.

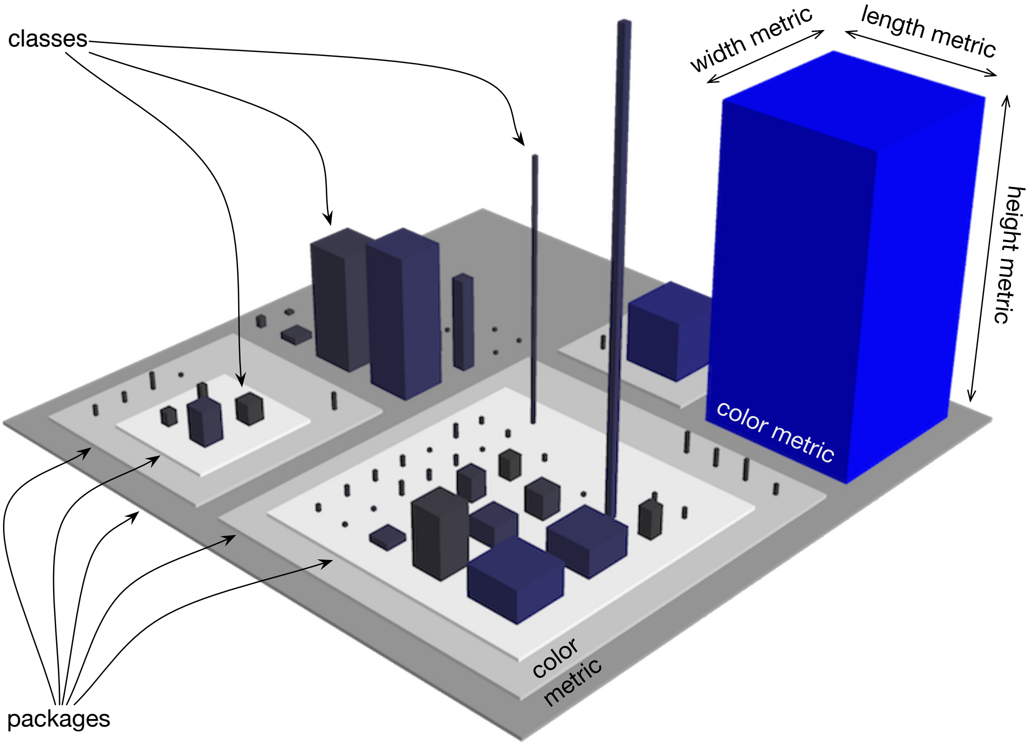


Figure 3.2. An example illustrating the principles of our city metaphor

The final step in defining our metaphor is the property mapping, according to which the visual properties of the city artifacts reflect properties of the software elements.

3.3.2 Property Mapping

Software metrics have been extensively used in software visualization to obtain highly condensed views. An influential approach in this context is the polymetric views of Lanza et al. [LD03], which are 2D visualizations enriched with software metric information. In a polymetric view, nodes represent software entities, edges represent relationships between the entities, and their visual properties (i.e., width, height, position, and color) reflect the values of a set of software metrics.

Our approach is an extension of the polymetric views, in that we also map a set of software metrics on the visual properties of artifacts, as shown in Figure 3.2. The five properties of the city artifacts able to depict software metrics are the three dimensions (i.e., width, length, and height), the color, and the transparency, while the position property expresses locality.

We have access to a broad set of software metrics provided by the Moose reengineering platform [NDG05, DGKR09], which allows the creation of a whole range of visualizations to support reverse engineering activities. However, finding the right combination of mappings for a particular task remains an open challenge, given the limited number of visual properties that can be visualized at any moment.

After experimenting with several property mappings we found one, called *magnitude*, which is well suited for the creation of software system overviews. According to this mapping, depicted in Figure 3.3, the *Number Of Methods* (NOM) metric (i.e., a measure of a class's functionality) is mapped on the height of the buildings, the *Number Of Attributes* (NOA) metric (i.e., a measure of a class's state) on both their width and length[1], and the *number of Lines Of Code* (LOC) metric (i.e., a typical measure of size) on the color of the buildings, from dark gray (low) to intense blue (high). For packages, the value of the *Nesting Level* (NL) metric is mapped on the color of the districts, according to a gray color scheme: the deeper a package is nested in the hierarchy, the lighter is the shade of gray of the district depicting the package.

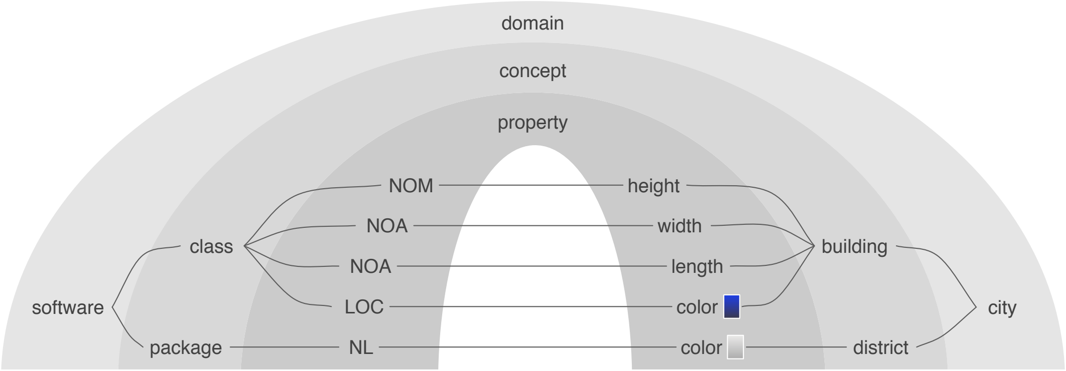


Figure 3.3. The *magnitude* property mapping presented in the grand scheme of our metaphor

We exemplify this mapping by applying it on version 0.23.4 of ArgoUML, a Java system with 88 packages, 1,817 classes, and 143,682 lines of code. The resulting city is shown in Figure 3.4.

Visualization is known to reveal reveal patterns and code cities make no exception. The code city of ArgoUML exemplifies several building archetypes:

- "skyscrapers" — classes with few attributes and many methods (i.e., thin and very tall),
- "parking lots" — classes with many attributes and few methods (i.e., wide and flat),
- "office buildings" — classes with many attributes and many methods (i.e., wide and tall),
- small houses — classes with few attributes and few methods.

[1] Mapping different metrics on the two horizontal axes is unpractical. While one can easily identify the vertical axis due to the effect of gravity, one cannot distinguish between the two horizontal axes without additional visual clues.

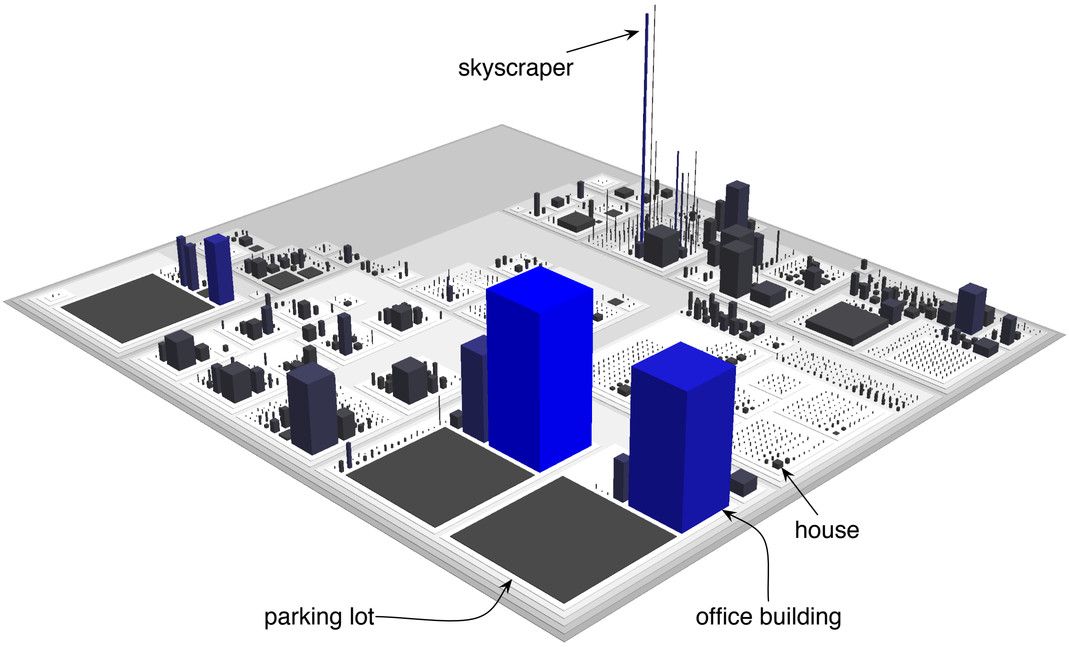


Figure 3.4. The code city of ArgoUML, with annotated building archetypes

The resulting overview represents a good starting point for further analysis, according to the visual information seeking mantra of Shneiderman: “Overview first, zoom and filter, then details-on-demand” [Shn96].

Figure 3.3 shows the pairs of properties that are connected via mapping, but not the mapping functions, i.e., the mathematical functions that take the value of one of the input property (i.e., the software metric) and compute the value of the output property (i.e., the visual property of a city artifact). To learn the effect of the mapping function on the habitability of the code cities, we experimented with several mapping functions.

Identity mapping. The identity mapping strategy is the most straightforward mapping strategy, based on the *identity* function: $f(x) = x$. With this type of mapping, a visual property reflects most accurately the value of the assigned software metric. We used this mapping strategy to map the metrics onto the three dimensions of the buildings shown in Figure 3.4.

One problem associated with this mapping is that the variety of shapes and sizes of the buildings in this code city goes against one of the *gestalt* principles [Few04], which shows that humans are able to distinguish efficiently only four to six different shape sizes. We addressed this issue by building mapping functions that reduce the cardinality of the output range to five, corresponding to a five-point scale (i.e., very low, low, average, high, and very high).

For each building dimension, i.e., width (w), length (l), and height (h), we conceived a five-point scale and each of the five categories corresponds to a building type, as illustrated in Table 3.1. The unit corresponds to the size of a “floor”, e.g., an apartment block is a six-floor building. Figure 3.1 exemplifies the visual representation of the buildings. However, given that we map two different metrics, it is possible to have combinations that make more or less sense, such as a one-floor house with the base size of an apartment block.

Category	*h*	*w* & *l*	Building Type
very low	1	1	House
low	3	2	Mansion
average	6	4	Apartment Block
high	12	8	Office Building
very high	40	12	Skyscraper

Table 3.1. Dimensions and building types

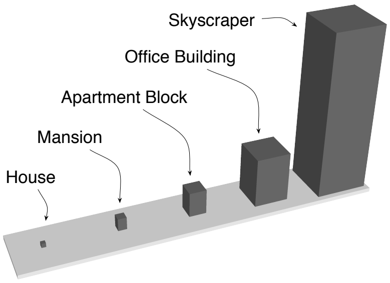
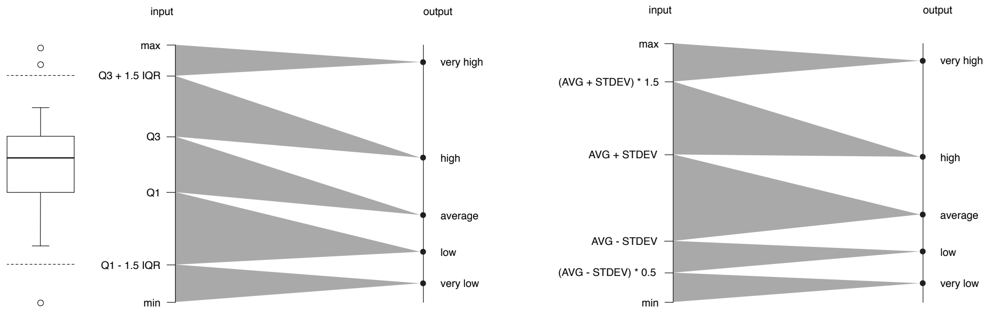


Figure 3.5. Building type representations

The final step in reducing the variety of building sizes was finding the means to split the input range into five contiguous sub-ranges and map them to the five output values. Next, we present two mappings, based on two different splitting strategies.

Box plot based mapping. The first technique we used for splitting the input range is the box plot [Tuk77], widely used in statistics to reveal the center of the data, its spread, its distribution, and the presence of outliers. The construction of a box plot implies the computation of the following three quartiles: the lower or first quartile (Q1), the median or the second quartile, and the upper or third quartile (Q3). The range between the first and the third quartiles, called interquartile range (IQR), hosts the middle 50% of the values. Besides the three quartiles, the box plot separates the normal values from the outliers.

We use the lower outlier limit, the lower quartile, the upper quartile, and the upper outlier limit to split the input range into five value ranges, corresponding to the five-point scale. Consequently, this function will map any input value to one of the five output values, according to the sub-range that the input value belongs to, as illustrated in Figure 3.6(a).



(a) Box plot based mapping

(b) Threshold based mapping

Figure 3.6. Mapping strategies aimed at reducing the complexity within code cities

An advantage of the box plot based mapping is that the boundaries can be computed automatically from the input data set. Moreover, by mapping the interquartile range to the central building type, we ensure that at least half of the buildings are apartment blocks, which leads to well-balanced code cities, in terms of their building types. We consider this an increase in habitability. However, we achieve this improvement at the cost of comparability and generalization, given that the boundaries obtained from the box plot only depend on the input data set (i.e., the metric values present in the system). A visual comparison of two code cities using such a mapping is not possible, because there is no correspondence between the input ranges of two different systems.

Threshold based mapping. Comparing code cities in this context requires some "magic numbers" for the boundaries that hold across system frontiers. Lanza and Marinescu measured a wide range of commercial and open-source systems in terms of sizes, domain, and type and presented a set of statistics for several software metrics [LM06]. We used the data from this work to split our input range into five sub-ranges, as illustrated in Figure 3.6(b). For example, the resulting set of boundaries for the number of methods metric in the case of Java systems is $\{2, 4, 10, 15\}$.

The advantage of this approach is that the code cities have common splitting boundaries and therefore can be visually compared. The main disadvantage is that it relies on the statistic data, which is currently limited to just a number of metrics and for only two programming languages, i.e., Java and C++.

Comparison between the mapping strategies. To illustrate the effect of the mapping functions on the appearance of code cities, we present in Figure 3.7 the code city of ArgoUML using each of the three described strategies (i.e., identity mapping, box plot based mapping, and threshold based mapping) to map the number of methods on the height of the buildings. To enable a clean observation of the variance in height, we maintained the same strategy (i.e., box plot based) for mapping the number of attributes on the base size.

The detail in Figure 3.7 shows the effect on the three mapping strategies on one of the system's packages. The top detail shows the broad variety of building heights in such a small district: The 28 classes in this small package exhibit a broad range of values for the number of methods metric. With a box plot based strategy, most of the buildings in the district are skyscrapers: Most of the classes in this package are outliers in terms of the number of methods metric. However, according to the statistics we used for the threshold based strategy, only some of them are considered outliers for Java systems.

The two mapping strategies aimed at reducing the complexity of code cities have advantages and disadvantages. On the one hand, the box plot based mapping improves the habitability in an artificial way. After all, if a software system is not habitable, it should not be represented as a habitable city. On the other hand, the threshold mapping is of limited use, since it relies on hard to get statistical data. After experimenting with the different strategies, we settled on the identity mapping, because of its accuracy of representation. Therefore, in the remainder of the dissertation we use the identity mapping.

3.3.3 Rectangle Packing Layout

To achieve a scalable visualization for large-scale software systems, we were interested in a layout algorithm which:

1. does not waste much of the cities' real-estate,
2. reflects the given containment relationships, and
3. takes into account the dimensions and proportions of the buildings.

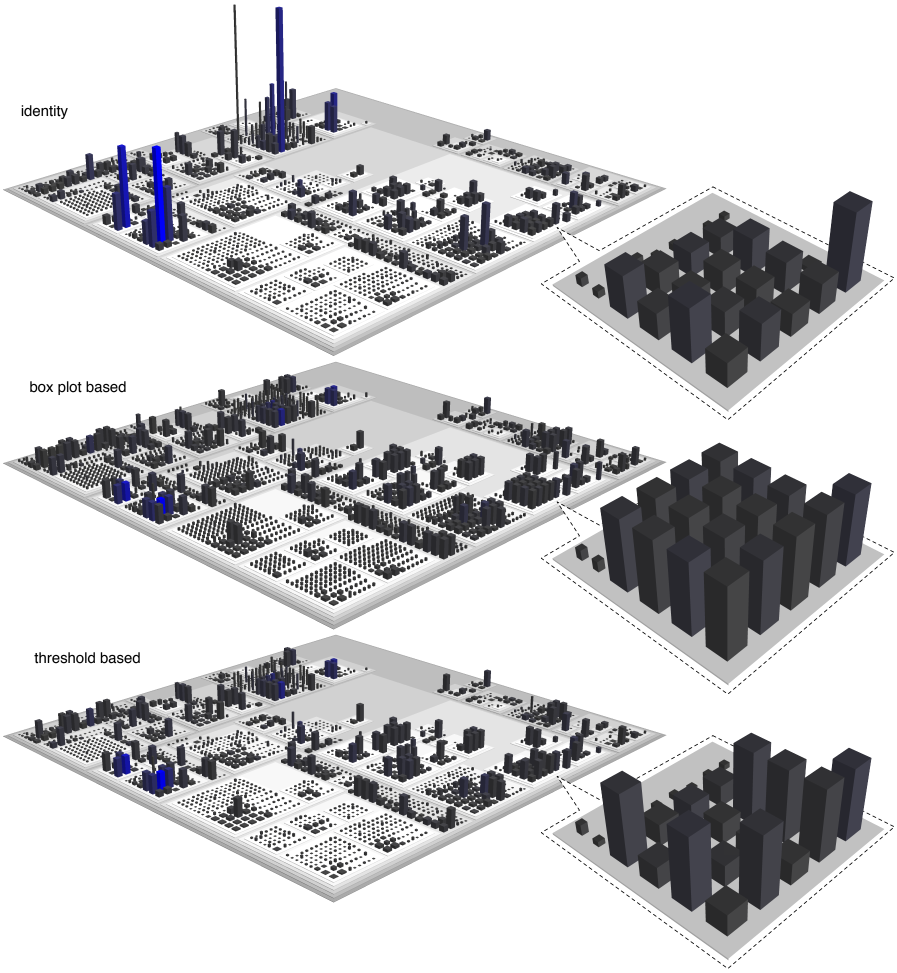


Figure 3.7. Identity, box plot based, and threshold based mappings compared

An efficient layout for hierarchical data is the tree-map technique proposed by Shneiderman in 1992 [Shn92], which splits the available rectangular space in rectangular areas proportional to an attribute of the node, without any waste of space. However, the tree-map technique does not satisfy our last requirement.

Our layout must deal with elements of given dimensions, which makes it similar to rectangle packing, a well-known problem in the chip design field. Moreover, contrary to the tree-map algorithm, our layout cannot establish *a priori* how much space it will need to place the element.

We started designing our algorithm from an algorithm for packing lightmaps[2], i.e., data structures used to encapsulate the brightness of surfaces in video games. Our layout, whose algorithm is presented next, is able to handle very large code cities. Figure 3.8 presents a top view of the same code city of ArgoUML, containing 1,817 buildings (i.e., very dark gray to blue squares) laid out on top of 88 districts (i.e., gray rectangles).

[2] `http://www.blackpawn.com/texts/lightmaps`

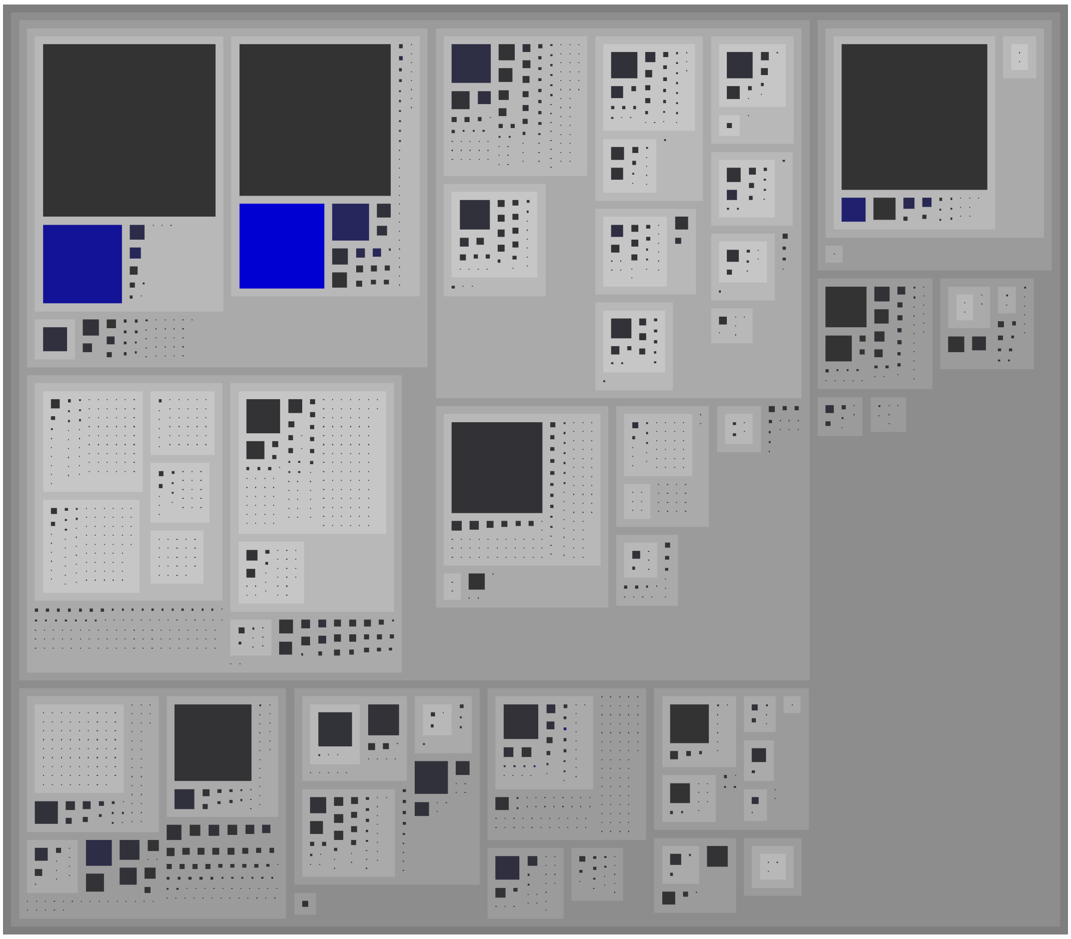


Figure 3.8. A top-down view of the layout in the code city of ArgoUML

The Layout Algorithm

The core of the algorithm deals with laying out a set of rectangles. The structure of a code city is a tree, i.e., the city contains main districts (i.e., representing root packages), which contain other districts or buildings (i.e., classes), etc..

Laying out the entire city implies performing the algorithm recursively at each hierarchical level starting with the leaves and resizing the containers to fit their content as the algorithm goes up the hierarchy, in a post-order traversal.

To partition the space, the algorithm uses *ptree*, a 2-dimensional *k*d-tree [Ben75]. The root of *ptree* corresponds to the entire available space, while each of the other nodes of *ptree* corresponds to a particular partition of the space. A node is aware of its assigned space *rectangle* and whether it has been *occupied* or not. The algorithm also keeps tracks of the area currently covered by elements, i.e., *covrec*.

Algorithm 3.1 Rectangle Packing Layout for a collection of *elements*

```
Require: elements are sorted by size, descending
Ensure: elements are efficiently laid out efficiently and without overlapping
ptree.root.rectangle.size ← $\sum_i elements[i].size$
covrec ← (0, 0)
for el in elements do
    pnodes ← empty leaf nodes in ptree with $pnode.rectangle.size \geq el.size$
    preservers ← new dictionary
    expanders ← new dictionary
    for pnode in pnodes do
        if placing el in pnode would preserve the size of covrec then
            waste ← amount of remaining space if pnode was split to place el
            add pnode : waste to preservers
        else
            ratio ← aspect ratio of covrec if pnode was used to place el
            add pnode : ratio to expanders
        end if
    end for
    if preservers is not empty then
        targetnode ← key corresponding to the lowest value in preservers
    else
        targetnode ← key corresponding to value in expanders closest to 1
    end if
    if targetnode.rectangle perfectly fits el then
        fitnode ← targetnode
    else
        fitnode ← perfect fitting node for el after splitting targetnode
    end if
    fitnode.occupied ← TRUE
    move el to fitnode.rectangle.position
    if fitnode is a boundary expander then
        expand covrec to the newly covered area
    end if
end for
```

In addition to the basic form presented here, the algorithm we use in CodeCity also deals with spatial separators, i.e., margins and gaps between elements.

Example. To illustrate our algorithm, we walk the reader through a simple example, depicted in Figure 3.9. The figure has a grid structure, i.e., it is split both horizontally and vertically. The five columns correspond to the initial state and to the four steps required to place the four elements, one at a time. The rows illustrate various aspects of the algorithms. The top row shows the elements that have not yet been placed, with the next element in the highest position. The middle row shows the configuration of the layout space, which includes elements that have already been processed. Finally, the bottom row shows the evolution of the binary partition tree, whose structure reflects the layout space.

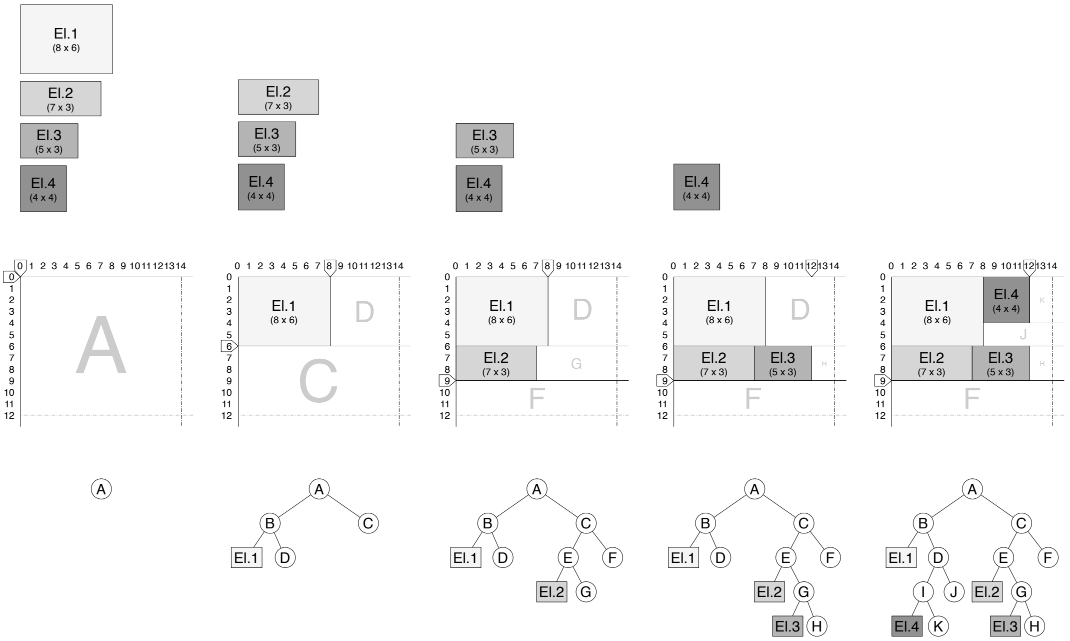


Figure 3.9. Example: four elements (top) laid out (middle) using a partition tree (bottom)

We start with the four elements (i.e., *El.1* to *El.4*) whose sizes are specified beneath their names, e.g., element *El.1* has a dimension of 8 × 6 units. To increase the efficiency of the space usage, we order the elements by one of the sizes.

Since we initially do not know how much space we need, we assign a space of 24 × 16 units (i.e., $width = 8+7+5+4$; $height = 6+3+3+4$) to the root node (i.e., node *A*) of the space partitioning tree, which makes up for the worst case scenario. However, instead of showing the entire layout space in the figure, we limit ourselves to show only the space that is relevant for this scenario. Initially, there are no placed elements yet, and therefore the covered area (i.e., $coverec$) is initialized to $(0,0)$, as illustrated by the two boundary markers in the figure (i.e., second row, first column).

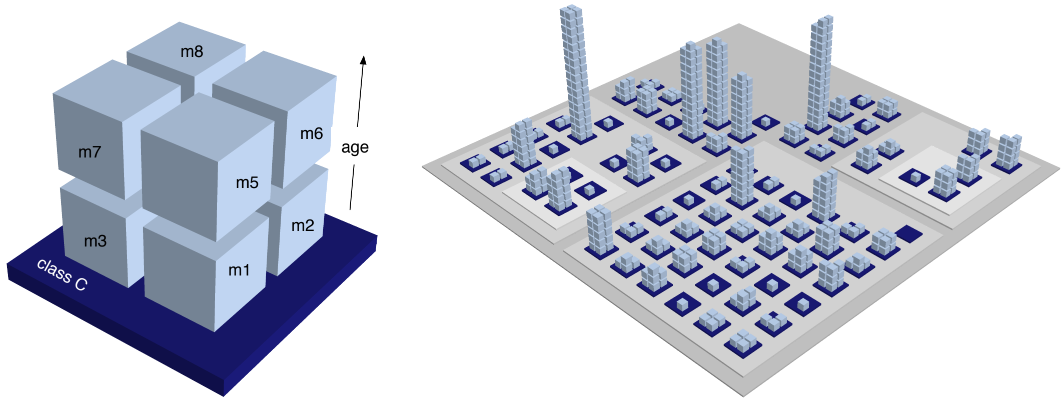
In the first step after initialization, the first element is placed in the only available node, i.e., *A*. However, this node's assigned space is larger than the size of the element. Therefore, it is split with two cuts. The first is a horizontal cut performed at the height of the element which splits *A* into nodes B and C. The second is a vertical cut performed at the width of $El.1$, which further splits B into a node of the exact needed size ($fitnode$) and node D. $El.1$ is placed in $fitnode$, $covrec$ is updated to $(8,6)$, and the algorithm passes to the next element.

For $El.2$ there are two potential candidates, namely C and D, both large enough to host the element and both boundary expanders. Between the two, we need to chose the one that expands the boundaries such that the resulting covered area has an aspect ratio closer to a square. Placing $El.2$ in C would lead to a covered area of 8 × 9, while placing it in D would lead to an area of 13 × 6. Since C is clearly our $targetnode$, the algorithm continues with splitting C in a similar manner to the one in which we split *A* earlier.

The algorithm continues in the same way until it reaches the last step, when placing $El.4$. Although there are three free nodes (i.e., D, F, and H), only two of them (i.e., D and F) are large enough to host the element. While F is a boundary expander, D turns out to be a boundary preserver, because it would allow placing the element in it and preserve the current boundaries. Consequently, this node is chosen and the algorithm continues. Finally, after cropping the space along the two axes pointed by the boundary markers, the only wasted space in this example remains between elements $El.3$ and $El.4$.

3.3.4 Fine-Grained Representation

The representation granularity presented so far is *coarse*, i.e., it only shows packages and classes.



(a) Principle

(b) District `cognitive` in the code city of ArgoUML

Figure 3.10. Fine-grained representation

To address the need for a more fine-grained representation, we depict methods as cuboids ("bricks") laid out on top of each other in layers of four, as seen in Figure 3.10). The method bricks are placed in the order of their creation (i.e., older down, newer up)—according to the information from the versioning repository—on top of a platform, which represents the class itself. The height of a building continues to be proportional to the class's number of methods.

However, when using the fine-grained representation for cities with extremely tall buildings, the overview of the system is compromised. Next, we illustrate this problem and our solution.

3.3.5 The Progressive Bricks Layout

Applying the fine-grained representation to a software system whose classes include at least one with an extremely high number of methods, the building representing this class will contribute to a dominantly vertical appearance of its code city. Given that the orientation of computer screens is typically landscape, the use of the screen real estate for the representation of such code cities is suboptimal. Looking at the whole city is only possible from a distance, which invalidates the point of having a finer-grained representation, for details are no longer visible.

To address this drawback, we developed a self-adapting vertical layout called *Progressive Bricks*, which maps the functional magnitude of a class on the volume of the building. Depending on its number of methods, the layout will place the "bricks" along the imaginary walls of the building representing the class, such that there is a harmonious ratio between the building's base size and its height. In other words, the width of the walls, in terms of number of bricks, is adapted to the total number of bricks of the building in order to obtain reasonable heights. The result of this is not only a visual categorization (i.e., the number of bricks per layer side) of the classes in terms of functionality, but also a categorization within each category, based on the number of layers of the building. The layout algorithm we devised is presented next.

Algorithm 3.2 Progressive Bricks Layout for a collection of *elements*

```
1.  $sc \leftarrow 0$
2.  repeat {find the side capacity $sc$ such that it satisfies: $bc_{min} \leq sc \leq bc_{max}$}
3.      $sc \leftarrow sc + 1$
4.      $lc(sc+1) \leftarrow sc * 4$
5.      $nol_{min}(sc+1) \leftarrow sc * 2$
6.      $bc_{min}(sc+1) \leftarrow lc(sc+1) * nol_{min}(sc+1)$
7.      $bc_{max}(sc) \leftarrow bc_{min}(sc+1) - 1$
8.  until $bc_{max} \geq elements.size$
9.  for $i = 0$ to $elements.size - 1$ do
10.     if $sc = 1$ then
11.         $lc \leftarrow 1$
12.         $biws \leftarrow 0$
13.         $si \leftarrow 0$
14.     else
15.         $lc \leftarrow (sc-1) * 4$
16.         $biwl \leftarrow i \mod lc$
17.         $biws \leftarrow biwl \mod (sc-1)$
18.         $si \leftarrow biwl \div (sc-1)$
19.     end if
20.     if $si = 0$ then {northern side}
21.         $pi_x \leftarrow biws$
22.         $pi_y \leftarrow 0$
23.     else if $si = 1$ then {eastern side}
24.         $pi_x \leftarrow sc - 1$
25.         $pi_y \leftarrow biws$
26.     else if $si = 2$ then {southern side}
27.         $pi_x \leftarrow sc - biws - 1$
28.         $pi_y \leftarrow sc - 1$
29.     else {western side}
30.         $pi_x \leftarrow 0$
31.         $pi_y \leftarrow sc - biws - 1$
32.     end if
33.     $pi_z \leftarrow i \div lc$
34.     move $element_i$ to the position corresponding to $pi$
35. end for
```

The algorithm first computes the side capacity sc (i.e., the number of bricks that can be placed horizontally along a wall's side) that best fits the given number of elements, i.e., the maximum capacity of the building bc_{max} is large enough. The ratio between the buildings' dimensions is set by means of the relation on line 5, which establishes that the minimum number of layers of a building is double the size capacity of the previous category. Then, for each element we compute the position index pi, according to the following variables: layer capacity lc (i.e., the number of bricks that can be placed on an entire horizontal layer), brick index within the layer $biwl$, brick index within the side $biws$, and side index si (i.e., north, east, south, west). The position of each element is computed from its position index and the dimensions of the elements.

To enable a deeper understanding of the algorithm, we illustrated the algorithm by means of a practical example. Figure 3.11 shows the ranges covered by the first four categories of buildings in the *Progressive Bricks* layout from both a top-down (top) and a lateral (bottom) perspective. Each of the columns in the figure represents one of the first four categories, corresponding to $sc = 1..4$. The top part of the column shows the positioning of the bricks and the indices of the bricks belonging to the first layer $biwl$. The lower part of the figure shows, for each column, the configuration (from a lateral perspective) corresponding to the minimum (bc_{min}) and the maximum (bc_{max}) number of bricks for the category, which appear under each configuration. The number of layers corresponding to the bc_{min} value is nol_{min}.

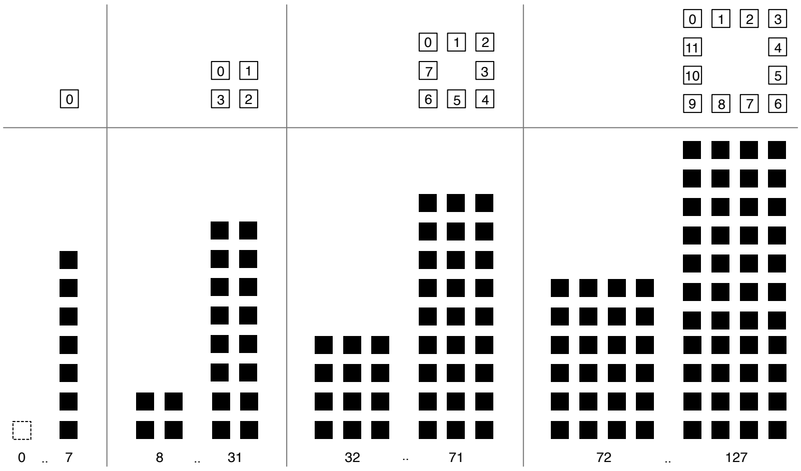


Figure 3.11. The first four levels in the *Progressive Bricks* layout

We exemplify the computation of these key values for the third category in Figure 3.11: $sc(3) = 3$, $lc(3) = 8$, $nol_{min}(3) = 4$, $bc_{min}(3) = 32$, $bc_{max}(3) = bc_{min}(4) - 1 = 71$. These numbers appear in the column corresponding to the third category: There are 8 bricks per layer, as shown by the configuration at the top of the column, a minimum of 4 layers corresponding to 32 placed bricks (bottom of the column, left) and a maximum of 71 bricks distributed in 9 layers (bottom, right) for the category. The value 72 is computed using the same formulas applied to the next category (i.e., where $BPLS = 4$).

To illustrate the improvement of the adaptive layout over the non-adaptive one, we present in Figure 3.12 a parallel between the two in the same screen-estate. The visualized system is Jmol (i.e., a 3D viewer for chemical structures) and the problematic class is `Viewer` with its 750 methods, which has been annotated in both views. In the right part of the figure, this class appears as a building with 10-bricks-wide walls, which continues to stand out due to its unusual volume, without obstructing the overview of the city anymore. Apart from the efficiency of the overview, the code cities resulting after using the algorithm are more habitable than the ones obtained with the conventional layout, as seen in Figure 3.12.

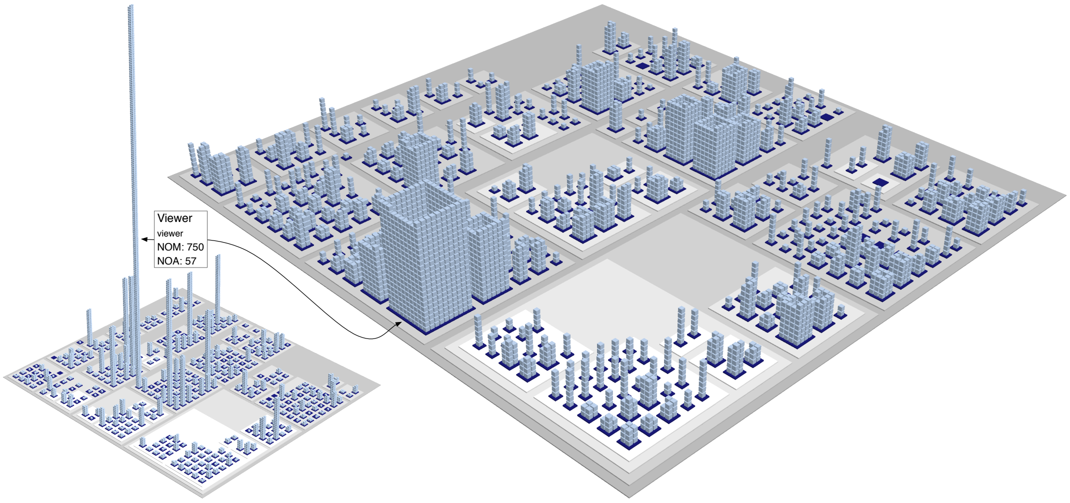


Figure 3.12. Comparison between the *bricks* (left) and *progressive bricks* (right) layouts on Jmol

The building categories can be easily identified from a top perspective of the Jmol city (See Figure 3.13), whose `org.jmol.viewer` district exhibits seven different categories.

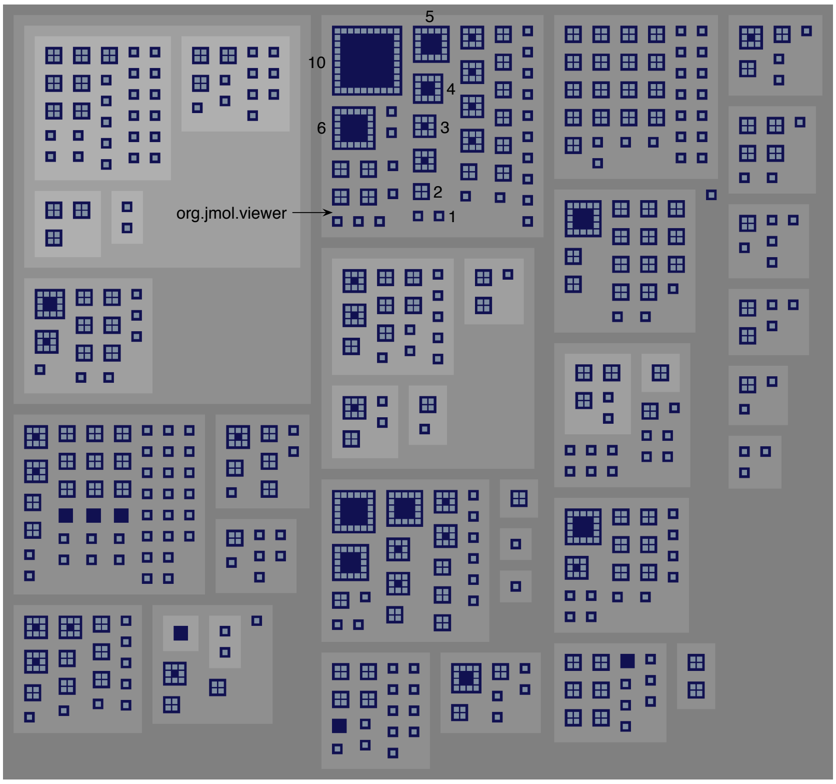


Figure 3.13. A top-down view of the progressive bricks layout in the code city of Jmol

3.3.6 Depicting Relations

Visualizing relations is an open challenge in software visualization. Naive representations of relations, such as direct edges, are not feasible for visualizing software systems, because they lead to over-plotting problems. The reason for this is the difference in cardinality between relations and entities in a software system: One pair of classes may share tens or more relations of different types, such as class inheritance definitions, method invocations, and attribute accesses.

To tackle this problem, we looked into efficient representations for relations. A very promising technique is Holten's bundled edges [Hol06] used in dynamic visualization for program comprehension [CHZ$^+$07]. The technique bundles edges that follow similar routes to reduce complexity. During the history of visualization presented earlier in Chapter 2, we presented an example of bundled edges visualization (See Figure 2.12(b)).

By extending the bundled edges approach to 3D, we obtained intriguing code city visualizations. In spite of the increase in complexity over 2D, there are advantages of using a 3D interactive environment. One of them is the variety of perspectives one can get by navigating the environment, in search for vantage points. By navigating our 3D environment, one is able to look at the code cities in one of the three fundamentally different types of overview perspectives:

1. **Skyline.** This perspective shows the city skyline, i.e., the artificial horizon created by the city's overall structure. Figure 3.14 shows a code city skyline of ArgoUML, whose invocations are visualized as bundled edges.

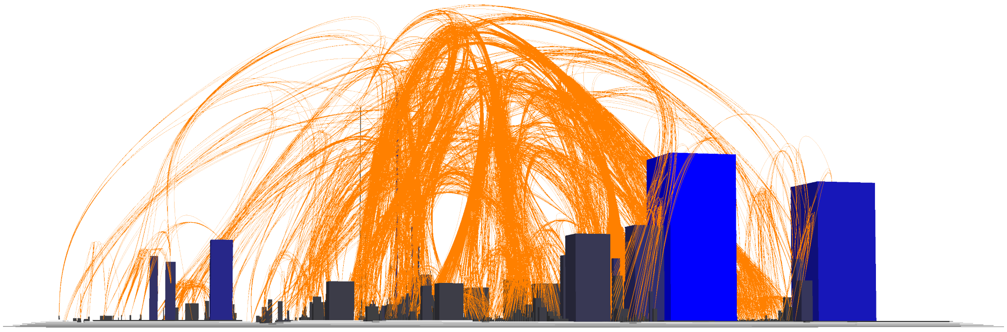


Figure 3.14. Skyline perspective over the code city of ArgoUML

2. **Top**, which shows the city from above, similar to a satellite view. This perspective emulates a 2D visualization of the city's map, which allows an understanding of the overall structure of the system. Figure 3.15 shows the top perspective of the code city of ArgoUML.

3. **Aerial**, which covers the entire range of perspectives between the skyline and the top perspectives. To illustrate the added value of bundled edges for code cities, we show in Figure 3.16 how very different the code city of ArgoUML presented earlier (See Figure 3.4) looks, when all the invocations in the system are visualized as bundled edges.

Although visualizing relations as bundled edges in code cities provides a raw view of the structure of software systems, it raises two main problems. The first is a complexity problem. In spite of the efficiency of the bundled edges, showing *all* the relations may still lead to over-plotting, due to the magnitude of the relations in software systems, in particular in large ones.

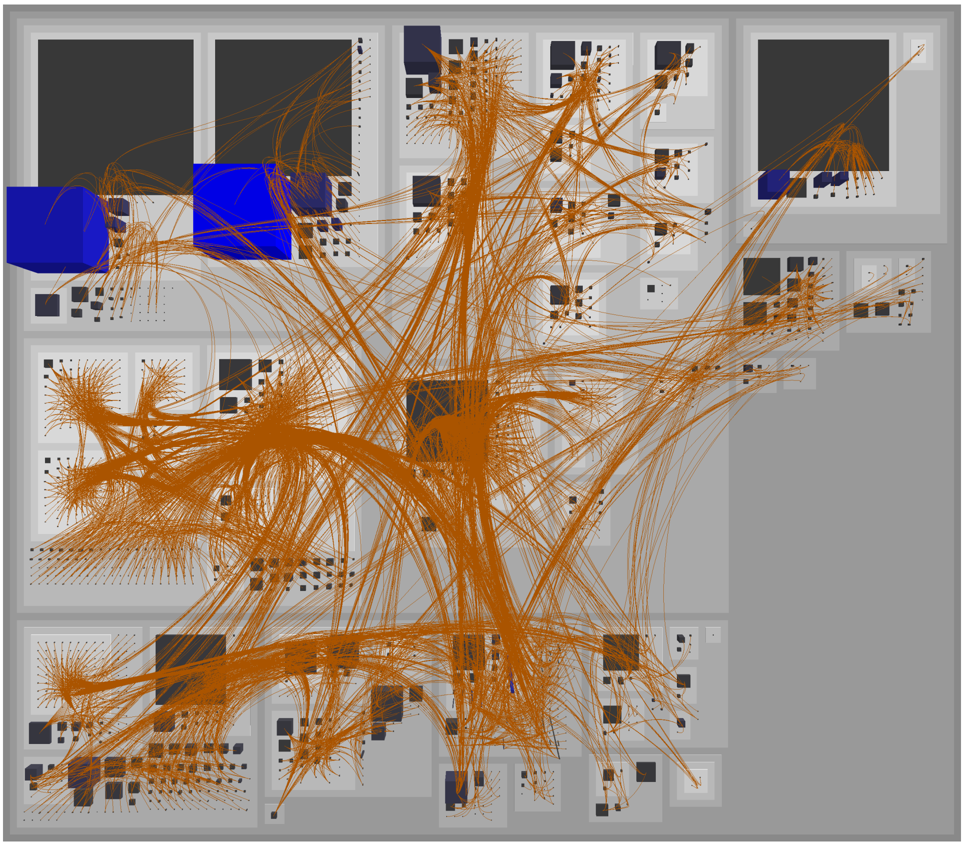


Figure 3.15. Top perspective over the code city of ArgoUML

Although the 3D environment allows the viewer to tackle occlusion by moving around and finding better perspectives, oftentimes the view is just too complex. We looked for solutions to presenting the dependency information, while maintaining the views clean and uncluttered.

One solution to showing dependencies is by means of the entities that share them, i.e., we can perform a query on the dependencies of a certain entity and as a result we see the entities in the system selected.

Another solution we came up with is an opportunistic display of relations, which enables the viewer to show or hide the relations for only one entity or for a reduced set of entities. This on-demand technique addresses the issue of visualizing relations, but avoids the visual clutter that comes with showing all the edges, all the time. This technique maps well on the last part Shneiderman's information visualization mantra: "overview first, zoom and filter, then details-on-demand" [Shn96].

The second problem is that this visual representation of relations is not compliant with the city metaphor, unless we are talking about a futuristic one. This shows one of the limitations of metaphor-based visualization: the creativity of the approach is constrained by the boundaries of the metaphor.

This was only one example of dependencies representation and it was clear to us that we had barely scratched the surface. Although there is plenty of space for improvement, we believe

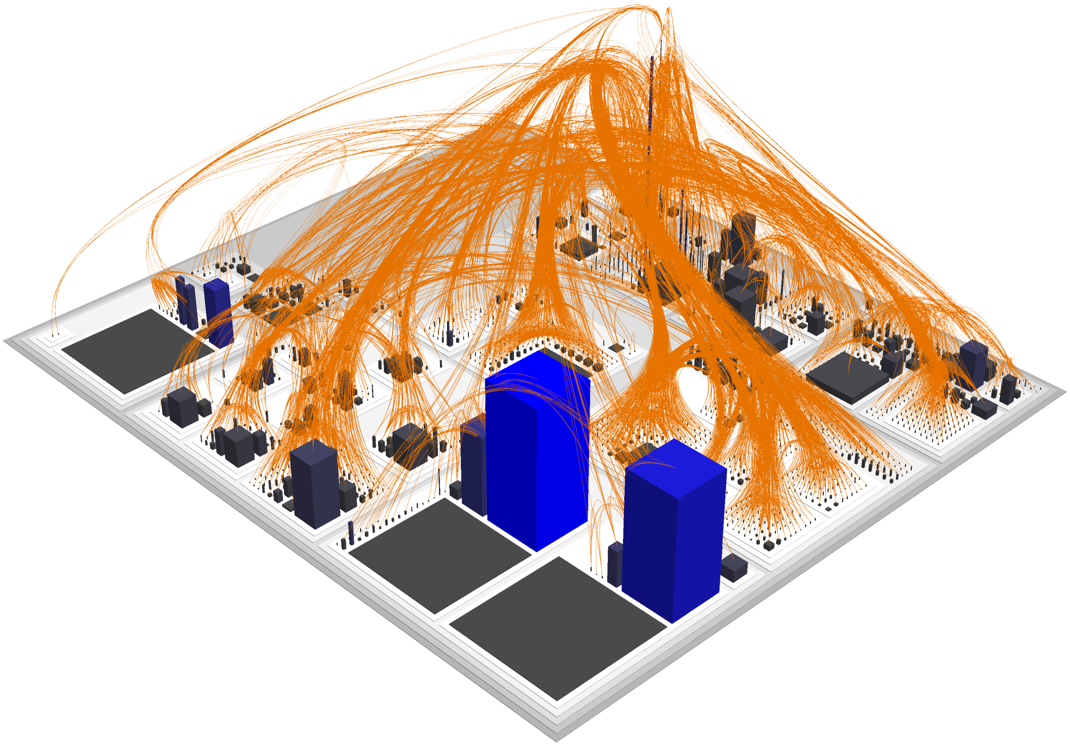


Figure 3.16. Aerial perspective over the code city of ArgoUML

that the visual representation of dependencies is a topic in itself and therefore, we preferred to concentrate our effort on extending our city metaphor to support other facets of software, complementary to the structural program comprehension, i.e., software evolution and design assessment.

At this point, the description of our city metaphor contains all the elements required for applying our approach to supporting program comprehension on case studies.

3.4 Case Studies

To illustrate the first application of our city metaphor to program comprehension, we use two Java systems as case studies. The characteristics of these systems in terms of the number of packages (NOP), number of classes (NOC), and number of lines of code (LOC) metrics are described in Table 3.2.

Name	**Version**	**NOP**	**NOC**	**LOC**
JDK's *java* namespace	1.5	53	1,966	160,287
ArgoUML	0.23.4	96	1,768	136,325

Table 3.2. Case study systems for program comprehension

3.4.1 JDK's java Namespace

We use the `java` namespace of the Java Development Kit (JDK) version 1.5, to illustrate how to interpret the language of the city metaphor. The code city of this sub-system is presented in Figure 3.17.

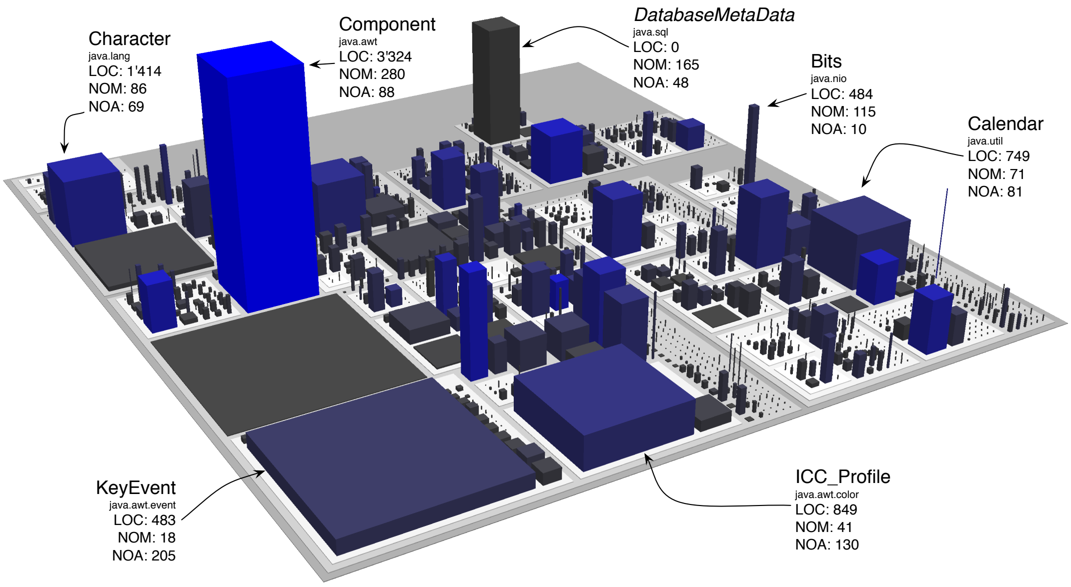


Figure 3.17. The code city of JDK's `java` namespace

From this first overview, we learn that the `java` namespace is a moderately large subsystem, with a shallow package nesting level (i.e., we count at most four stacked district platforms) and a broad variety of classes, in terms of the mapped metrics.

There are wide and flat buildings, similar to parking lots, such as class `KeyEvent`, which is characterized by relatively few methods (i.e., reflected by the low height of the building) and many attributes (i.e., reflected by the building's large base size). The blue shade of the building shows that the 18 methods of this class hide a relatively large number of lines of code, i.e., 483.

One can spot very thin towers, which represent classes with the number of methods much higher (i.e., very tall building) then the number of attributes (i.e., thin). An example of such tower is `Bits`, with 115 methods and only 10 attributes.

Other types of buildings are the massive office buildings, whose massiveness reflects a relatively large number of attributes and an even larger number of methods. An example of massive building is class `Component`, whose intense blue color depicts it as the class with the largest number of lines of code in this namespace, i.e., 3,324.

Moreover, there are also large cubical buildings, which represent classes that have roughly the same number of methods and attributes, such as `Character` or `Calendar`.

Besides various outliers, we can also see how functionality is distributed within packages, for example `java.awt.event` contains classes with a similar amount of functionality (NOM) and state (NOA), with the exception of `KeyEvent`, which has many more attributes (the events for each key are saved as constant attributes).

3.4.2 A City Tour of ArgoUML

Initially, we conceived our city metaphor to primarily support program comprehension activities. The very first experience with our approach was trying to gain insights into a software system, based on its most recent version [WL07a]. In this context, we decided on ArgoUML[3], the leading open source UML modeling tool, whose most recent version at the time was 0.23.4.

First impressions are lasting impressions: The city overview gives us a first sense of both the magnitude and the structural complexity of a software system, which is a first step towards building a mental model required for understanding the "big picture". Moreover, this first contact influences one's decisions on where to start the investigation of a software system. The code city in Figure 3.18 depicts ArgoUML as a fairly large system, composed of many classes built on top of a complex package hierarchy, in which the functionality seems to be distributed rather heterogeneously, as depicted by the variety of building heights.

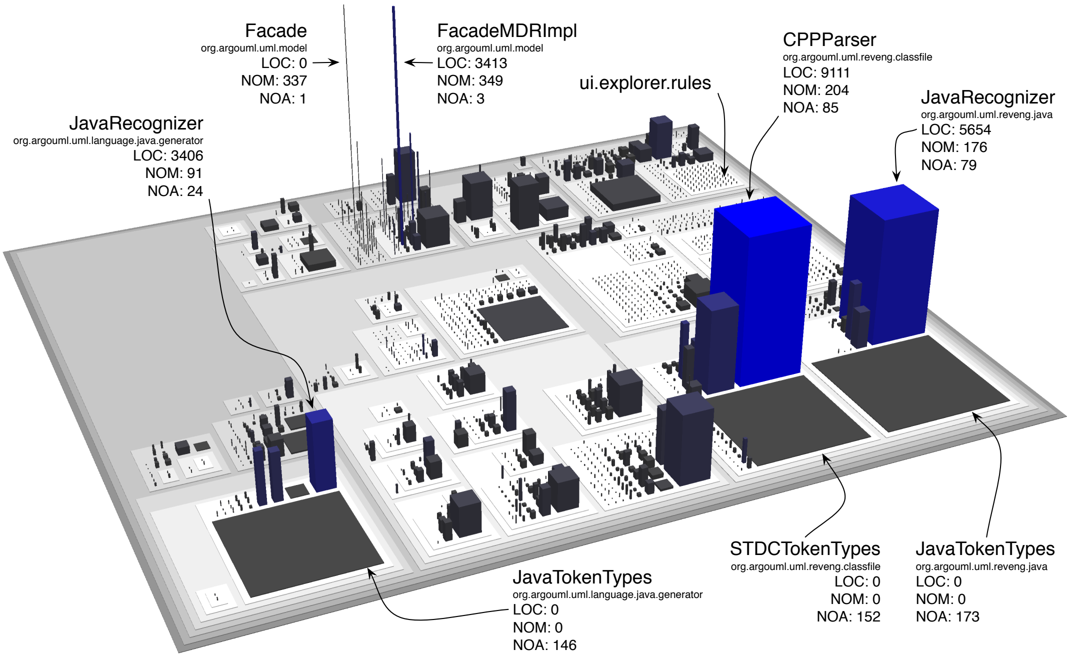


Figure 3.18. A glimpse in the code city of ArgoUML

We can identify a number of hot spots, pointed out by several striking buildings, which represent classes whose LOC, NOM, or NOA metric values qualify them as outliers: two antenna-shaped skyscrapers, which are the tallest buildings in the city, and three parking lots that each have at least one large office building in their vicinity. There are also interesting districts that do not exhibit any outlier building, yet stand out precisely for this reason, e.g., the `ui.explorer.rules` district which is made exclusively of small houses. Next, we take a closer look at each of these interesting artifacts, annotated in Figure 3.18 with the values of the software metrics mapped on their visual properties.

[3] `http://argouml.tigris.org`

The Antenna-Shaped Skyscrapers

The appearance of these two tall and thin buildings indicate that the two underlying classes have many methods and few attributes. Moreover, the high resemblance of the two buildings may indicate a relation between the two entities.

A closer look at the underlying data reveals interesting details. The leftmost antenna-shaped building in Figure 3.18 is `org.argouml`[4]`.uml.model.Facade`, an interface with 0 lines of code (i.e., a normal situation for an interface), 337 methods, and 1 attribute. The rightmost antenna-shaped building in Figure 3.18 is `uml.model.mdr.FacadeMDRImpl`, a class with over 3,400 lines of code, 349 methods, and 3 attributes. And indeed, there is a relation between the two: As the name suggests, the class `FacadeMDRImpl` implements the enormous `Facade` interface. The similar height of the buildings, hence the similar number of methods of the two software entities is rooted in the fact that in Java, a class that implements an interface is contractually bound to provide implementations for all the methods defined in the interface.

In a code city, a class implementing an interface is represented by a building at least as tall as the building that represents the interface. An interesting insight in this context, which is revealed by the city skyline perspective in Figure 3.19 is that `FacadeMDRImpl` is the only class that implements the huge `Facade` interface.

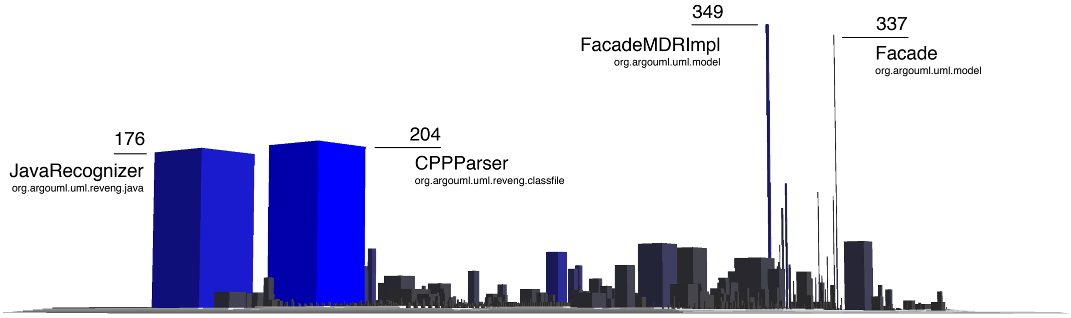


Figure 3.19. Tallest buildings in the code city of ArgoUML

A Java interface is used to either provide a flexibility point for further extensions, as in the "Program to an interface, not an implementation" principle [GHJV95], or to group together a set of global attributes which are made accessible to the classes that implement the interface. The `Facade` interface clearly falls in the first category, given that it defines a huge number of methods. However, implementing the interface in just one class seems pointless and defeats the purpose of this practice. Why is there only one class implementing the `Facade` interface?

While trying to find a rationale for this apparently questionable design, we have formulated several hypotheses. The first hypothesis was that, during the history of the system, there were other classes that implemented `Facade`, which are not parts of the system anymore. Second, it could reveal the developers' intention of writing further implementations of the `Facade` interface. Third, it may just be a poor use of the interface mechanism. However, without further information about the history of the system, all these remain speculations.

Modifying an interface with the magnitude of `Facade` or any class that implements it, is very likely to cause changes in many other classes that depend on one or more of its numerous methods.

[4] From here on, we omit the common prefix `org.argouml` from package names and qualified class names in ArgoUML.

We use our opportunistic edge representation to display only the incoming invocations of the Facade interface. The top perspective in Figure 3.20 shows that more than one third (i.e., 767 out of 1,817) of ArgoUML's classes depend on this interface and its implementing classes, which makes changing this interface a maintainer's nightmare.

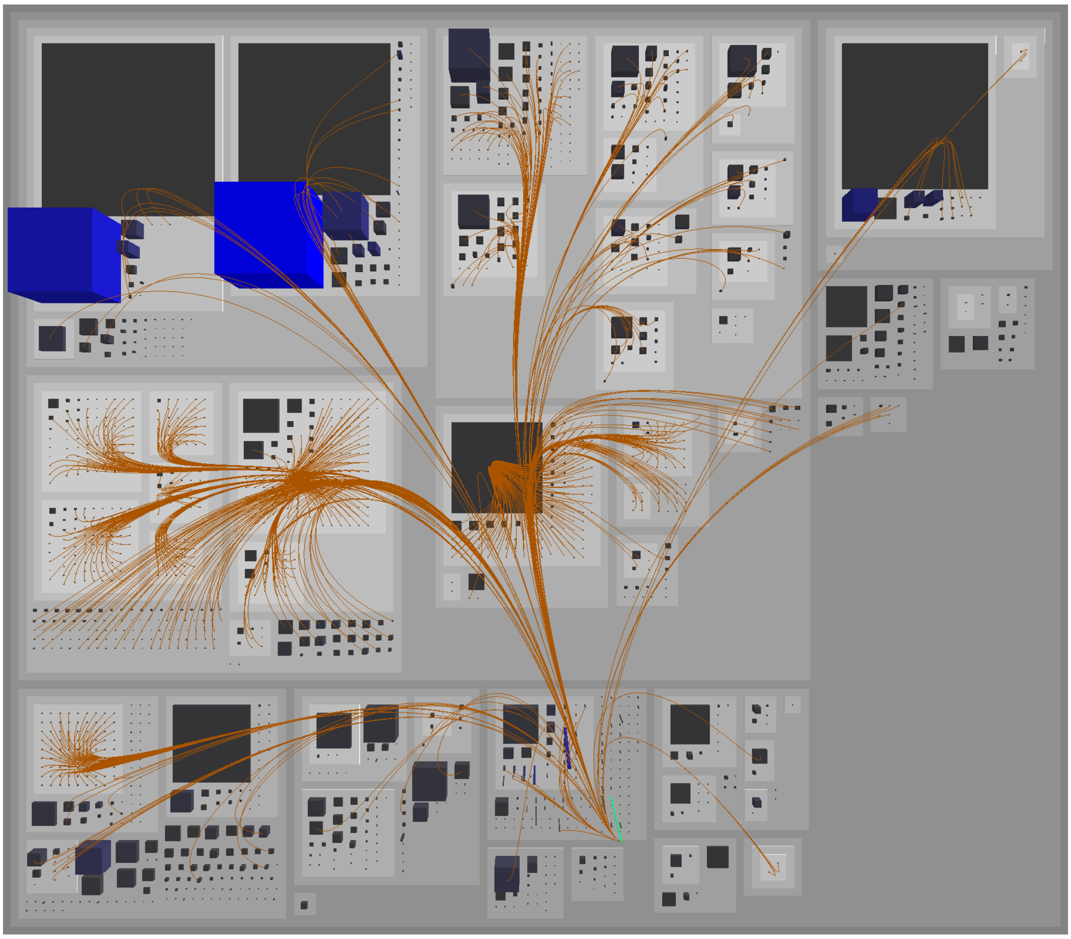


Figure 3.20. The methods defined in Facade are popular in ArgoUML

The Office Buildings and the Parking Lots

Since buildings that look like parking lots, i.e., wide and flat, depict classes or interfaces with many attributes and few methods, we analyze them in the context of the classes that access their attributes.

One of these parking lots is an interface called JavaTokenTypes, with 0 methods and 173 attributes, defined in package uml.reveng.java. The only class that accesses the interface's attributes is JavaRecognizer, a huge class (i.e., with 5,654 lines of code and 176 methods) defined in the same package. The role of the uml.reveng.java package is to support the parsing of Java source code.

Next, we move to the next instance of this visual pattern, manifested through two neighbor buildings, one massive and the other wide and flat. The visual similarity with the previous pair is by no means accidental. The parking lot is another token-related interface, called `uml.reveng.classfile.STDCTokenTypes`, with 0 methods and 152 attributes that represent tokens found in the standard C language. And similarly to its Java peer, the interface's attributes are exclusively used by another huge (i.e., having 204 methods and 85 attributes) parser class for C++, called `uml.reveng.classfile.CPPParser`. Confident enough that this is the package that provides the parsing support for C++ source code, we continue with the next pair.

With some expectations to find a third parsing duo, we turn our attention to the last instance of this pattern, located in package `language.java.generator`. To our surprise, we discover another interface called `JavaTokenTypes` with 0 methods and 146 attributes and a unique data accessor called `JavaRecognizer` with 91 methods and 24 attributes.

As illustrated by Figure 3.21, each of the three parking lots is used exclusively by the large nearby office building, i.e., the numerous attributes in each of the three interfaces are only accessed by the large parser class defined in the same package.

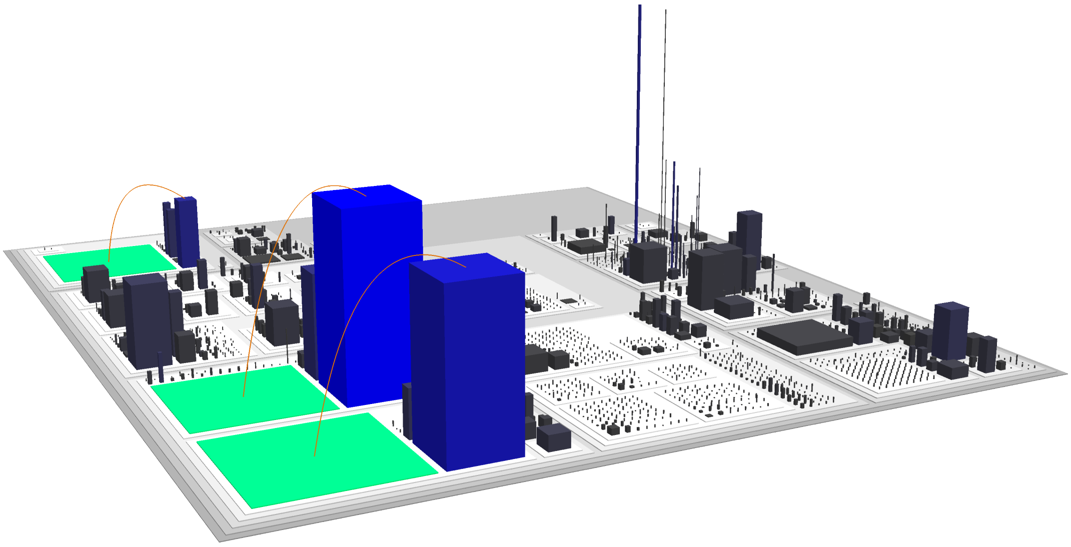


Figure 3.21. Office buildings exclusively served by their private parking lots

The two pairs of classes and interfaces defined in two different packages share more than just names; they also share a significant amount of duplicated code. The two `JavaTokenTypes` have 145 duplicated attributes, while the two `JavaRecognizer` share 88 methods and 22 attributes. A closer look at the source code showed that these classes are very likely generated, i.e., built automatically by a programming language parser generator.

From an economical perspective, this insight is reassuring, because it implies that these classes are not being manually maintained. However, from a logical point of view, we do not know the reason behind this situation. It is possible that they are the result of an incremental migration of a class hierarchy from one package to another, with both source and destination coexisting in the same version of the system. However, getting a clear idea of the situation would require additional information, such as the rich data buried in the history of the system.

The Suburbs

We complete our tour in the green district[5] made entirely of small houses, shown in Figure 3.22.

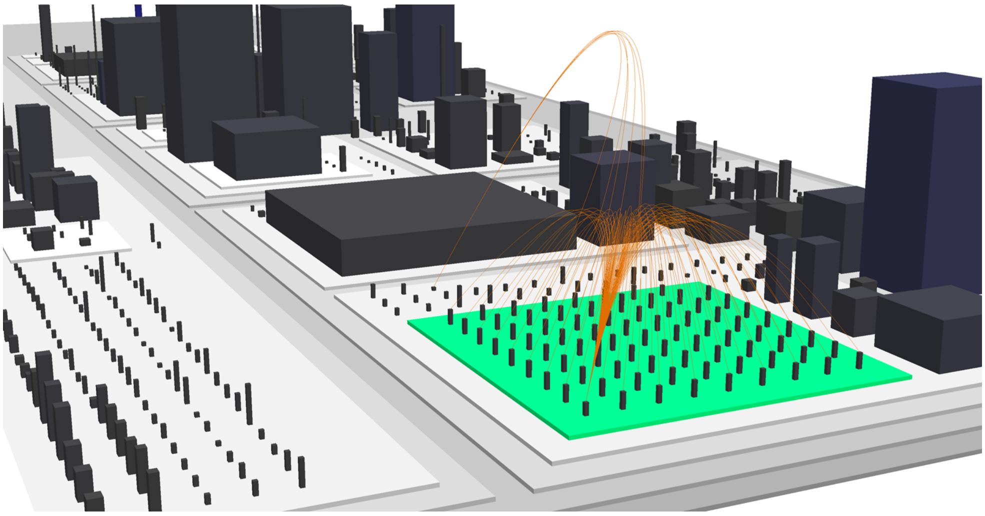


Figure 3.22. `ui.explorer.rules`, a package made of mainly sibling classes

This suburb represents `ui.explorer.rules`, a package containing 80 classes, all featuring roughly the same number of methods and attributes. The main hierarchy implemented in this package is made of the abstract class `AbstractPerspectiveRule` and its 72 subclasses. It turns out that `AbstractPerspectiveRule` is the only class in the system that implements the interface `PerspectiveRule`, which is suboptimal: If there was no common functionality, the interface would suffice; in the opposite case, the abstract class could take over the contract from the interface. Given that the abstract class only overrides the very common `toString()` method in `Object` and superfluously declares as abstract methods two of the three methods already declared in the interface, the obvious solution is to declare the remaining method from the interface as an abstract method in the abstract class and remove the interface.

Most of `AbstractPerspectiveRule`'s subclasses provide only the implementation for the three abstract methods in their superclass, which is why all these buildings have the same height. However, a closer look at the source code showed that even these classes, in spite of the low complexity, have their share of design problems. First, there are some *NOP* method implementations (`return null`), which imply that the method makes sense in only some of the subclasses, which questions the current structure of this class hierarchy. Second, there are many type checks in these methods, which is a sign of bad object-oriented design. According to design guidelines, type checks should be removed by using subclassing and polymorphism. In this package, the type checking is performed both directly (i.e., via the `instanceof` operator) and indirectly, by using the methods of the notorious `Facade`.

This takes us back to `Facade`, where we find that dozens of its methods are only meant to perform type checking, such as `isAAssociationRole`, `isAState`, `isAStereotype`, or `isANode`. This insight might show the intention of the developers to keep the "bad" code in one place, in spite of occasional direct type checks outside `Facade`.

[5] An artifact which has been selected in CodeCity is colored in bright green.

3.4.3 Analysis Summary

To illustrate the types of insights one can build using our visualization approach for program comprehension, built on the presented city metaphor, we presented two case studies.

The first case study consisted in the *java* namespace, a sub-system of JDK, which we used to illustrated how one interprets the information presented in a code city and how one is able to seek for artifacts according to different criteria, relatively to the displayed metrics.

The second case study was ArgoUML, which we used to perform a more in-depth analysis. The city overviews allowed us to identify and localize some of the system's outliers in terms of size. Based on these first impressions, we looked closer at some of them and discovered interesting insights. We learned that ArgoUML contains a large interface with an extremely high change impact and only one implementation. Next, we learned about the Java and C++ language support in ArgoUML and discovered a strange duplication in the Java part. Finally, we learned that even small packages are prone to design problems.

The city tour of ArgoUML gave us a first positive indication of the metaphor's usefulness in the context of program comprehension. Moreover, it showed that performing investigations based on only one version of a software system is not sufficient and that we needed to investigate the history of the system, which eventually led to the second application context of our city metaphor, i.e., software evolution.

3.5 Related Work

Although there are several 3D software visualization approaches, only few of them use a metaphor. In fact only 1% of the software visualizations approaches are metaphor-based [Kos03]. Among these metaphor-based 3D visualization approaches, only a handful use a city-like metaphor.

Before presenting the closely related work, we briefly describe several contributions, only remotely related to our approach, i.e., they are only related to particular aspects of our work.

3.5.1 Remotely Related Work

Andrews et al. created a 3D visualization of file systems called the Information Pyramids [AWP97]. In this work, directories are represented as platforms, while files are depicted as cubes, colored according to a particular measure of the file, e.g., file type, age. Our visual representation resembles the representation of the Information Pyramids.

DeLine proposed a software visualization approach called Software Terrain Map [DeL05], based on the metaphor of cartographic map. In this work, DeLine aims at helping developers avoid disorientation in IDEs, by providing an overview of the system enriched with additional cues specific to the development process, such as the currently edited part of the system, or the program execution path up to the breakpoint in the debugger. The main similarity between our approaches consists in the fact that they rely on the particularities of the analyzed system to produce a memorable "big picture" of the system. However, our approaches are not comparable, because they are based on different metaphors and support different types of activities (i.e., program comprehension and reverse engineering, as opposed to forward engineering).

A more recent work related to maps, is the work of Kuhn et al. on software cartography. The main contribution of this approach is that it uses the vocabulary extracted from the source code to compute the locations of the elements on the map, and thus obtain a layout which is resistant to evolution [KLN08], because the vocabulary of the domain is not very likely to change.

Another work remotely related to our approach is the work of Lange, who proposed a collection of views, similar to UML diagrams and enriched with software metrics [LC07]. One of these views, called the UML-city, was presented earlier in Figure 2.16(b). This visualization is similar to a certain extent to a code city, in that it has structures similar to the buildings in our approach with software metrics mapped on their properties. Unlike our approach, the buildings are placed on UML diagrams, while in our case the terrain represents the package hierarchy.

3.5.2 Closely Related Work

Knight and Munro [KM00] proposed a visual approach based on a world metaphor, called Software World (See Figure 2.14(a)). They presented a representation for Java systems, in which classes were represented as districts and methods as buildings. Our work differs from the work of Knight et al. in several aspects. First, their approach is limited to Java systems, while ours is based on a language-independent meta-model, which enables us to apply our approach to system written in various languages (e.g., Smalltalk, Java, C++, C#). Second, the domain mapping they chose is suboptimal (i.e., each method is a building), because it leads to very large cities for even small systems and thus raises the issue of scalability. Moreover, this work completely omits the representation of packages, which is a key concept for the organization of object-oriented software systems.

Two years later, the same authors joined by Charter and Thomas extended the Software World approach work from source code to components (i.e., units of executable code, which can take the form of packages, libraries, frameworks), in a work called Component City. According to their approach, the buildings represent components in this approach, while the districts are functionality groups of components. In Component City, there are three distinct types of buildings which encode the number of components: i.e., houses (1 component), mansions (2 components), and skyscrapers (more than 2 components). Moreover, there are monuments which act like landmarks, i.e., they help users getting oriented. Similarly to this approach, we attempted reducing the number of building types for our box plot based and threshold-based mapping strategies. The main difference between our approach and the work of Charter et al. lies in the domain mapping, i.e., the representation granularity in Component City is coarser than the one in our approach. Given that classes and packages are the main orientation points for the object-oriented paradigm, we believe that our domain mapping offers a good tradeoff between the information density and the scalability of the approach. Moreover, although the authors describe a scenario in which a user navigates the environment to search for a specific feature, they do not accompany their narrative with any screenshots of the tool in action, as one would expect. In addition, as with the other approach, the tool is not available either. We believe that every software visualization approach should demonstrate how it facilitates understanding of the software system, at least by means of case studies.

Marcus et al. proposed an approach called sv3D (i.e., Source Viewer 3D), based on an abstract geometric metaphor [MFM03]. We presented an example of sv3D visualization in Figure 2.14(b), in the context of the history of visualization presented in Chapter 2. The main representation in this approach is the poly cylinder and several poly cylinders are grouped together in floating containers. Since the representation looks similar to a landscape, we consider it related work. Marcus et al. extended the fine-grained SeeSoft approach by Eick et al. [ESEE92] through a 3D representation. Each poly cylinder represents a line of code and the containers represent files. We believe that the granularity of this mapping is not appropriate for software systems, because it does not scale well: The visualization of an even relatively small system of

50 kLOC implies the manipulation of 50'000 graphical objects. The interaction with the visualization is closer to a graphical editor than to an exploration environment: The user is allowed to move, rotate, and scale the graphical representations. Other interactions, such as queries, are only mentioned as future work. The similarities between our approach and the work of Marcus et al. are limited to the mapping of software metrics on the visual properties of very simple 3D graphical elements. The main difference between our approaches is that Marcus et al. aim to support source code analysis, while our goal is to support higher-level analyses of software systems. Consequently, in terms of scalability, the approach of Marcus et al. has been found to work well up to 40-50 kLOC, while our approach, with its coarser-grained representation, has allowed us to visualize larger software systems (i.e., up to almost 3 MLOC). Another advantage of our approach over this work is the representation of the package hierarchy. Moreover, in our approach there is a clear notion of locality, which helps the users get oriented in the 3D environment, as opposed to the approach based on the concept of poly cylinder, in which artifacts can be freely moved.

Software Landscapes is a visualization approach, proposed by Balzer et al., based on a landscape metaphor [BNDL04]. Using a granularity level similar to ours, this approach represents the hierarchy of packages as nested spheres, classes as circular discs, and the methods and the attributes as cuboids on top of the discs (See Figure 2.15(a)). For the representation of relations, the authors proposed an elegant solution called Hierarchical Net, which depicts relations as hierarchically routed connections. Balzer et al. visualized Java systems of different sizes, including a version of Eclipse with over 1 MLOC. This approach uses the level of detail technique, which renders only the details of elements that are close enough to the viewer. While this work represents an exquisite contribution, there are aspects in which we feel our approach has advantages. The first is that our approach makes use of software metrics to provide a "big picture" of the system that goes beyond just plain structure. In the case of the Software Landscapes, the use of software metrics has only been mentioned as future work. Moreover, due to their level-of-detail based navigation, it is difficult to build a mental overview of the system, since at any time details are only partially visible. Although Balzer et al. presented screenshots of their visualizations applied to several Java systems including large ones, they only showed these examples to illustrate the metaphor. However, none of the works on Software Landscapes [BNDL04, BD04] contains any demonstration of the use of the Software Landscape approach to gain insights in any of the described systems. This is unfortunate, because after all, the main goal of software visualization is to enable the understanding of software systems.

Another contribution involving a city metaphor is the work of Panas et al., who initially envisioned a city metaphor [PBG03], in which the city is a package and contains, for increased realism, elements with no semantic load, such as trees or street lamps, aimed to facilitate navigation. The metaphor would not only represent static, but also dynamic data, i.e., a program run would be represented as cars which leave traces from the origin to the destination. Although never implemented quite in this form, this concept metaphor represented a good starting point for Panas's later work.

Two years later, Panas et al. presented a concrete 3D visualization architecture implemented in a configurable tool called Vizz3D [PLL05], which supported supported several metaphors, including a city metaphor. Similarly to our approach, the authors employ the use of software metrics and strive for configurability. Differently from our approach, in which we focus on the extensive exploration of the city metaphor, Panas et al. deal only superficially with metaphors, as part of their complex visualization architecture. Unfortunately, this work does not contain any practical application of the approach to program comprehension, either.

After two more years, Panas et al. [PEQ+07] present an approach called multi-aspect single-view architectural visualization (See Figure 2.15(b)), which unifies the presentation of various kinds of architecture-level information to different types of stakeholders. The underlying idea was to present the structure of the system using a graph-based model and augment the model with a number of static and dynamic analyses for C and C++ programs. However, the illustration of this approach by means of real case studies is completely absent, the authors limiting themselves to imagine a number of scenarios in which their approach could be useful. In the absence of any demonstration of the application of their approach to a case study, it is difficult to make head-to-head comparisons between our approaches.

A somewhat similar approach to ours is the one of Langelier et al. [LSP05], who use 3D visualization to display structural information, by representing classes as boxes with metrics mapped on height, color and twist, and packages as borders around the boxes, which are placed using a tree layout or a sunburst layout (See Figure 2.16(a)). To support their approach, the authors implemented a tool called Verso. Unlike our layout, these layouts do not enable an easy visual interpretation of the package hierarchy. Similarly to our approach, Langelier et al. demonstrate, by means of case studies, the usefulness of their approach. Although the authors do not present their approach as driven by a city metaphor, they briefly address this issue.

In the context of IDE integration, Biaggi presented Citylyzer [Bia08], an Eclipse plugin based on our city metaphor, which supports the program comprehension application context.

Finally, several researchers collaborating with us in the context of the EvoSpaces project [ABW+09] have contributed to the field with work which is implicitly related to our approach.

The work of Alam and Dugerdil represents their vision of the city metaphor [AD07], with an implementation which is integrated in the Eclipse IDE and design decisions that differ from ours (e.g., using textures to distinguish between building types). A later application of their approach is the visualization of execution traces, based on the same city metaphor [DA08].

Another line of work from the same project is the work of Boccuzzo and Gall, who explored other metaphors [BG07] than the city metaphor to find the most appropriate one for our project. Later, Boccuzzo and Gall took an interesting take on reverse engineering, by enriching visualization with audio support [BG08].

3.6 Summary

We defined a city metaphor for software visualization, according to which software systems are represented as cities, packages as districts, and classes as buildings. We described the details of our metaphor in terms of property mapping strategies, novel layouts, and representation granularities, and discussed the advantages and disadvantages of each visualization technique.

The first step in demonstrating the versatility of our city metaphor, was to apply it in the context of program comprehension. A code city visualization provides an overview of an entire software system and allows the user to identify outliers in terms of the mapped metrics. We illustrated the program comprehension application by means of two case study. We performed an analysis of the ArgoUML software system, by exploring its code city. The insights we picked up during a short tour through the city of ArgoUML confirmed us the usefulness of the city metaphor in the context of program comprehension, which supports the first claim of our thesis.

However, we feel that many of the raised questions that were left unanswered could find an answer in the system's past versions. Therefore, in Chapter 4, we turn our attention towards extending our approach to enable the visualization of software system evolution.

Chapter 4

Visual Analysis of System Evolution

4.1 Introduction

One of the lessons we learned from applying our approach based on the city metaphor to program comprehension was that looking at only the most recent version of a software system is not enough: We need to broaden our perspective, by looking at the system's history, i.e., the sequence of transformations that a complex system went through to get to its current state. As a result, the second application of our approach, which we use to demonstrate the versatility of the city metaphor, is software evolution.

Visualizing evolving software raises a number of challenges. The first one is scalability, which is severely affected by the amounts of data extracted from a sequence of versions of the same system. Another challenge is finding a visualization able to illustrate the process of system evolution itself.

In this context, we present a set of elaborate interactive 3D visualizations which illustrate the structural evolution of large software systems at both a coarse-grained and a fine-grained level. The visualizations make the complex and intangible process of software evolution tangible and visible, and allow for insights into a system's evolution.

We illustrate this second application of the city metaphor by means of three case studies from two different perspectives. First, we look at one system that we have previously studied in the context of program comprehension (See Chapter 3) and whose evolution we expect to reveal insights complementary to the ones we previously acquired. Second, we look at the histories of two systems of which we have zero knowledge.

4.2 Modeling Software System History

For modeling software system evolution, we use the Hismo meta-model proposed by Gîrba [Gîr05], which extends the FAMIX meta-model with historical data.

In Hismo, the evolution of an element is captured by a model history, which is composed of a sequence of versions. Each version is built around an element snapshot, which captures the element's state at a certain moment in time. Figure 4.1 shows how the main structural entities of an object-oriented system (i.e., packages, classes, methods, and attributes) are modeled with Hismo.

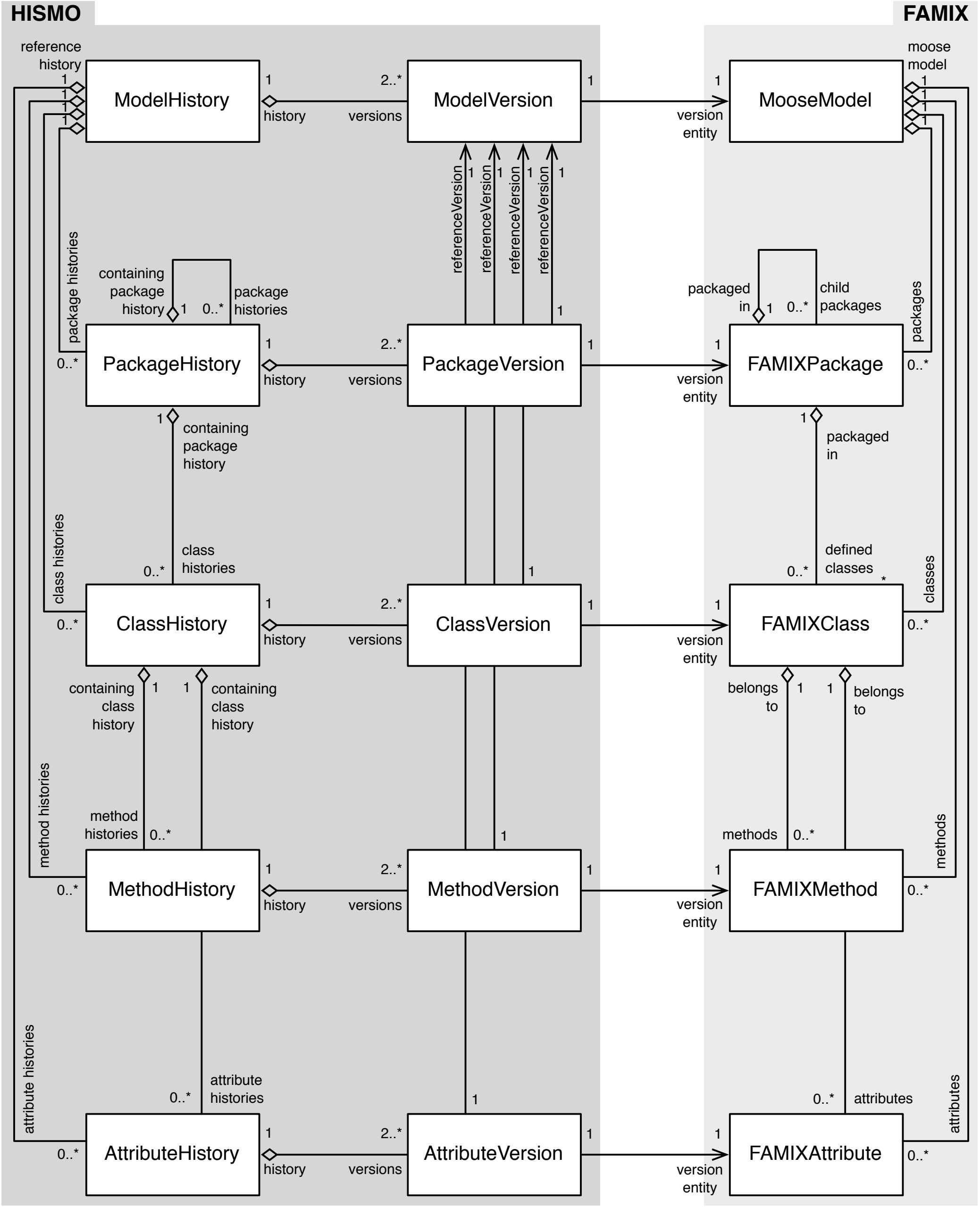


Figure 4.1. Modeling the history of object-oriented systems with Hismo

The rightmost column of Figure 4.1 is the snapshot layer, which is provided by FAMIX. The middle column is the version layer, which links the history layer on the left to the snapshot layer.

For the program comprehension application of our metaphor, the underlying model was limited to the snapshot layer. For the evolution analysis application, the underlying model of the cities is the version layer, which provides access to both the snapshot and the history layer.

4.3 Overview of the Approach

Each of our evolutionary visualizations is characterized by the granularity of the representation and by the technique (or combination of techniques) applied to reveal a particular evolutionary aspect of the system under investigation. We have experimented with two levels of granularity for the representation:

1. *Coarse-grained*, in which classes are represented as monolithic buildings.
2. *Fine-grained*, in which the representation is refined by representing the methods as pieces of the buildings' walls (i.e., bricks).

For visualizing the evolution of software systems, we conceived three visualization techniques:

1. *Age map*, for depicting the age distribution,
2. *Time travel*, for stepping through the system's history, and
3. *Timeline*, for capturing the entire evolution of a software artifact in a single view.

We introduce the different combinations of techniques and granularity levels according to the sequence described in Table 4.1, using a structure which contains the following information:

- a *description*,
- a set of comprehension *goals*,
- an exemplification of its *application*, illustrated by means of one or more of the case study systems,
- an optional "*reality check*", which implied discussions with the actual developers of the visualized systems, aimed at confirming or denying our findings, and
- a discussion of the *drawbacks*, which typically leads to another combination that solves the discussed shortcomings.

			Technique	
		Age map	**Time travel**	**Timeline**
Granularity	**Coarse**	Section 4.5	Section 4.6	-
	Fine	Section 4.7	Section 4.7	Section 4.8

Table 4.1. Roadmap for presenting the techniques at each granularity level

Before presenting the five combinations of visualization technique and representation granularity, we describe the three system histories we used as case studies for the application of our city metaphor to software evolution.

4.4 Case Studies

To illustrate the application of our approach to evolution analysis, we use the histories of three open-source Java systems as case studies.

ArgoUML is a UML modeling tool, whose version 0.23.4 we analyzed in Chapter 3, in the context of program comprehension. To shed light on the questions left unanswered from the aforementioned analysis, we chose to analyze the system's evolution. Instead of performing a periodic sampling of its history, we took a developer-oriented perspective and built a history from the system's major releases, including the one released after version 0.23.4. The eight versions, covering a period of four years and a half, are described in Table 4.2.

Sample no.	Rel. no.	Rel. date	NOP	NOC	LOC
1	0.10.1	Oct 2002	77	838	64,865
2	0.12	Aug 2003	90	937	71,903
3	0.14	Dec 2003	93	1,249	79,253
4	0.16	Jul 2004	97	1,240	85,601
5	0.18.1	Apr 2005	89	1,409	105,405
6	0.20	Feb 2006	97	1,647	148,751
7	0.22	Aug 2006	97	1,701	133,632
8	0.24	Feb 2007	98	1,776	138,468

Table 4.2. The history of ArgoUML's major releases in numbers

JHotDraw[1] is a 2D graphics framework. In a first step, we sampled the four-years history of JHotDraw, using a 24-weeks sampling period and ended up with the eight versions described in Table 4.3. In a second step, to better understand a particular part of the system's evolution, we sampled its history using a one-week period and, after removing the duplicates[2], we obtained 57 unique samples.

Sample no.	Rev. no.	Rev. date	NOP	NOC	LOC
1	18	Oct 2000	15	177	8,413
2	21	Mar 2001	15	190	8,923
3	31	Sep 2001	15	194	9,011
4	41	Mar 2002	15	265	12,552
5	47	Aug 2002	22	384	18,651
6	155	Jan 2003	37	578	29,445
7	217	Jul 2003	37	586	29,969
8	230	Jan 2004	37	600	29,546

Table 4.3. The sampled history of JHotDraw in numbers

Jmol[3] is a 3D molecular viewer for chemical structures. We sampled its eight-years evolution using an eight-weeks sampling period and obtained a history containing 51 unique samples, presented in Table 4.4.

[1] `http://sourceforge.net/projects/jhotdraw`

[2] It is possible to obtain duplicate snapshots of the system using a periodic sample, in the case in which there were no commits in the period between the two consecutive sample dates.

[3] `http://sourceforge.net/projects/jmol`

Sample no.	Rev. no.	Rev. date		NOP	NOC	LOC
1	69	Dec	1999	7	285	17,594
2	82	Feb	2000	7	289	17,766
3	94	Apr		7	290	17,852
4	141	May		7	297	18,496
5	148	Jul		7	298	18,604
6	162	Sep		7	295	18,185
7	180	Nov		7	318	19,654
8	206	Jan	2001	7	325	20,203
9	212	Mar		7	327	20,351
10	242	Apr		5	289	17,997
11	295	Jun		5	296	18,592
12	334	Aug		6	279	17,536
13	335	Oct		6	281	17,640
14	340	Dec		6	279	17,499
15	352	Feb	2002	6	279	17,499
16	385	Apr		7	268	18,882
17	498	May		7	277	19,822
18	500	Jul		7	277	19,831
19	538	Sep		7	295	20,457
20	641	Nov		7	324	24,432
21	780	Jan	2003	11	351	26,873
22	976	Mar		13	379	29,388
23	993	Apr		13	378	29,309
24	1,171	Jun		15	402	32,297
25	1,275	Aug		19	384	30,813
26	1,525	Oct		21	408	33,869
27	1,717	Dec		21	432	36,278
28	1,835	Feb	2004	21	423	36,763
29	2,128	Mar		19	297	24,729
30	2,307	May		19	311	26,680
31	2,486	Jul		19	333	28,067
32	2,584	Sep		19	336	28,730
33	2,748	Nov		18	340	30,319
34	3,116	Jan	2005	17	374	33,760
35	3,377	Feb		18	397	37,102
36	3,508	Apr		21	418	39,490
37	3,716	Jun		22	423	40,331
38	3,927	Aug		22	432	42,268
39	4,208	Oct		23	449	45,208
40	4,304	Dec		24	455	46,055
41	4,414	Jan	2006	26	452	47,240
42	4,709	Mar		26	450	51,641
43	5,154	May		27	452	48,533
44	5,319	Jul		27	452	48,347
45	5,480	Sep		27	452	48,347
46	6,098	Nov		29	503	67,265
47	6,555	Jan	2007	29	508	71,782
48	6,932	Feb		30	516	76,189
49	7,458	Apr		37	550	81,506
50	7,882	Jun		49	549	83,514
51	8,065	Aug		50	558	84,984

Table 4.4. The sampled history of Jmol in numbers

4.5 Coarse-Grained Age Map

Description. We overlay the city representing a version of a system with colors mapping the age of the artifacts. The age represents the number of sampled versions that the artifact "survived" up to the reference version. To encode numerical values, we designed a sequential color scheme, ranging from light yellow (i.e., new-born entities), passing through a spectrum of greens, all the way to dark blue (i.e., oldest entities).

Choosing the color scheme. This color scheme is based on several guidelines for the use of colors in computer graphics [Mac99]. First, it is known that a large percent of the human population (i.e., 8% of the males and 1% of the females in Europe and North America) suffer from at least one form of color deficiency, and therefore, have difficulties in discriminating colors by hue only. To address this issue, the two ends of our color scheme's spectrum are very different in terms of both hue (i.e., yellow vs. blue) and luminance (i.e., light vs. dark). Moreover, the luminous efficiency function of CIE (i.e., the International Commission of Lighting) peaks at a wavelength of 555nm, corresponding to a greenish-yellow hue, which represents the hue to which the human eye is most sensitive. Incorporating the yellow–green range in the spectrum of our color scheme allow us to maximize the range of values that can be encoded with our color scheme, such that they can be discriminated by the human eye.

Goals. Obtain a starting point for the evolution analysis. Discover the old parts of the system, discover the recently changed parts of the system. Get an overall feeling on the system's evolution by "looking back in time".

Application. In Figure 4.2 we see an *age map* of ArgoUML's history in version 0.24, a system with many of its "first generation" classes still in place.

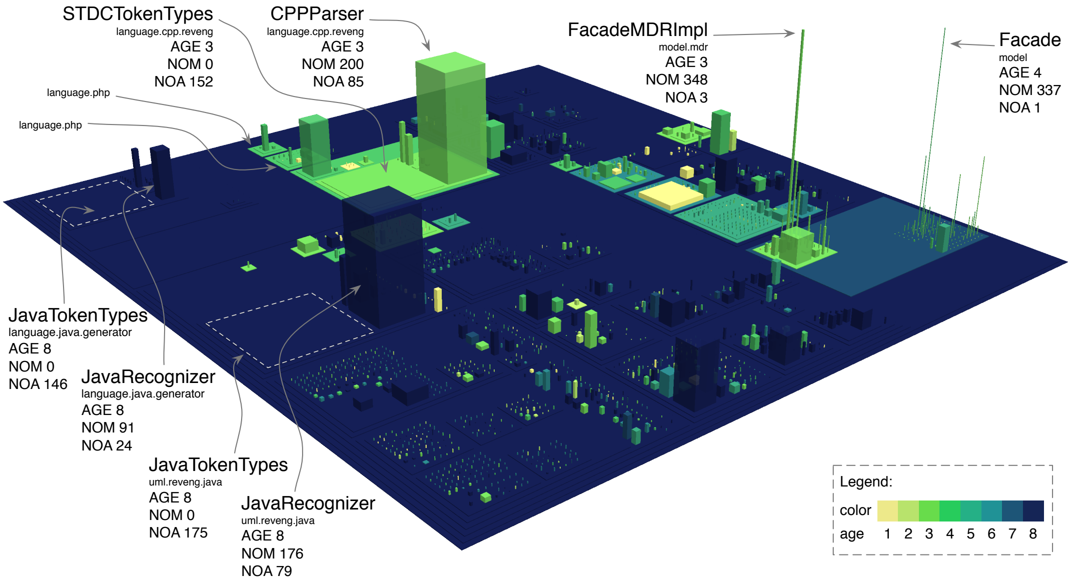


Figure 4.2. Coarse-grained *age map* of ArgoUML

Following up on our investigation from a previous analysis, presented in Section 3.4.2, we notice that the pairs of `JavaTokenTypes` and `JavaRecognizer` buildings which appear in two different districts (i.e.,`language.java` and in `uml.reveng.java`) are all as old as the city itself, which is indicated by their common dark color. With the help of this fresh information, revealed by the historical data, we can discard the hypothesis that it is a migration/replacement of one pair of classes with the other, since both pairs have been part of the system since its inception.

Another insight we obtained using the *age map* technique is that ArgoUML at the beginning has only supported the Java language. The support for C++, C#, and PHP has been added at a later stage, as shown by the light green color of the packages `language.cpp`, `language.csharp`, and `language.php`, respectively.

Drawbacks. The age map flattens the evolutionary information with respect to the currently visualized system version. What we need is a technique to visualize the process of evolution itself, i.e., we need to *travel through time*.

4.6 Coarse-Grained Time Travel

Description. *Time traveling* is achieved by stepping back and forth through the history of the system while the city updates itself to reflect the currently displayed version. Locality plays a major role: We make sure that every element history gets assigned an individual lifetime real estate in the city, i.e., if an artifact is removed from the system or it has not yet been created at a certain point in time, this is denoted by an empty space that cannot be occupied by other artifacts.

Goals. Observe the evolution both of the entire system and of individual artifacts, e.g., packages or classes. At the system level, discover which districts have been under heavy maintenance or barely touched between two consecutive versions or along their entire evolution. By focusing on a particular artifact in a city, observe its "birth", its evolution in terms of the chosen set of metrics, and in some cases its "death", i.e., its removal from the system.

Application. In Figure 4.3 we see the sequence of views obtained during our *time travel* through ArgoUML's major release history[4]. We omitted showing the version 0.24, for it was irrelevant for this discussion, because there were almost no observable changes between the last two samples. To provide a better sense of time, we marked the release numbers and dates on the figure. One question left unanswered during our program comprehension case study presented in Section 3.4.2, was the intriguing case of the `Facade` interface, whose over 300 declared methods were implemented by one class only. Stepping through time reveals the origin of this apparently questionable design decision: In version 0.14, a large building emerges, representing class `ModelFacade` with 60 attributes, 183 methods, and 1,280 lines of code, which becomes enormous (i.e., 108 attributes, 426 methods, and 3,275 lines of code) in version 0.16. In version 0.18.1 `ModelFacade` disappears, but its disappearing coincides with the appearance of two other tall and thin buildings: the interface `Facade` (1 attribute and 306 declared methods) and the concrete class `NSUMLModelFacade` (2 attributes, 316 methods, and 2,102 lines of code) implementing `Facade`.

[4] A movie of this *time travel* is located at `http://www.inf.usi.ch/phd/wettel/codecity-movies.html#ArgoCTT`

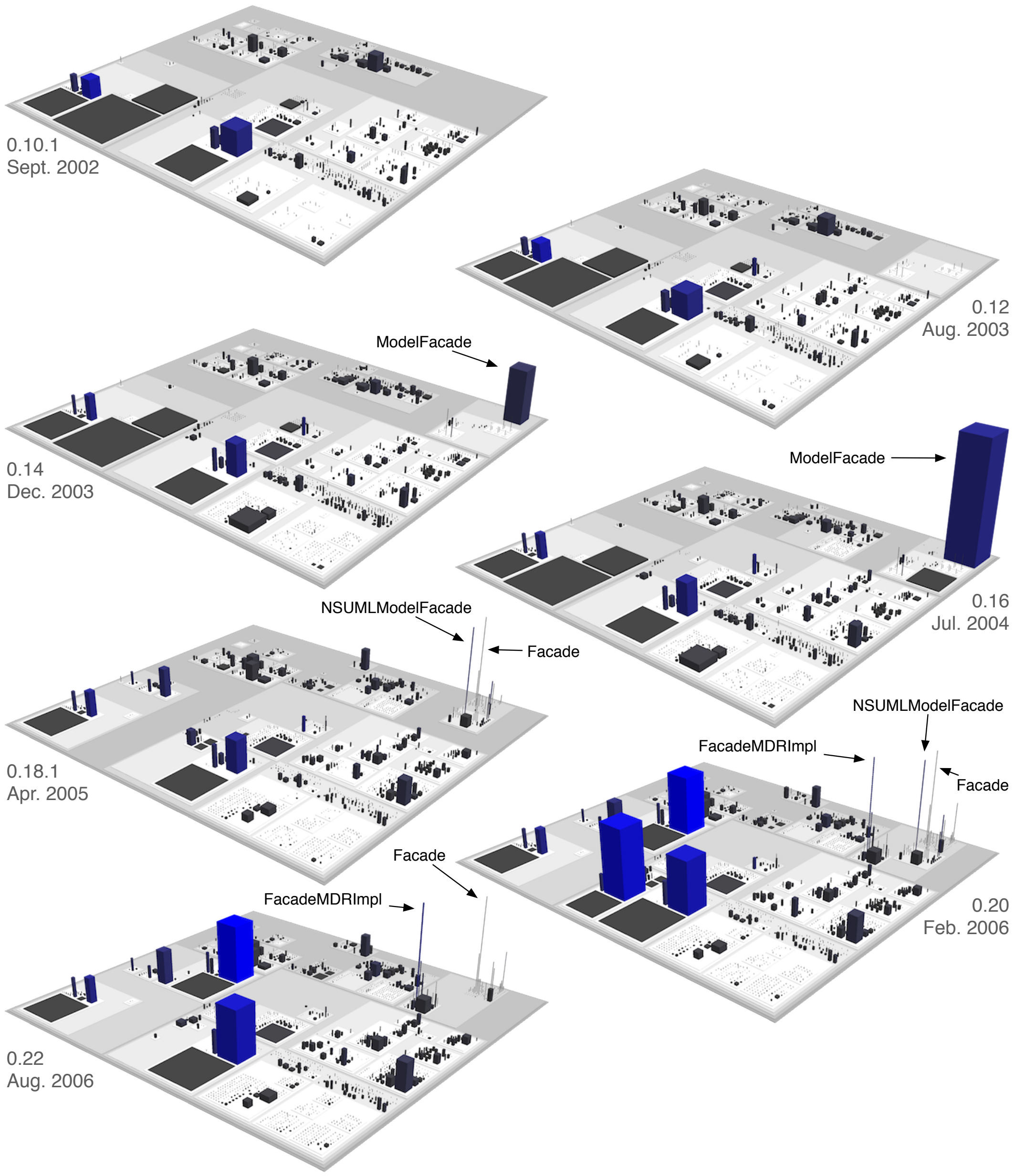


Figure 4.3. Coarse-grained *time travel* through the history of ArgoUML

This large-scale city transformation reflects a massive refactoring, caused either by the fact that `ModelFacade` was growing into a maintainer's nightmare, or by the developers' need to define variations of its behavior. They declared the common functionality (i.e., 306 methods) in an interface and moved the concrete behavior from `ModelFacade` to the new class

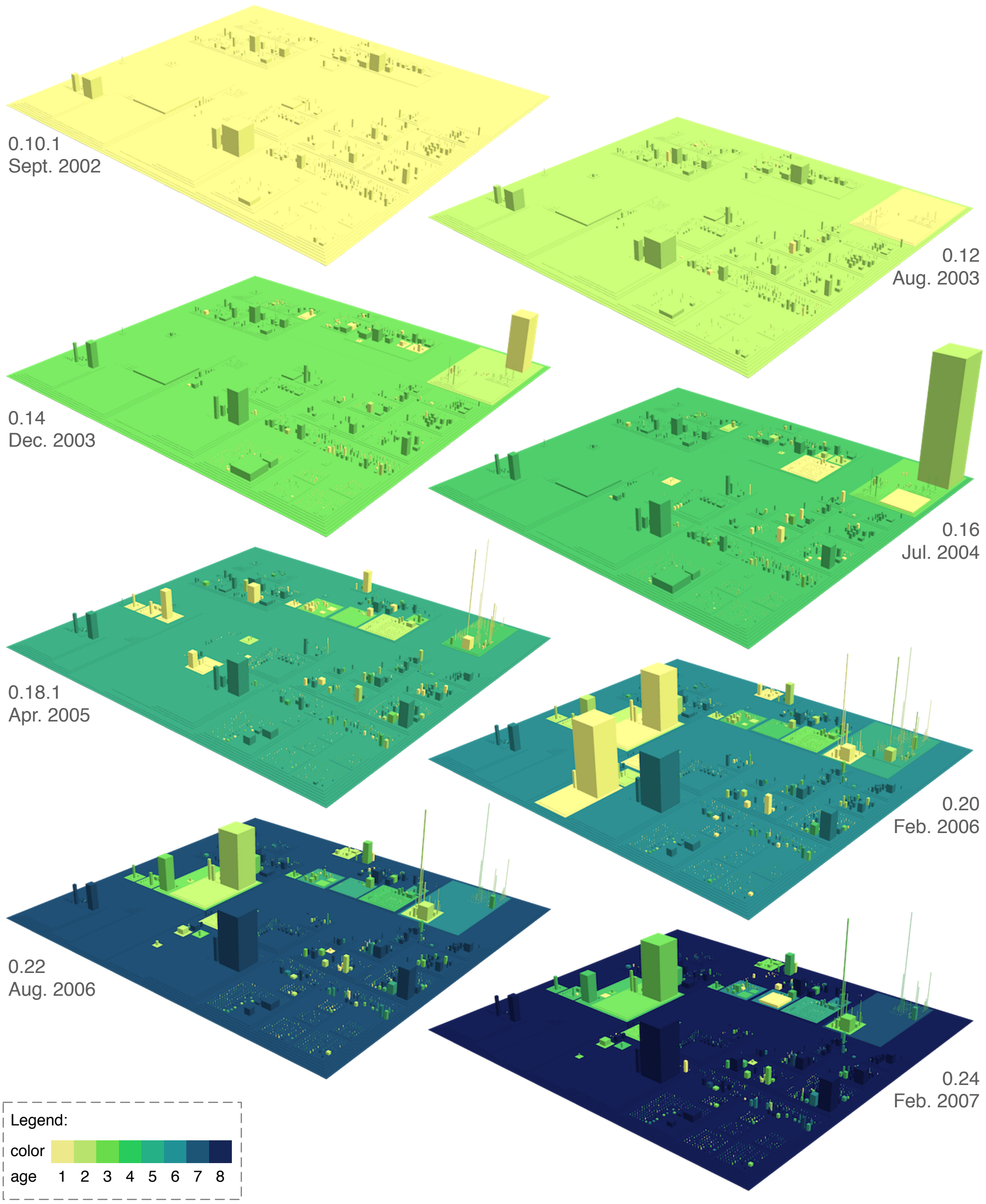


Figure 4.4. Applying both *time travel* and *age map* to ArgoUML

NSUMLModelFacade, the first implementor of Facade. Version 0.20 gives birth to a second implementor of Facade, called FacadeMDRImpl, with 2 attributes, 329 methods, and 2,709 lines of code. Version 0.20 is the only version in our sampled history in which the two implementors coexist, as in version 0.22 the class `NSUMLModelFacade` is removed, leaving `FacadeMDRImpl` the only implementation of `Facade`.

Reality Check. A key developer of ArgoUML confirmed our insights gained during the *time travel* and provided more details: "`ModelFacade` was an implementation of the `model` subsystem using NSUML repository. When the change was made to MDR we turned this to a regular interface allowing for several different repositories. The attributes in the `ModelFacade` are not attributes but constant tokens used in the NSUML repository implementation. Because this was for JDK1.3, and JDK1.4 enums were not available so the number exploded. When we refactored the `ModelFacade` I think these constants were partly not needed and partly replaced by methods to retrieve the constants from the chosen implementation."

Drawbacks. At this granularity level, the technique does not depict the internal evolution of classes, which is the level at which the changes happen. For example, if a developer removes and adds the same number of methods between two consecutive versions, the NOM metric value remains the same, and so does the building's height, in spite of the substantial changes. This loss of detail is evident in Figure 4.4, which shows a combination of *age map* and *time travel* applied to the history of ArgoUML[5]. Between versions 0.22 and 0.24 not much seems to have happened in the system, in spite of the amount of changes typically implied by a new major release.

4.7 Fine-Grained Age Map & Time Travel

Description. The same techniques described previously, applied at fine granularity.

Goals. Obtain insights into the method-level evolution. Discover classes created in one "bang" versus classes grown in an incremental manner.

Application. Our first history of JHotDraw was obtained by sampling the evolution of the system based on a sampling period of 24 weeks, which resulted in eight versions, described earlier in Table 4.3. By applying the fine-grained *age map* technique on this history we obtained the visualization in Figure 4.5, which provides a look back in the past from the most recent version of the system. This first view leads to several interesting facts about its evolution, discussed next.

The districts colored in dark blue represent the most rooted packages of the system, such as `CH.ifa.draw`[6]`.standard`, `framework`, and `figures`, because they have been there since the system's inception. The city's tallest buildings are the classes `application.DrawApplication` and `standard.StandardDrawingView`, which not only have been part of the system starting with its first version (their bases have the same color as the city's ground), but they have permanently required adaptation during the system's evolution (they are painted in a wide range of colors).

The light green color of district `test` shows that the package is relatively new. Indeed, the `test` package first appears in the sixth version of our sampled history. The small buildings colored in light yellow, represent classes called `test.AllTests`, which have been added in the seventh version. Each of these classes contains two methods: `main` and `suite`. Each `main` method is the starting point for running the `suite` of tests located in the sub-packages of `test`.

[5] A movie of this *time travel* is located at `http://www.inf.usi.ch/phd/wettel/codecity-movies.html#ArgoCTTAM`

[6] From here on, we omit the common prefix `CH.ifa.draw` from package names and qualified class names in JHotDraw.

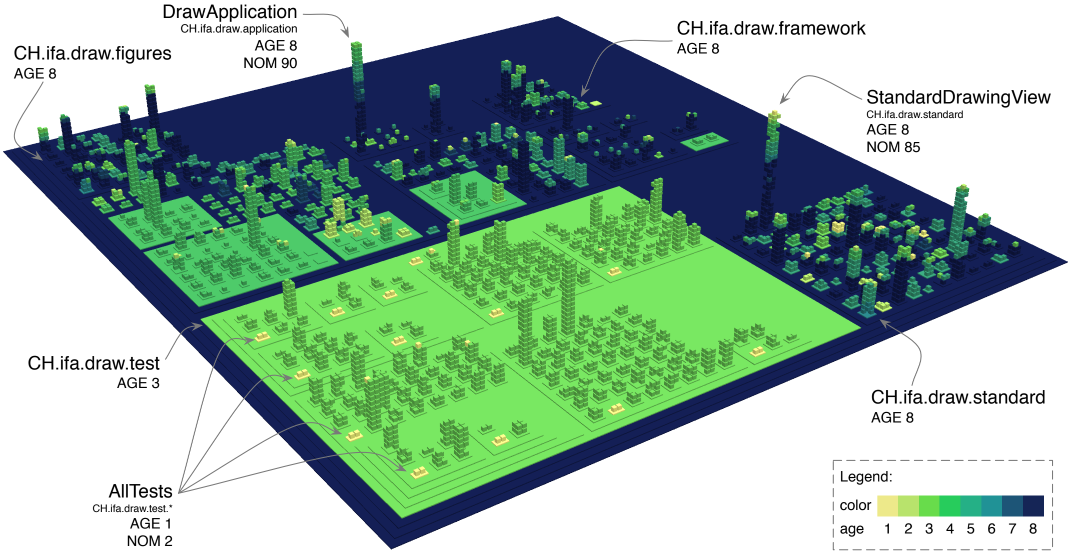


Figure 4.5. Fine-grained *age map* applied to the most recent version of JHotDraw

To learn more about the `test` package, we employ again a combination of the *time travel* technique and *age map*, presented in Figure 4.6, which confirmed what the *age map* depicted: the entire package appeared all at once towards the end of JHotDraw's history[7]. Since our sampled history of JHotDraw contained only eight versions covering $3^1/_2$ years, we did not want to jump to conclusions: The sudden appearance of all the unit tests could have happened gradually over a period of 6 months, between January and July 2003. To reason more accurately about this evolutionary fact, we sampled the system using a weekly sampling period but, to our surprise, this did not change our conclusion. We were able to reduce the period in which the entire `test` package was created to one week, spanning from revision 121 (24/01/2003) to revision 155 (31/01/2003), during which 34 commits were performed.

Writing all tests at once and at a late stage in a project is awkward. We thought of two hypotheses that could explain the sudden appearance of the entire `test` package at a late stage in the lifetime of the system. It was possible that, although the tests were written gradually, they were added to the versioning repository only very late. The second hypothesis was that, since JHotDraw is a Java port of the well-designed HotDraw system written in Smalltalk [Joh92], the tests were only later ported to the project.

Reality Check. The main developer of JHotDraw shed light on this matter: "Regarding your question of the new `test` package: I used a JavaDoc-based code generator to automatically generate test cases for the JHotDraw. Therefore, the test methods are still not implemented and only the work to do is outlined. Because it is an incomplete package and only created as an example for the test generator I think it it best to ignore this package." One lesson we learned from this case study is that, although we perform most of the analyses at a higher abstraction level, a look at the source code is every now and then required.

[7] A movie of this *time travel* is located at `http://www.inf.usi.ch/phd/wettel/codecity-movies.html#JHotDrawFTTAM`

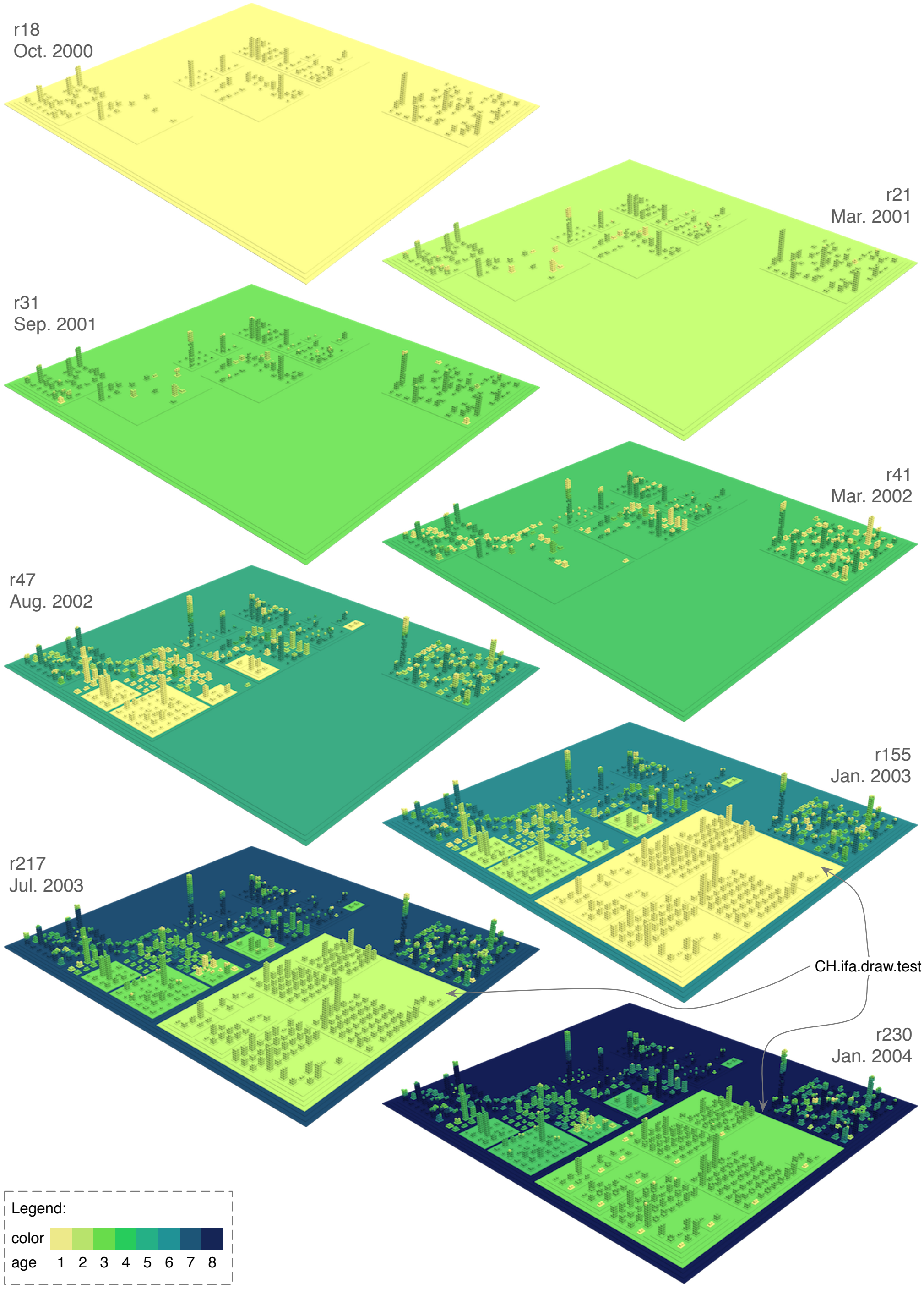


Figure 4.6. Combining fine-grained *time travel* with *age map* on JHotDraw

Drawbacks. One problem with the techniques that display only one version of the system at a time is that they do not show, without traveling back and forth, *when* a particular method disappeared, since its representation is an empty space. Moreover, they are prone to scalability issues, due to the large number of graphical elements employed to visualize a system (e.g., ArgoUML is represented by over 16'000 elements). In addition, if one visualizes complete systems, the methods of a class are barely visible. To address these issues, we devised the *timeline* technique.

4.8 Fine-Grained Timeline

Description. The class versions are represented as platforms next to each other along a timeline, from left (first version) to right (last version) and the methods are represented as "bricks". We combine this with the *age map* technique to enable a clearer visual distinction between the different "generations" (i.e., groups of methods created in the same version) of methods.

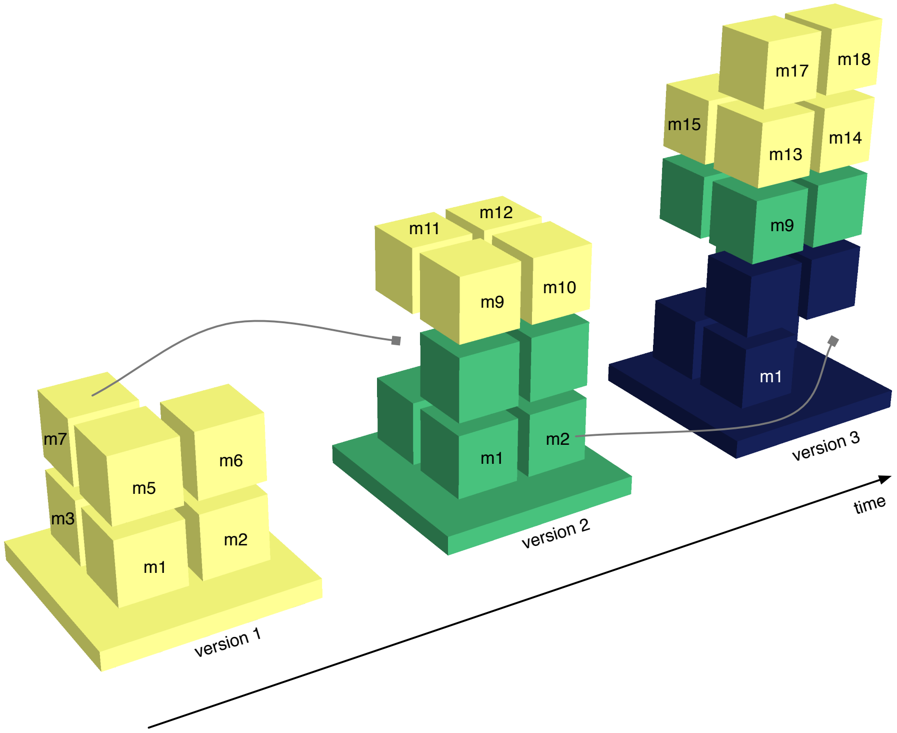
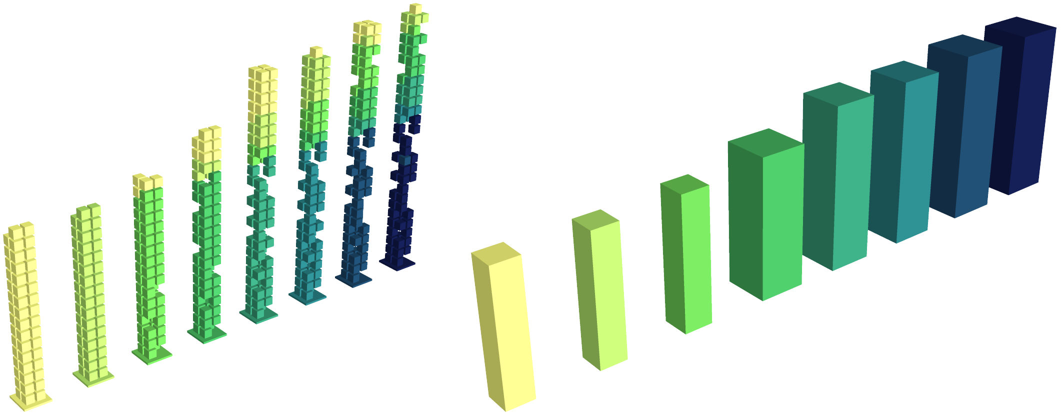


Figure 4.7. Example illustrating the principles of the *timeline* technique

Figure 4.7 illustrates these principles applied to the three-versions history of class `C`. In the first version, class `C` has seven methods (`m1` to `m7`). In the second version, method `m7` is removed and five other methods (`m8` to `m12`) are added to the class. The new bricks appear at the top of the building in a lighter color than the rest of the bricks. The place formerly occupied by `m7` will remain empty from here on. In the third version, the older method `m2` is removed and six new methods (`m13` to `m18`) are added. The benefit of this visualization is that it provides a complete representation of an artifact's evolution, thus allowing for the detection of evolutionary patterns.

We use this technique mostly at the class level, by visualizing the evolution of a class in terms of its methods. The importance of choosing the right granularity is shown in Figure 4.8 by means of a comparison: The fine-grained timeline of class `standard.StandardDrawingView` from JHotDraw, depicts interesting events, such as the creation or removal of methods (See Figure 4.8(a)). In contrast, the coarse-grained timeline of the same class is only able to show details at the class level, such as the modification of the NOA or NOM metric values (See Figure 4.8(b)). We consider the coarse-grained timeline a poor combination for evolution analysis and, therefore, exclude it from the discussion.



(a) Fine-grained

(b) Coarse-grained

Figure 4.8. The timeline of class `standard.StandardDrawingView` at different granularities

This technique can also be applied at package level, to depict the evolution of a package in terms of its sub-packages and classes. Figure 4.9 shows a fine-grained timeline of a district, whose tallest building represents precisely the class whose timeline was shown in Figure 4.8(a).

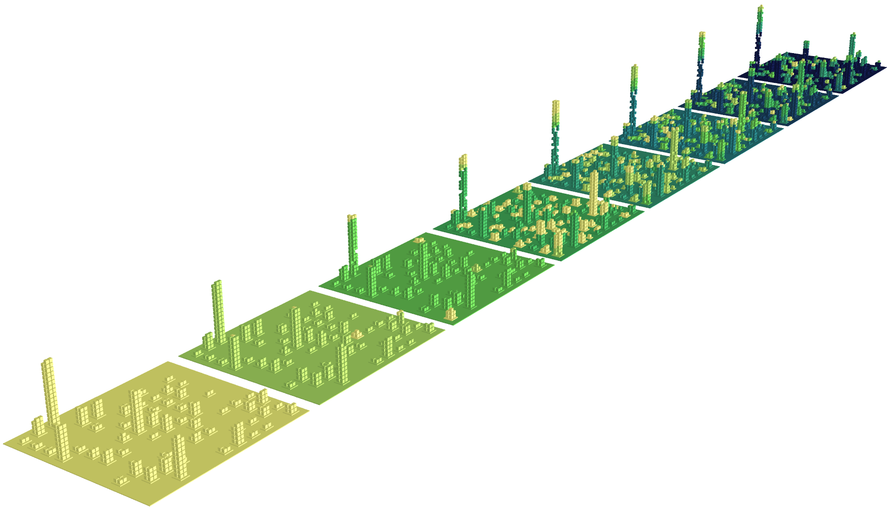


Figure 4.9. The timeline of package `standard` in JHotDraw

Goals. Isolate a reduced set of artifacts to create a view of their entire history including all the inner components. Observe evolution patterns, such as incrementally grown classes, recurring methods, etc.

Application. We analyzed a number of Jmol's classes using the *timeline* technique. Figure 4.10 shows the timeline of the `org.jmol`[8]`.g3d.Graphics3D` class, which spreads over the last 22 versions of Jmol's history made of 51 versions. This class is likely to be an important one, given the structure of its timeline. Already in its first version, it contained 103 methods and it has been subject to massive changes. With each new version, new functionality (i.e., methods) was added to this class and at the same time old functionality was removed. Gradually, the initial structure of the class got lost in time and the final column of the timeline accurately reflects its continuous adaptation: Out of the 311 methods that existed throughout its history, only 158 have made it to the current version.

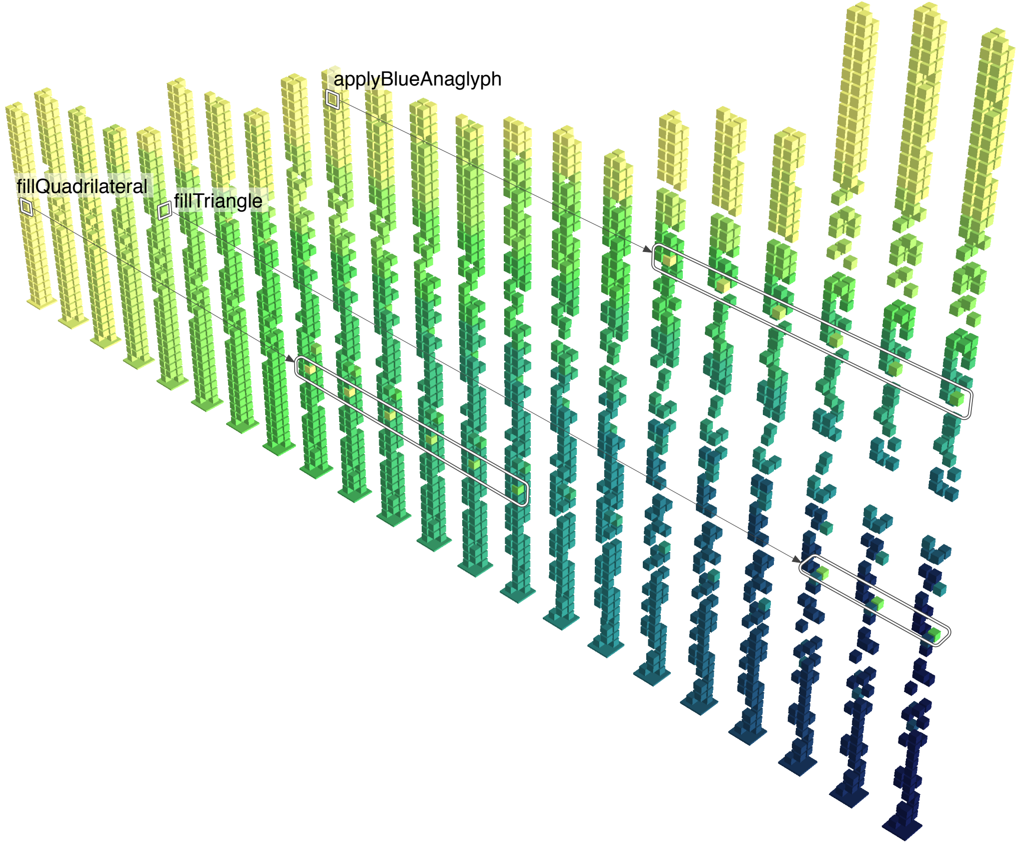


Figure 4.10. Timeline of class `Graphics3D`

The timeline of `Graphics3D` illustrates an interesting pattern, manifested through bricks exhibiting color anomalies. After looking into this pattern, we learned that this pattern indicates a restoration of groups of methods after a period of time from their removal.

[8] From here on, we omit the common prefix `org.jmol` from package names and qualified class names in Jmol.

Through the combined effect of the *age map* technique and the chronological order imposed on the layout, the color of the brick representing a restored method stands out as an anomaly in every subsequent version of the class, i.e., it breaks the color pattern of its surrounding bricks. Although its position denotes the fact that it has been created at roughly the same time as its neighbor bricks, the color reflects a shorter life (i.e., fewer versions) than the one of its neighbors.

Figure 4.10 shows three examples of such methods. The first one is `fillQuadrilateral`, which disappeared after the first version of the class, reappeared in the ninth version and after six more versions was removed again. The second method is `fillTriangle` which was created in the first version of the class, survived for five versions, and then disappeared for fifteen more versions. As this case illustrates, the more versions pass between removal and restoration, the more striking the color anomaly is. The third method is `applyBlueAnaglyph`, which was created later (i.e., in the tenth version), removed right after that, and restored in version 17.

We looked at the timelines of several long-lived classes in Jmol, four of which are illustrated in Figure 4.11 along with a description of their evolution in numbers (See Table 4.5). At a first glance, we see how each *timeline* reflects the evolutionary characteristics of the underlying class history. For instance, the peak of each *timeline* (the height of the last version of the building) depicts the number of method histories: `Eval` is twice as tall as `JmolViewer` or `TransformManager`, but half the height of `Viewer`, which encapsulates 1,029 method histories in its evolution. `Eval` lost many of its initial methods during its evolution (166 out of 432) and this is well reflected in its *timeline*: the most recent version (i.e., the last column) looks unstable, with many missing bricks in its structure.

Class History	**Total**	**Current**	**Removed**
`api.JmolViewer`	177	150	27
`viewer.Viewer`	1,029	750	279
`viewer.TransformerManager`	220	161	59
`viewer.Eval`	432	266	166

Table 4.5. The number of methods for the class histories in Figure 4.11

Table 4.5 presents the *timelines* of the classes `Viewer`, `JmolViewer` (i.e., the superclass of `Viewer`), `Eval`, and `TransformManager`. An intriguing observation we made was that each of these timelines shows a group of bricks, which disappears in revision 5,154 from 22/05/2006 and reappears after exactly three sampled versions, in revision 6,098 from 6/11/2006, as illustrated by the common time coordinates in the top left part of Figure 4.5.

The hypothesis that first came to our mind was that the developers massively removed methods from these logically coupled classes, thus generating bugs which were not detected right away and which could only be fixed later by reviving the removed methods.

Reality Check. Our hypothesis was confirmed both indirectly and directly. We first looked at the logs of the Jmol versioning repository. The version in which the methods were removed had to be in the range between revisions 4,709 and 5,154. The log of revision 5,091 from 10/05/2006 reads: “No more `javax.vecmath.Point3f` in g3d shape drawing routines. There were some cases where screen coordinates were being passed in as `Point3f` objects [. . .] ”. Similarly, we searched for a revision which would give away the recovery of the previously

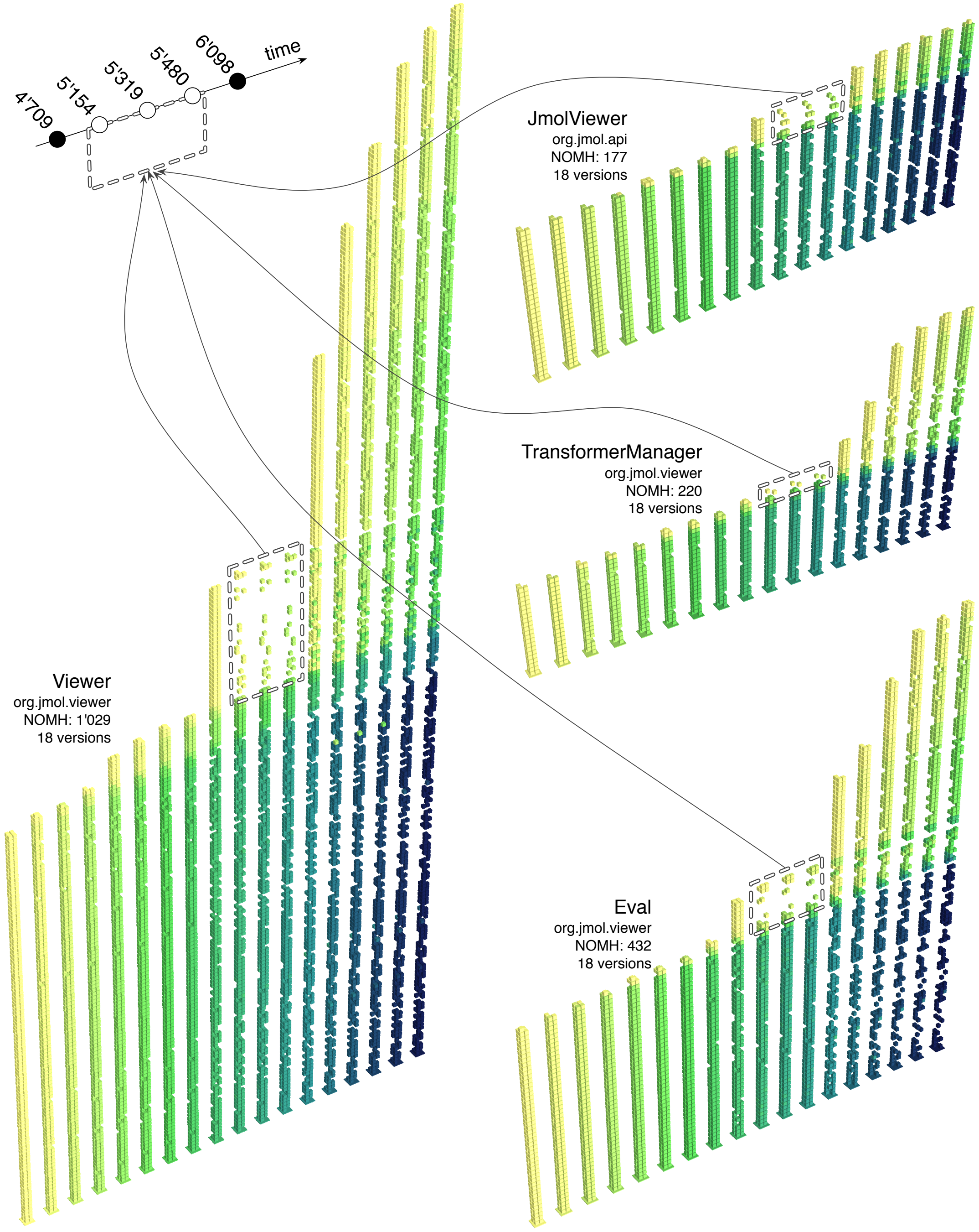


Figure 4.11. Learning from the past by correlating several class timelines of Jmol

removed methods, in the range between revisions 5,480 and 6,098. We found revision 5,579 from 17/09/2006 whose log acknowledges the restoration: “Revert of `vecmath` lib change”.

This was a fortunate case, because oftentimes the developers do not document their modifications in the logs of the versioning system. However, since we wanted to learn more about this system, we contacted the developers who committed in the repository during that period. Three of the developers shared with us interesting insights on this period in the system's evolution. *Developer1* agreed with us: "Your hypothesis is probably correct" and recalled: "We found some major problems, and diagnostic was too difficult, so we reverted to a stable version and tried to apply patches in small batch." *Developer2* remembered that "It was quite a nightmare for everyone involved. The issue was that I was new to the project and had committed quite a few additions to Jmol thinking that *Developer3* was monitoring; as it turned out he found my additions too much too fast, and because there was a problem with the graphics display module g3d that resulted in some slow performance, he opted to revert to an earlier state. In the end it turned out to be a recent addition to transparency in the graphics, not anything I had done, that caused the problem. In any case, we did sort of start over – or at least I did. I think I have that right. So I think your hypothesis is correct [. . .] ".

Drawbacks. Given the real estate of the screen, there are scalability issues in the case of artifacts with hundreds or more revisions.

4.9 Discussion

We take a step back and reflect on several issues related to the application of the city metaphor to the analysis of software system evolution.

Tracking events in time. The actual time at which a particular event happened can be anywhere between the commit times of the earliest sampled version in which the event occurs and the time of the previous sampled version. In this context, the sampling period plays a major role in establishing accurate time localization of events.

Sampling policies. The sampling policy has a major influence on the information that can be extracted from the history and intuitively, one is tempted towards using shorter sampling periods and obtaining richer histories. However, manipulating numerous versions of an industrial system raises the issue of scalability with respect to memory requirements and processing time. We believe that an incremental approach can be applied in such cases, by starting first with a sparse history, i.e., few samples distant from each other in time, and then focusing on interesting intervals by increasing the number of samples and decreasing their temporal distance. We applied this approach in the case of JHotDraw. We also experimented with two different sampling policies, namely time-based (JHotDraw and Jmol) and release-based (ArgoUML). The time-based sampling allows us to observe the evolutionary process as a "slow motion movie" with the drawback of potential duplicated samples. The release-based sampling has the advantage that the frames are steps of the actual development cycle. This eliminates duplicate samples but has the disadvantage that one needs to correlate the development steps with the actually elapsed time.

Entity identity. Since we rely on the Hismo meta-model to model system histories, the entity identity is reduced to name identity, i.e., if two entities of the same type have the same name in two different versions, they are considered to be part of the same history[Gîr05]. Therefore a simple rename refactoring performed on an entity (e.g., package, class, method) leads to the loss of its identity. The consequence of this shortcoming of the underlying meta-model is that a method renaming is represented in a fine-grained representation as a brick removal and a brick addition.

Color scheme limitations. Any linear color scheme can be expressed as a finite series of colors and the human eye's color sensitivity cannot differentiate among very close colors in terms of hue, luminosity and contrast. Therefore, the more versions there are in a system history, the smaller will be the difference between two colors representing successive ages. Although we strived for efficiency and picked very carefully the colors while designing our color scheme, our experience taught us that this color scheme works best up to a maximum of ten versions. Increasing this number hinders the visual distinction among consecutive versions.

Consistent locality. Due to the evolutionary layout which takes into account the entire history of every software element in the system, we support consistent locality, which helps in keeping the viewer oriented at any time. This enables the user to observe hot-spot neighborhoods with respect to the evolutionary phenomenon: conservative districts (i.e., which rarely change), districts permanently "under construction" or moving districts. However, the price payed for providing consistent locality is the extra space used by the layout for allocating lifetime estates to every element, including those with a short life.

The techniques. Despite the evolutionary structural overview that the *age map* and the *time travel* techniques provide with the coarse-grained representation, their drawback is the low level of detail provided for classes. To make up for this, we created a fine-grained representation, to see how method addition and removal drives the evolution of classes. However, this level of detail raises scalability problems in the case of very large systems. Moreover, the overview is lost for such large systems, i.e., the details are not visible from far away and zooming in leads to context loss. Since our views present only one version of the system at a time, with the possibility to perform *time travels*, we needed to be able to focus on a single element throughout its entire evolution. The *timeline* technique makes this possible and allows for the detection of evolutionary patterns.

4.10 Related Work

As illustrated in Chapter 2, many visualization approaches aimed at software evolution have been proposed over the last decade. Visualizing the evolution of software structure is by no means new and many researchers have proposed various approaches, which we consider only remotely related to our work.

4.10.1 Remotely Related Work

Eick et al. [ESEE92] also used color to depict the age, however at at a lower abstraction level: Each line of code is visualized as a row and the files are visualized as columns in SeeSoft, in which the color of the row depicts the age of a line of code. The age maps in our approach depict information at a higher abstraction level and from an object-oriented point of view (i.e., the age of packages, classes, and methods, rather than that of the lines of code in a file) and puts all of these in the structural context of the system. Gall et al. [GJR99] used 3dSoftVis to visualize evolution, by means of a compacted 3D visualization that shows 2D tree graphs aligned in time and a compact 2D visualization obtained by projecting 3D diagrams onto 2D space.

Our timeline representation is partially inspired from the Evolution Matrix of Lanza et al. [Lan01], which shows the evolution of classes, represented as rectangles, in terms of a set of metrics mapped on the dimensions of the rectangles. Girba et al. [GLD05] raised the granularity level and looked into the evolution of class hierarchies using a 2D visualization which correlates the histories of classes and inheritance relationships.

Wu et al. [WHH04] proposed a visualization technique called evolution spectrographs, which portrays the evolution of a spectrum of components based on a particular property measurement which reduces every version of a file to a just number. Similarly to our age map technique, Wu et al. made use of color to depict the recency of the last change.

Pinzger et al. [PGFL05] work at a higher granularity and visualize various evolutionary aspects of complex software systems using Kiviat diagrams to depict multiple evolution metrics, which provide static visualizations of a large number of metrics for the entire evolution of modules. However, this work lacks both the system overview and a fine-grained level of detail.

In spite of all this work targeted at visualizing software evolution, the contributions in the context of the city metaphor applied to evolution are rather scarce.

4.10.2 Closely Related Work

One of the many visualizations of VizzAnalyzer mentioned by Panas in its work on generic software visualization [PLL05] is program evolution visualization, exemplified with a visualization similar to a timeline. However, their evolution visualization is not based on the city metaphor.

Xie et al. extended the SourceViewer3D approach to enable the visualization of CVS repository information [XPM06]. However, the approach is very low-level (i.e., it computes every metric at the level of the line of code) and follows the structure of the CVS repository, which is organized in terms of files. Our work is aimed at object-oriented systems and its focus is on higher-level entities, e.g., packages, classes. The approach of Xie et al. is not a visualization of the process of evolution, such as our time travel technique, but a metric-based static visualization, which employs evolution-related metrics instead of the typical structural metrics. From this point of view this work is more related to the work of Pinzger et al., who visualized multiple evolution metrics [PGFL05].

Langelier et al. proposed an application of their approach to analyze the evolution of software quality using animated visualization [LSP08]. To characterize software quality, the authors employ a set of structural software metrics and version control information. The similarity to our approach is the technique they propose, which apart from the use of animation, is very similar to our time traveling technique. The authors enable the user to go back and forth in time and to observe the visualization changing gradually, according to the changes in the system. Moreover, similarly to our consistent locality, the authors use what they call static position, which employs maintaining the same position for each element in the visualization. One of the differences between our approaches is that we allow observing the evolution at different levels of granularity, i.e., a coarse-grained, similar to their representation, and a fine-grained, which does not have a correspondent in their approach and enables us to track finer-grained changes. Another difference of our approach is that, apart from the time travel, we devised two other complementary techniques, i.e., age map and timeline, whose usefulness we demonstrated by means of several case studies.

A somewhat related contemporary work, although not very related to ours, is the approach of Kuhn et al. called Software Cartography [KLN08], which visualizes software evolution in a similar manner to our time travels. We showed a picture of this approach earlier, in Figure 2.11(h), in the context of the history of software visualization. The authors employ techniques from cartography, such as hill-shading and contour lines, to represent the properties of the software entities. However, the focus of the visualization in their case is not the system's structure as in our approach, but the vocabulary extracted from the source code of software systems.

4.11 Summary

To further demonstrate the versatility of the city metaphor defined in Chapter 3, we applied it in a second context, namely software evolution. We described a number of visualizations obtained by combining one of three techniques we devised (i.e., age map, time travel, and timeline) with one of the two granularity levels of our metaphor (i.e., coarse and fine). The fact that we were able to build new visualizations aimed at a different aspect of software systems on top of the same metaphor is an indication of the versatility of the metaphor, which supports our thesis.

We validated the described set of visualizations by applying them to the histories of three open-source software systems, including one that we already had experience with. By using our visualizations to analyze these evolutions, we were able to find explanations for several open questions from a previous case study. Furthermore, we were able to learn interesting facts from the lifetime of two systems we had not previously analyzed. Overall, the facts we learned using this application context could have not been revealed by any of the versions of these systems in isolation, but only in the historical context.

Moreover, the insights we acquired during our analyses were not only interesting, but also accurate: Some of the main developers of the systems we analyzed confirmed the correctness of our findings. The results we obtained encourage us to carry on in Chapter 5 with the third application, i.e., the design quality assessment.

Chapter 5

Visual Assessment of Design Quality

5.1 Introduction

The third application of our city metaphor for software visualization, after program comprehension and software evolution, is the quality assessment of software systems.

Designing complex software systems is a difficult task, a process which takes a long time to learn and a skill that must be perfected constantly. Over the past two decades a number of design guidelines and recipes have been formulated, usually in the form of patterns [GHJV95] or heuristics [Rie96].

Nonetheless, due to external factors, namely a changing environment which triggers new requirements on a system, even the best design degrades over time, leading to a phenomenon aptly termed as "architectural drift" [Pin05], "design erosion" [vGB02], or "code decay" [EGK$^+$01]. At a fine-grained level such a decline in quality appears in the form of "bad smells" [FBB$^+$99].

One approach to assessing the quality of software design is based on Marinescu's detection strategies [Mar04a, LM06], i.e., metric-based logical functions able to find violations of design guidelines in source code. The approach defines the concept of "design disharmony" by translating a set of design guidelines into detection strategies, with which violations against the design disharmonies can be discovered.

We present a visualization technique which focuses on the software artifacts affected by design disharmonies, based on the city metaphor presented in Chapter 3. Using an approach inspired by geographical information systems, we enrich the described visualizations with results returned by a number of design anomaly detection strategies. The resulting visualizations, called *disharmony maps*, focus on the design flaws [Mar04a], while maintaining the system's structural context. The main advantage of disharmony maps is that they provide an overview of the system's design and allow the viewer to mentally map the disharmony-affected entities to locations within the city.

We apply our approach on several open-source medium to large Java system, both at a coarse granularity, which targets classes-level disharmonies, and at the fine granularity, which targets method-level disharmonies.

5.2 Design Harmony

One aspect of particular interest when analyzing a software system is the quality of its design, which influences both its comprehensibility and the required amount of maintenance over its lifetime. One approach to assessing design is centered around the concept of design harmony and its opposite, design disharmony.

5.2.1 An Overview of Design Disharmonies

Design disharmonies are formalized design shortcomings to denote pieces of a system that exhibit design problems [LM06]. Informal design rules and guidelines [Rie96, FBB^{+}99] are transformed into detection strategies [Mar04a] which are metrics-based logical conditions that detect violations against design guidelines. The antonym of design disharmony is design harmony: a software artifact is found to be harmonious when it is implemented in an "appropriate" way. This "appropriateness" is composed of three distinct harmonies that concern every software artifact:

1. *Identity harmony*, which translates to the question "How do I define myself?". Every entity in a software system must justify its existence: does it implement a specific concept and how does it do that? Is it doing too many things or nothing at all? In the context of this dissertation we focus on the following identity disharmonies:

   God Class is a class that performs too much on its own and does not collaborate much with other classes, but uses data from other classes.

   Brain Class is a class that accumulates an excessive amount of intelligence, usually in the form of several *Brain Methods*.

   Data Class is a "dumb" data holder class without complex functionality and on which other classes rely on.

   Brain Method is a method that tends to centralize the functionality of a class.

   Feature Envy refers to methods that seem more interested in the data of other classes than in their own data.

2. *Collaboration harmony*, which translates to the question "How do I interact with others?". Every entity collaborates with others to fulfill its tasks. Does it do that all on its own, or does it use other entities? How does it use them? Does it use too many? We focus on the following collaboration disharmonies:

   Intensive Coupling refers to a method that is tied to many other operations located in only a few classes within the system.

   Dispersed Coupling is complementary to the *Intensive Coupling* and it refers to a method which is tied to many operations dispersed among many classes throughout the system.

   Shotgun Surgery refers to the fact that a change in a method implies many changes of different methods and classes [FBB^{+}99].

3. *Classification harmony*, which translates to the question "How do I define myself with respect to my ancestors and descendants?" This harmony combines the two other harmonies in the context of inheritance. For example, does a subclass use all the inherited services, or does it ignore some of them? Since we do not focus on this type of disharmonies, we omit the presentation of these disharmonies, and refer the interested reader to [LM06].

5.2.2 Example of Detection Strategy: The *God Class* Disharmony

The *God Class* design flaw, first described by Riel [Rie96], refers to classes that tend to incorporate an overly large amount of intelligence and whose characteristics are described by the following rules:

1. They heavily access data of simpler classes, either directly or using accessor methods;
2. They are large and complex;
3. They have a lot of non-communicative behavior, i.e., there is a low cohesion between the methods belonging to that class.

These informal rules can be transformed into the detection strategy depicted in Figure 5.1.

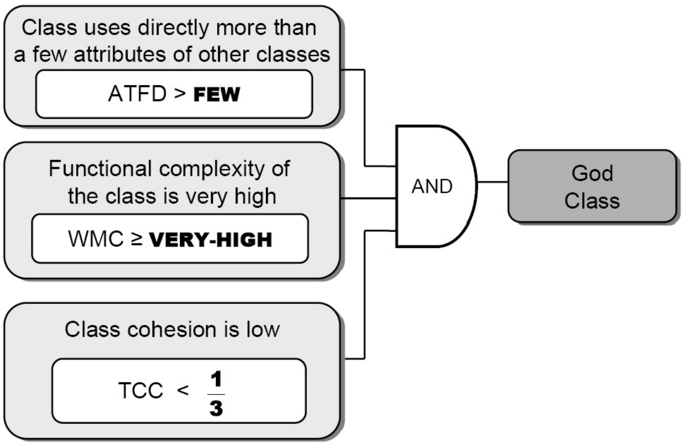

Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY-HIGH
Class cohesion is low
TCC < 1/3
AND
God Class

Figure 5.1. The *God Class* detection strategy [LM06]

The filtering conditions are expressed in terms of the following metrics (the left part of the expressions) and related to thresholds (the right part of the expressions):

- *Access To Foreign Data* (ATFD) represents the number of external classes whose any subset of attributes are accessed by the given class.
- *Weighted Method Count* (WMC) is the sum of the statistical complexity in a class [CK94], using McCabe's cyclomatic complexity metric [McC76].
- *Tight Class Cohesion* (TCC) is the relative number of methods connected via attribute accesses [BK95, BDW98].

The thresholds included by the logical conditions have been statistically determined by Lanza and Marinescu using a large number of software systems [LM06]. We already encountered these thresholds in the context of the metaphor, when we used them to compute the boundaries required by the threshold-based property mapping strategy, described in Section 3.3.2.

As illustrated in Figure 5.1, the result of applying the *God Class* detection strategy on a class is a boolean value, which indicates whether the class is affected or not by the design problem associated with the detection strategy. In fact, all the detection strategies can be considered as tests for a particular design "affection" and their result is either *positive* or *negative*.

5.3 Design Disharmony Maps

To integrate the results of running detection strategies with our city metaphor, we drew inspiration from a particular type of theme map, called *disease map*. In a disease map, the regions of a world map are colored according to the diseases that affected the regions at some point in time. Such a disease map allows one to quickly assess which are the dominating diseases in the world for a particular period of time, and also how they are distributed around the globe.

On the one hand, the metaphor fits the problem well, because with this approach we target design problems, which can be seen as software “diseases”. On the other hand, it does not clash with the city metaphor, because they are strongly linked by the geographical context.

Similarly to the disease maps, we assign a vivid colors to each disharmony. We color affected elements of the city with the color corresponding to the design problems they suffer from and unaffected elements with a neutral gray. This technique enables us to focus on the design problems in a non-distracting global context.

We integrate this technique with our city metaphor, which provides the concept of locality to the software elements and suits well the geographical context. Since the only resource used for representing the design problems is the color, we are still able to display structural metrics on the dimensions of the buildings. The resulting visualization, which we call *disharmony map*, provides an overview of the problems affecting a software system in terms of proportion, distribution and dominant types.

By combining the results of design problem detection with our visual city metaphor, we obtain the big picture of the system’s design problems, which can hardly be imagined using a non-visual, text-based approach. To illustrate this aspect, we further present the same data of detection strategy results, using both a textual representation and our visualization.

5.3.1 Design Problem Presentation

Running the *God Class* detection strategy on the `java` namespace of JDK 1.5 returns a list of 81 affected classes out of the system’s almost 1’700 classes.

Figure 5.2 shows this data in MooseBrowser, an exploration tool from the Moose tool suite [NDG05]. In this view, the second rightmost panel shows all the classes and serves as the context, while the rightmost panel contains only the *God Classes* and serves as the focus.

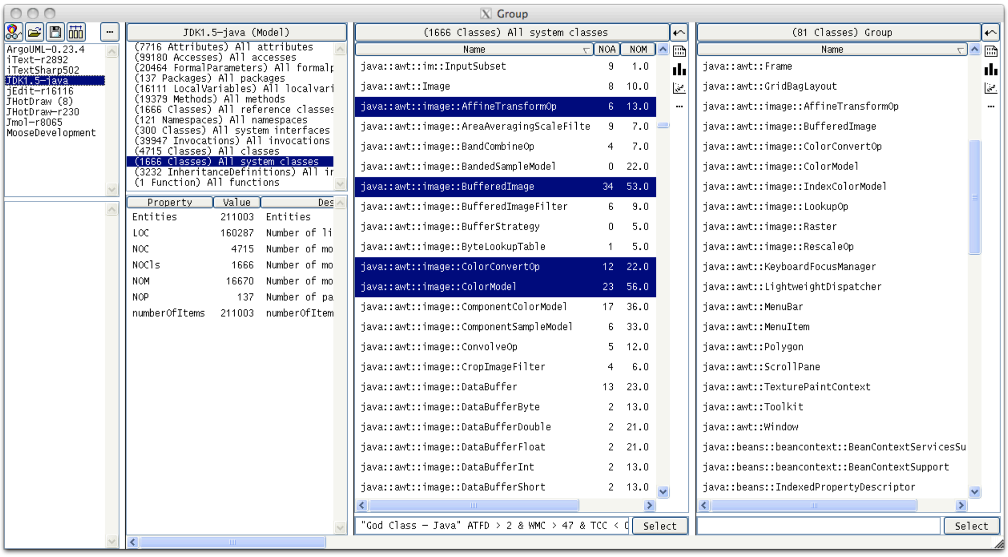
Although it includes all the data we display in a *God Class* disharmony map, the text-based presentation has several shortcomings:

1. It lacks the overview, since it is impossible to look at the results as a whole, due to the limited screen real-estate, and scrolling through the list leads to context loss.

2. To localize the *God Classes*, one has to process the list by clustering it based on the packages in which the classes are defined and then sort the clusters based on the number of occurrences.

3. Finally, it is completely unfeasible to correlate several disharmony types, even in the case of more effective textual representations, such as trees.

To enable a direct comparison, we present a disharmony map of the *God Classes* in JDK from both an aerial perspective, in Figure 5.3(a), and a top perspective, in Figure 5.3(b). While the dimensions of the buildings depict the same metrics (i.e., NOA and NOM) as in the code city



ArgoUML-0.23.4
iText-r2892
iTextSharp502
JDK1.5-java
jEdit-r16116
JHotDraw (8)
JHotDraw-r230
Jmol-r8065
MooseDevelopment

JDK1.5-java (Model)

(7716 Attributes) All attributes
(99180 Accesses) All accesses
(20464 FormalParameters) All formalp
(137 Packages) All packages
(16111 LocalVariables) All localvari
(19379 Methods) All methods
(1666 Classes) All reference classes
(121 Namespaces) All namespaces
(300 Classes) All system interfaces
(39947 Invocations) All invocations
(4715 Classes) All classes
(1666 Classes) All system classes
(3232 InheritanceDefinitions) All ir
(1 Function) All functions

Property	Value	Des
Entities	211003	Entities
LOC	160287	Number of li
NOC	4715	Number of mo
NOCls	1666	Number of mo
NOM	16670	Number of mo
NOP	137	Number of pa
numberOfItems	211003	numberOfItem

(1666 Classes) All system classes

Name	NOA	NOM
java::awt::im::InputSubset	9	1.0
java::awt::Image	8	10.0
java::awt::image::AffineTransformOp	6	13.0
java::awt::image::AreaAveragingScaleFilte	9	7.0
java::awt::image::BandCombineOp	4	7.0
java::awt::image::BandedSampleModel	0	22.0
java::awt::image::BufferedImage	34	53.0
java::awt::image::BufferedImageFilter	6	9.0
java::awt::image::BufferStrategy	0	5.0
java::awt::image::ByteLookupTable	1	5.0
java::awt::image::ColorConvertOp	12	22.0
java::awt::image::ColorModel	23	56.0
java::awt::image::ComponentColorModel	17	36.0
java::awt::image::ComponentSampleModel	6	33.0
java::awt::image::ConvolveOp	5	12.0
java::awt::image::CropImageFilter	4	6.0
java::awt::image::DataBuffer	13	23.0
java::awt::image::DataBufferByte	2	13.0
java::awt::image::DataBufferDouble	2	21.0
java::awt::image::DataBufferFloat	2	21.0
java::awt::image::DataBufferInt	2	13.0
java::awt::image::DataBufferShort	2	13.0

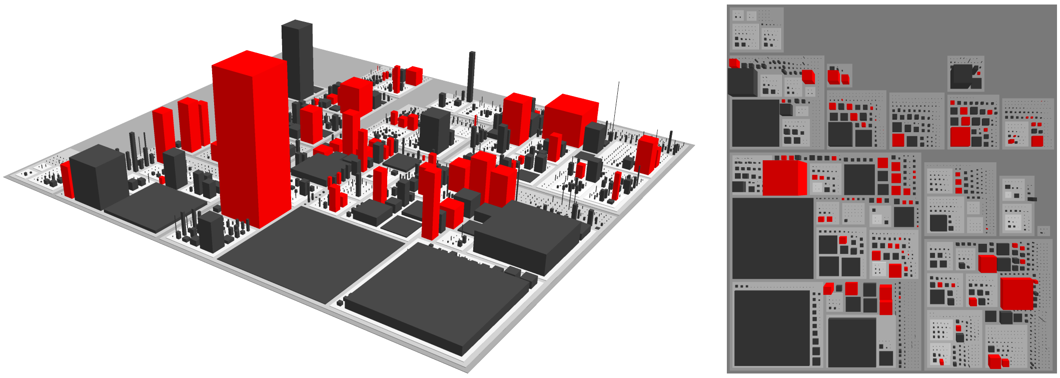
"God Class - Java" ATFD > 2 & WMC > 47 & TCC < 0

(81 Classes) Group

Name
java::awt::Frame
java::awt::GridBagLayout
java::awt::image::AffineTransformOp
java::awt::image::BufferedImage
java::awt::image::ColorConvertOp
java::awt::image::ColorModel
java::awt::image::IndexColorModel
java::awt::image::LookupOp
java::awt::image::Raster
java::awt::image::RescaleOp
java::awt::KeyboardFocusManager
java::awt::LightweightDispatcher
java::awt::MenuBar
java::awt::MenuItem
java::awt::Polygon
java::awt::ScrollPane
java::awt::TexturePaintContext
java::awt::Toolkit
java::awt::Window
java::beans::beancontext::BeanContextServicesSu
java::beans::beancontext::BeanContextSupport
java::beans::IndexedPropertyDescriptor

Figure 5.2. The God Classes of JDK's `java` namespace in MooseBrowser

presented earlier in Figure 3.17, the colors of the buildings reveal the presence of the *God Class* design disharmony: red buildings are affected classes, while gray ones are unaffected classes.



(a) Aerial perspective

(b) Top perspective

Figure 5.3. The God Class *disharmony map* of JDK's `java` namespace in CodeCity

Compared to the textual presentation, the *disharmony map* provides a number of advantages. First, the entire map fits in one screen, which allows the viewer to perceive the system as a whole. Second, it allows the user to estimate the distribution of *God Classes* in the system, which in this case is dispersed. Finally, by assigning different colors to different design problems, it can address multiple disharmonies simultaneously.

To illustrate this and to demonstrate the insights one can acquire using the *disharmony maps*, we apply our approach on a number of open-source systems and explore several design disharmonies in correlation.

5.4 Case Study Validation

We applied our approach on four open-source Java systems: the `java` namespace of JDK (Java Development Kit), ArgoUML, Jmol, and iText[1] (i.e., a PDF library). In Table 5.1 we present the version for each system and their magnitudes in terms of lines of code, number of packages, number of classes, and number of methods (NOM).

System	Version	NOP	NOC	NOM	LOC
iText	r. 2,892	75	711	6,320	80,389
Jmol	r. 8,065	50	558	6,653	84,984
ArgoUML	v. 0.24	98	1,776	12,528	138,468
JDK's *java* namespace	v. 1.5	53	1,966	18,355	160,287

Table 5.1. Case studies for the application of the city metaphor to design quality assessment

5.4.1 Class-Level Disharmonies

We encode each disharmony in a different color: yellow for *Brain Class*, red for *God Class*, orange (i.e., the combination of yellow and red) for the classes that suffer from both *Brain & God Class*, and violet for *Data Class*. For better visibility, in the case of some buildings obstructing other buildings relevant to the discussion, we manually set their transparency (user-modifiable) to 20%. Each visualization provides an interactive legend which displays, for each of the design disharmonies, the color that encodes it and the number of entities affected by it.

JDK's `java` Namespace

Before diving into details, the first impression we get by looking at the overview of JDK (See Figure 5.4) is that the system, although apparently well-organized, buries many class-level design disharmonies: we see violet districts, where mostly *Data Classes* are localized and districts of increased complexity, in which several *God Classes* and *Brain Classes* are defined.

An interesting district is `java.awt.event`, made of one wide and flat building, which represents the class `KeyEvent` and many small houses, of which many violet ones, all representing other type of events (e.g.,`InputEvent`). Although by looking at the properties of the classes one would be tempted to categorize `KeyEvent` as a *Data Class* due to its 205 attributes and only 18 methods, it actually is one of the few classes in this package which is *not* affected by the disharmony. This is due to the fact that it contains a number of non-accessor methods, some of which are fairly complex. Another interesting area of the city is the “violet” district, located next to the largest orange building in the city. This district represents the package `java.awt.geom`, which hosts 17 of the 109 *Data Classes* in JDK.

Many of the classes that are both *God Classes* and *Brain Classes* (i.e., depicted by orange buildings) are defined in the `java.awt` package, which handles the core graphics functionality in Java: `Component` (the dominating building in the city, due to the class's 88 attributes and 280 methods), `Container`, or `Font`. Moreover, some of the most commonly used core classes in JDK

[1] `http://sourceforge.net/projects/itext`

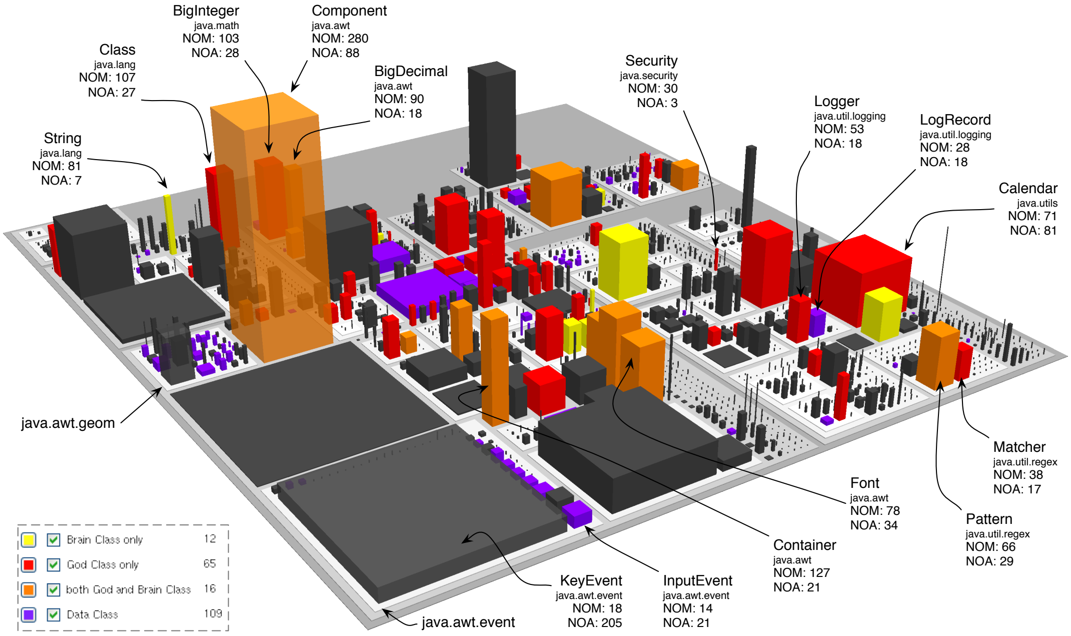


Figure 5.4. Class-level disharmonies in JDK's `java` namespace

are either *Brain Classes* (e.g., `String`), *God Classes* (e.g., `BigInteger`, `Class`, `Calendar`), or both (e.g., `BigDecimal`).

Some of the *God Classes* are easy to overlook in the absence of disharmony data. An example of such class is `java.security.Security` with its only 3 attributes and 30 methods, which encodes some complex encryption algorithms. Our approach allows us to complement the structural information of the elements with the actual disharmony data, revealing even the more subtle design disharmonies.

An interesting package is `java.util.regex` with its share of complexity in the form of *God Class* `Matcher` and *God* & *Brain Class* `Parser`, which practically accumulate the entire intelligence of the package, used for the processing of regular expressions. This is illustrated by a district containing two rather large "corporate" buildings contrasting with the small houses that surround them.

Package `java.util.logging` illustrates another pattern, a *God Class* together with the *Data Class* it misuses: `Logger` (18 attributes, 53 methods), and `LogRecord` (17 attributes, 28 methods), a *Data Class* in spite of its many methods. To verify this hypothesis, we inspected the relations of the involved classes, which revealed that more than one half (48 out of 86) of the statically-determined invocations of class `LogRecord`'s methods (most of which are getters and setters) are performed by class `Logger`.

iText

The first impression given by the overview of iText is one of a bulky system (See Figure 5.5), with a large number of outlying classes. The system seems to have a poor organization and the disharmonies are chaotically spread all over it. The dominating colors in the disharmony map reveal many problems: 8 pure *Brain Classes*, 32 pure *God Classes*, 20 classes affected by both *God Class* and *Brain Class* disharmonies, and 35 *Data Classes*.

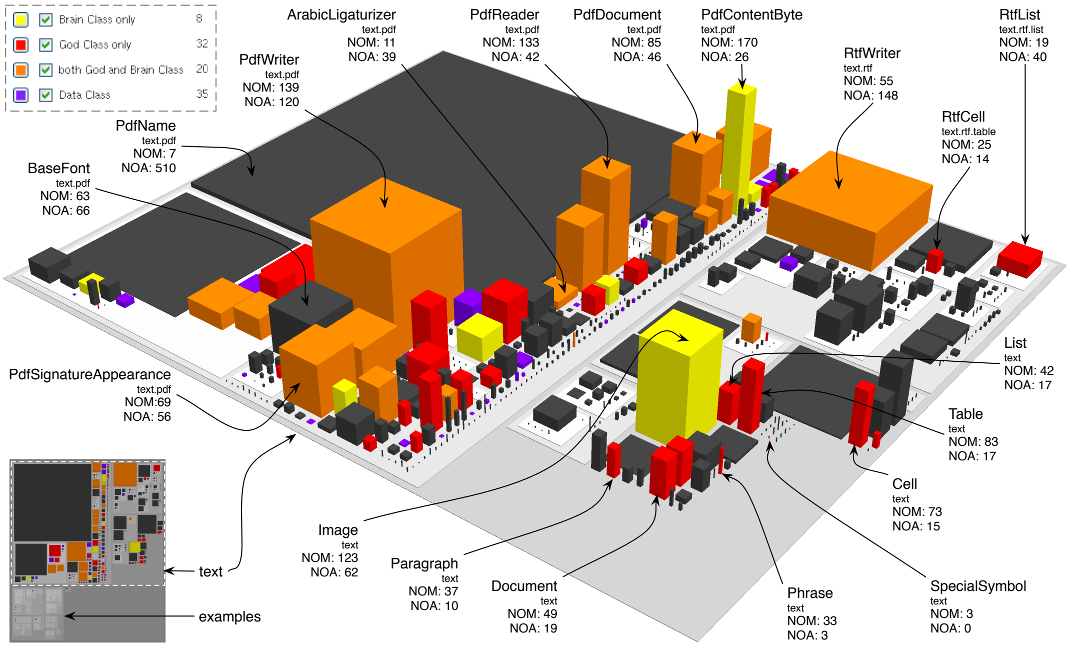

🟨 ☑ Brain Class only 8
🟥 ☑ God Class only 32
🟧 ☑ both God and Brain Class 20
🟪 ☑ Data Class 35
ArabicLigaturizer text.pdf NOM: 11 NOA: 39
PdfReader text.pdf NOM: 133 NOA: 42
PdfDocument text.pdf NOM: 85 NOA: 46
PdfContentByte text.pdf NOM: 170 NOA: 26
RtfList text.rtf.list NOM: 19 NOA: 40
PdfWriter text.pdf NOM: 139 NOA: 120
RtfWriter text.rtf NOM: 55 NOA: 148
RtfCell text.rtf.table NOM: 25 NOA: 14
PdfName text.pdf NOM: 7 NOA: 510
BaseFont text.pdf NOM: 63 NOA: 66
List text NOM: 42 NOA: 17
PdfSignatureAppearance text.pdf NOM:69 NOA: 56
Table text NOM: 83 NOA: 17
Cell text NOM: 73 NOA: 15
Image text NOM: 123 NOA: 62
Paragraph text NOM: 37 NOA: 10
Document text NOM: 49 NOA: 19
Phrase text NOM: 33 NOA: 3
SpecialSymbol text NOM: 3 NOA: 0
text
examples

Figure 5.5. Class-level disharmonies in iText

The lower-left part of Figure 5.5 shows a top perspective over the class-level disharmony map of iText, composed of two districts, i.e., the core package `com.lowagie`[2]`.text` and `examples`.

Since `examples` seems to be a small utility package, we only focus on the rest of the system, i.e., the `text` package, which is presented as an annotated detailed aerial view in Figure 5.5. The main package consists of several sub-packages, one for each file format: `text.xml`, `text.html`, `text.rtf`, and `text.pdf`.

The `text.pdf` package is vast, with 239 classes (out of which 61 affected by at least one class-level disharmony) and only 8 sub-packages, each with just a few defined classes. With that many classes defined in it, this single package has grown into a module which is difficult to understand and manage—a fact reflected by the over one quarter of disharmonious classes.

The system contains hierarchies spreading over the packages specialized on different file-formats (i.e.,`text.xml`, `text.html`, `text.rtf`, and `text.pdf`), whose base classes are defined in the main package `text`. Among these base classes there are many *God Classes* (e.g.,`Cell`, `Table`, `List`, `Phrase`, `Document`, `Paragraph`), all annotated on Figure 5.5.

In package `text.rtf`, we see some examples of "hereditary" disharmony, illustrated by the *Brain & God Class* `RtfWriter`, and by the *God Classes* `RtfCell` and `RtfList`, all disharmonious like their superclasses. The most striking harmony breakers reside in the `text.pdf` package, in which the orange color dominates, due to the large number of *Brain & God Classes*, such as `PdfWriter` (with 139 methods and 120 attributes), `PdfReader` (with 133 methods and 42 attributes), or `PdfDocument` (with 85 methods and 46 attributes).

[2] From here on, we omit the common prefix `com.lowagie` from package names and qualified class names in iText.

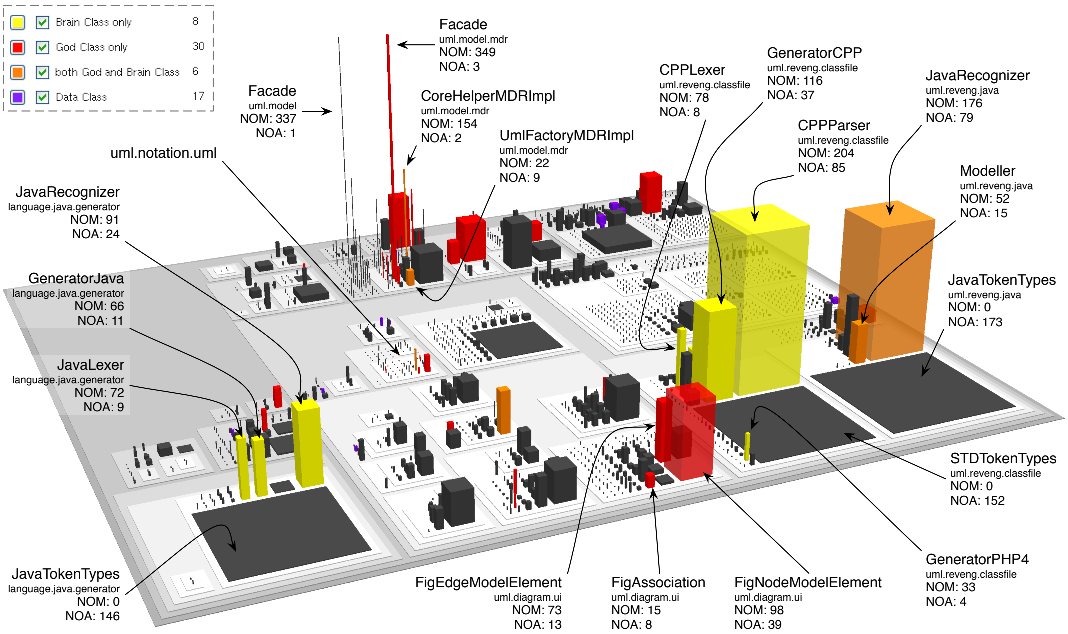
Another remarkable phenomenon comes in the form of the apparently tiny buildings affected by design disharmonies that imply an increased complexity (i.e., *God Class* or *Brain Class*). Inspecting one of these classes, called `Phrase` reveals that its scale is reduced only in the context of the iText system, as NOM=33 is a value considered very large for a Java class [LM06]. The disharmony map indicates it as a *God Class* and thus does not allow the maintainers of the system to overlook this potentially problematic class.

An extreme example is the one of class `SpecialSymbol`, which is a *God Class*, in spite of its apparently reduced size, i.e., only three methods and no attribute. However, a closer look at this class reveals that in terms of its 122 lines of code, it is by no means small. Another example of deluding disharmonious class is `ArabicLigaturizer`, which contains enough complexity in its only 11 methods to qualify as both a *God Class* and a *Brain Class*.

During our experiments we noticed no evident correlation between simple metric values (e.g., NOA and NOM) for a class and the disharmonies affecting it. To illustrates this observation, we present two classes with more or less the same magnitude in terms of the NOM and NOA metrics, yet which are complete opposites: While `BaseFont`, with 63 methods and 66 attributes, appears as a healthy class with respect to the class-level harmony, `PdfSignatureAppearance`, with 69 methods and 56 attributes, is both a *God Class* and a *Brain Class*, due to the complexity of its methods and to the way it collaborates with other classes.

ArgoUML

ArgoUML has 17 *Brain Classes* and 33 *God Classes*, 9 of which affected by both disharmonies, and 17 *Data Classes*. As revealed by the disharmony map in Figure 5.6, these disharmonious elements are not distributed all over the system, but rather sparsely.



🟨 ☑ Brain Class only 8
🟥 ☑ God Class only 30
🟧 ☑ both God and Brain Class 6
🟪 ☑ Data Class 17
Facade uml.model.mdr NOM: 349 NOA: 3
Facade uml.model NOM: 337 NOA: 1
CoreHelperMDRImpl uml.model.mdr NOM: 154 NOA: 2
UmlFactoryMDRImpl uml.model.mdr NOM: 22 NOA: 9
CPPLexer uml.reveng.classfile NOM: 78 NOA: 8
GeneratorCPP uml.reveng.classfile NOM: 116 NOA: 37
CPPParser uml.reveng.classfile NOM: 204 NOA: 85
JavaRecognizer uml.reveng.java NOM: 176 NOA: 79
Modeller uml.reveng.java NOM: 52 NOA: 15
JavaTokenTypes uml.reveng.java NOM: 0 NOA: 173
STDTokenTypes uml.reveng.classfile NOM: 0 NOA: 152
GeneratorPHP4 uml.reveng.classfile NOM: 33 NOA: 4
FigNodeModelElement uml.diagram.ui NOM: 98 NOA: 39
FigAssociation uml.diagram.ui NOM: 15 NOA: 8
FigEdgeModelElement uml.diagram.ui NOM: 73 NOA: 13
JavaTokenTypes language.java.generator NOM: 0 NOA: 146
JavaLexer language.java.generator NOM: 72 NOA: 9
GeneratorJava language.java.generator NOM: 66 NOA: 11
JavaRecognizer language.java.generator NOM: 91 NOA: 24
uml.notation.uml

Figure 5.6. Class-level disharmonies in ArgoUML

We look at the three formations—known to us from a previous case study presented in Section 3.4.2—each composed of one wide, flat building and two to three massive neighbor buildings. The first one resides in the `uml.reveng.java` district, and is made of the huge orange building (i.e., *Brain & God Class* `JavaRecognizer`), a smaller orange building (i.e., class `Modeller`), and a wide and flat building which looks like a parking lot (i.e., class `JavaTokenTypes` which contains 173 attributes). Although we would expect the latter to be a *Data Class* it is *not*, because all its attributes are declared as `final public`, i.e., they are pure Java constants.

The second similar package is `uml.reveng.classfile`, with two *Brain Classes*: the city's dominating building, class `CPPParser` (204 methods, 85 attributes), and the smaller affected one, class `GeneratorCPP` (100 methods, 34 attributes). The "parking lot" representing class `STDCTokenTypes` (152 attributes) serves as the repository for constants dedicated to the C++ parsing. Another example of elusive *Brain Class*, revealed only due to the availability of the disharmony data, is `GeneratorPHP4` with its 33 methods and 4 attributes.

The third similar package is `language.java.generator`, whose district is on the left side of Figure 5.6. It contains three *Brain Classes*: `JavaRecognizer` (91 methods, 24 attributes), `GeneratorJava` (66 methods, 11 attributes), and `JavaLexer` (72 methods, 9 attributes). As reported in our previous case study for program comprehension previously described in Section 3.4.2, having the same code twice (i.e., the two `JavaTokenTypes` interfaces share almost 150 constants) is questionable, yet less harmful in the case of generated classes and interfaces, which are not subject to manual maintenance.

By contrast, the three red buildings, which represent the *God Classes* `FigNodeModelElement`, `FigEdgeModelElement` and `FigAssociation`, located in `uml.diagram.ui`, are core classes and thus, very likely to be subject to continuous maintenance and changing requirements.

Another disharmonious agglomeration is a district characterized by a "forest" of very thin and extremely tall buildings (i.e., few attributes and many methods), representing package `model.mdr`. Out of its 35 classes, 8 are *God Classes* and 2 are *God & Brain Classes*. The doubly-affected classes are `UmlFactoryMDRImpl` (22 methods, 9 attributes) and `CoreHelperMDRImpl` (154 methods, 2 attributes). The largest affected class of this package, depicted by a building that literally touches the sky, is the *God Class* `FacadeMDRImpl` (349 methods, 3 attributes). All these classes are the only implementations of the interfaces `UmlFactory`, `CoreHelper`, and `Facade`, respectively. In spite of their large number of methods, the interfaces are not affected by disharmonies due to their lack of functionality. However, perceived through their implementing classes, these apparently harmless interfaces qualify as *God Class* and *Brain Class* factories. By analyzing ArgoUML's history in Chapter 4, we learned that the `Facade` interface (and by analogy the other interfaces in the hierarchy) had two concrete implementations in one of the versions of the system. Some versions later, one of the implementations disappeared, leaving the MDR implementations as the only one until these days. This potentially problematic package is therefore one of the case studies for the method-level disharmony maps, presented in Section 5.4.2.

Finally, we observed several hardly visible colored buildings in district `uml.notation.uml`: the *God & Brain Classes* `NotationUtilityUML` (24 methods, 6 attributes), `MessageNotationUML` (29 methods, 2 attributes), `AttributeNotationUML` (8 methods, 2 attributes), and the negligible `OperationNotationUML` (9 methods, 0 attributes). Since both disharmonies require high complexity, it was unexpected to find these apparently low-functional classes (i.e., reduced height) among the affected. To our surprise, these classes privately held the following amounts of code expressed in LOC: 1,240, 1,538, 432, and 450, respectively. These classes were not programmed in the object-oriented spirit and should be reviewed by ArgoUML's maintainers.

Jmol

The disharmony map of Jmol in Figure 5.7 shows the class-level problems of this system. Jmol contains 5 pure *Brain Classes*, 21 *God Classes*, 9 *God* & *Brain Classes*, and 83 *Data Classes*, the latter being also dominant disharmony of the Jmol.

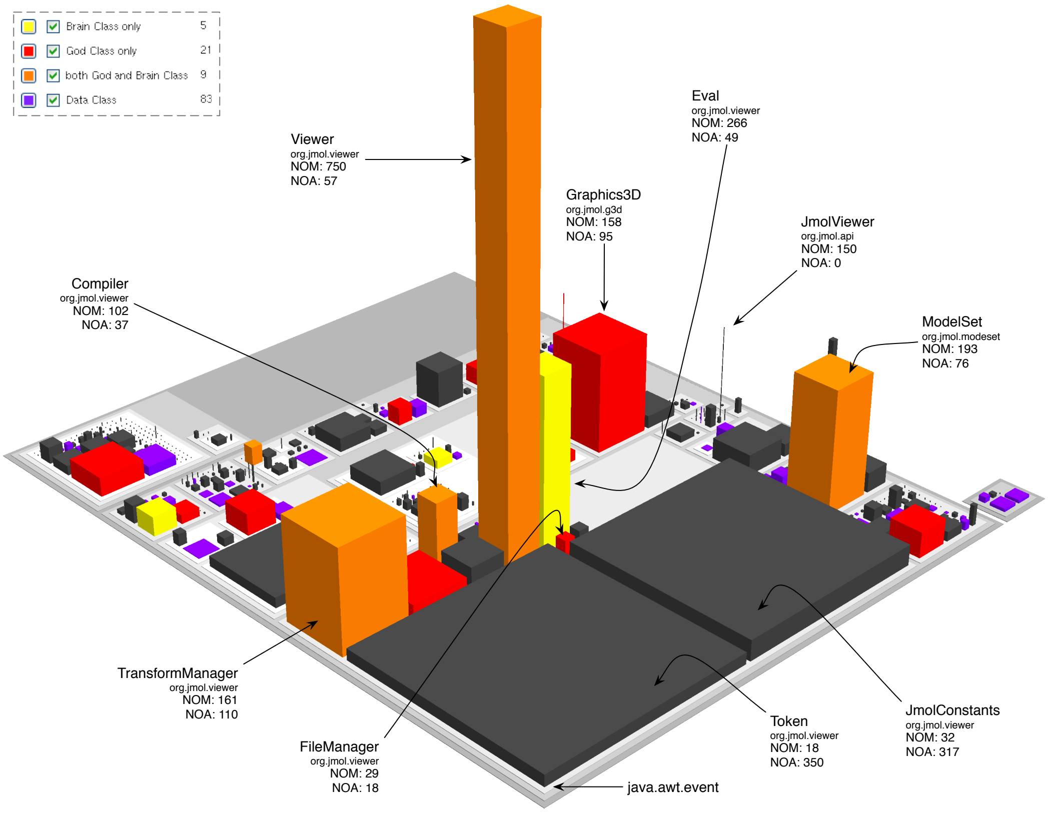

🟨 ☑ Brain Class only 5
🟥 ☑ God Class only 21
🟧 ☑ both God and Brain Class 9
🟪 ☑ Data Class 83
Viewer org.jmol.viewer NOM: 750 NOA: 57
Eval org.jmol.viewer NOM: 266 NOA: 49
Graphics3D org.jmol.g3d NOM: 158 NOA: 95
JmolViewer org.jmol.api NOM: 150 NOA: 0
ModelSet org.jmol.modeset NOM: 193 NOA: 76
Compiler org.jmol.viewer NOM: 102 NOA: 37
TransformManager org.jmol.viewer NOM: 161 NOA: 110
FileManager org.jmol.viewer NOM: 29 NOA: 18
Token org.jmol.viewer NOM: 18 NOA: 350
JmolConstants org.jmol.viewer NOM: 32 NOA: 317
java.awt.event

Figure 5.7. Class-level disharmonies in Jmol

The huge orange building in the center is class `Viewer`, whose 750 methods and 57 attributes make it the largest *God* & *Brain Class* we have visualized so far. To the right of this building we see the yellow building of *Brain Class* `Eval` (266 methods, 49 attributes) and at its base we see the tiny red building representing *God Class* `FileManager` (29 methods, 18 attributes). These three harmony breaking classes, which appear close to each other clearly illustrate the fact that there are degrees of severity with respect to design disharmony.

Around the central tower, there are three *God Classes*: `ModelSet` (193 methods, 76 attributes), `TransformManager` (161 methods, 110 attributes) and `Graphics3D` (158 methods, 95 attributes), the first two being also *Brain Classes*. This system would benefit from a reengineering effort, especially since it is a rather active project, given its over 8,000 revisions.

The Jmol system was also a case study for the software evolution application of our metaphor presented in Chapter 4, in the context of the *timeline* technique presented in Section 4.8. Unsurprisingly, many of the disharmonious classes which where discussed there are also key players in the design disharmony context, e.g.,`Graphics3D`, `Viewer`, or `Eval`.

5.4.2 Method-Level Disharmonies

To visualize method-level disharmonies, we use the fine-grained representation. Due to the fact that looking at entire systems using this granularity is impractical (i.e., too many depicted entities), we focus on specific parts of the systems.

Feature Envy in Jmol

After visualizing the method level disharmonies of Jmol using a fine-grained representation, we noticed that the *Feature Envy* design problem is by far the dominant design problem in Jmol. Figure 5.8 shows the *Feature Envy* disharmony map of the system using the *Progressive Bricks* adaptive layout. The methods affected by this problem are colored in yellow, while the unaffected methods are gray. The disharmony maps transmits the viewer the fact that this problem is rather severe in this system. Indeed, more than one quarter of Jmol's methods (e.g., 1,555 out of 5,968) exhibit the *Feature Envy* disharmony.

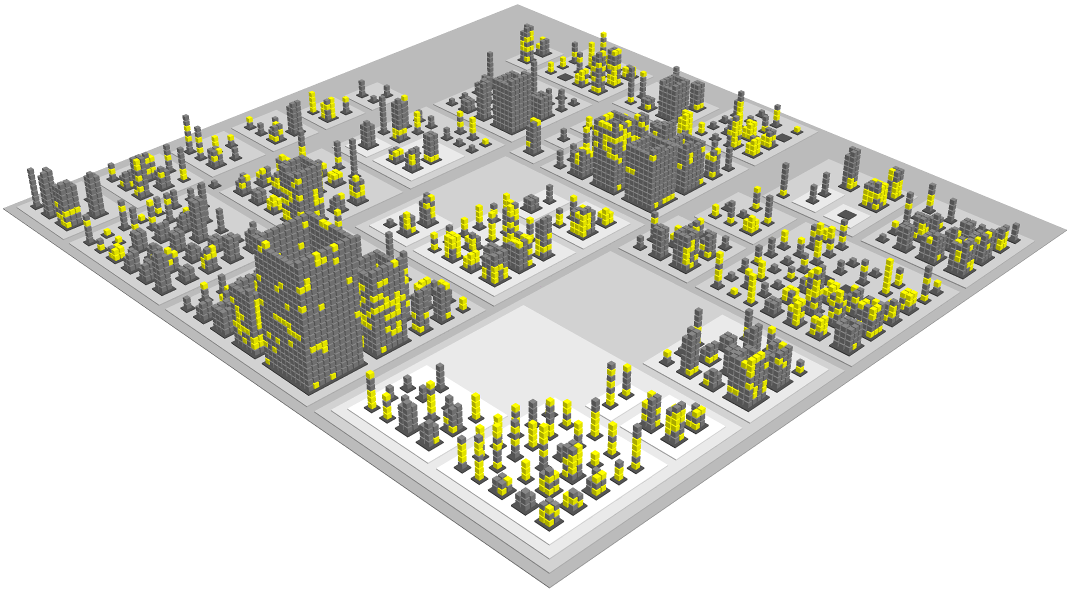


Figure 5.8. Yellow-colored **Feature Envy** in Jmol

Our visualization depicts the *Feature Envy* "epidemic" in a suggestive way and the picture says it all: the system could benefit from a serious session of reengineering.

Shotgun Surgery in ArgoUML

Figure 5.9 shows a visualization of package `model` in ArgoUML. This package, which was subject to discussions also during the class-level analysis, contains in this representation the two most massive buildings in the city of ArgoUML (i.e., depicting the classes or interfaces with the highest number of methods), representing the interface `Facade` and the only class that it, i.e.,`model.mdr.FacadeMDRImpl`.

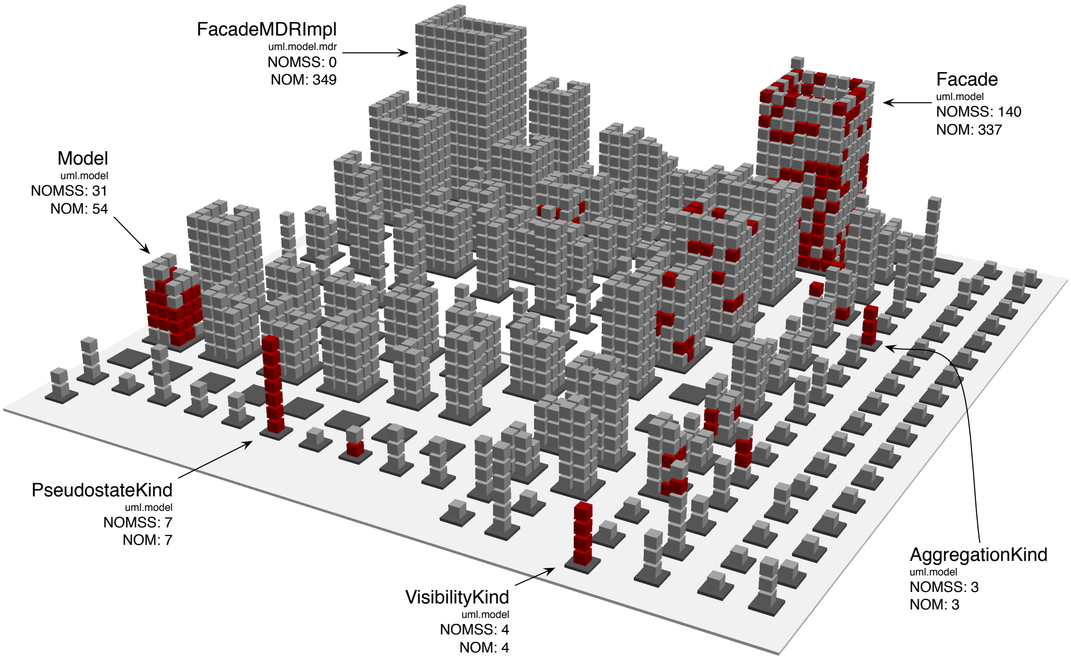


Figure 5.9. Red-colored **Shotgun Surgery** in the `model` district of ArgoUML

The dominating design disharmony characterizing this package is by far *Shotgun Surgery*, depicted by the many buildings “tainted” with the dark red color. Moreover, we see that most of the dark red “bricks” belong to only a reduced set of classes. The largest building affected by this disharmony is the `Facade` interface. In contrast to the class-level disharmonies discovered in this package, the method-level disharmonies are detected on the interface and not on the classes implementing it, due to the fact that the calls are done using polymorphism, i.e., they target references to the interface.

Of the 337 methods defined in `Facade`, 140 exhibit the *Shotgun Surgery* disharmony, as it is shown by the *Number of Methods with Shotgun Surgery* (NOMSS) metric. Apart from the `Facade` interface, there is another interface called `Model` which has many methods affected by Shotgun Surgery (31 out of 54) and three small classes made entirely of methods with Shotgun Surgery: `AggregationKind` (with 3), `VisibilityKind` (with 4), and `PseudostateKind` (with 7), respectively.

This disharmony is somewhat expected in this package, since it is part of the system’s model and all the other modules depend on it. A class with an increased number of methods affected by *Shotgun Surgery* is fairly difficult to change, since any change is likely to require many changes throughout the system.

5.5 Related Work

Among the approaches based on a city metaphor, only two addressed the visualization of design anomalies.

Panas et al. describe a visualization which shows a city with buildings in flames, obtained using their city metaphor, enriched with the *Lack of Documentation* (LOD) metric on fire textures [PEQ$^+$07]. However, the design problems we illustrate are much more complex than such simple metrics. We consider the approach of Panas et al. more related to the program comprehension application of the city metaphor, where we also mapped simple metrics on the city's visual properties.

The second approach is the one of Langelier et al., who addressed the problem of detecting design principle violations or anti-patterns by visually correlating outlying properties of the representations, e.g., a twisted and tall box represents a class for which the two mapped metrics have an extremely high value [LSP05]. There are several drawbacks of this approach in comparison to ours. First, the approach is prone to both false positives (e.g., classes which may seem good candidates for *God Class* or *Brain Class* due to their magnitude, but they are not, as seen in Section 5.4) and false negatives (i.e., classes delude through their apparently insignificant size, but in reality suffer from design problems typically found in very large classes). We have seen how the availability of precise design problem data removes the uncertainty and helps focusing on the parts of the city that are actually affected by such problems. The second problem with the approach of Langelier et al. is that for each anomaly one needs to map a different sets of metrics, because the number of metrics needed for the detection oftentimes exceeds the mapping limit of the representation (i.e., three visual properties: height, twist, and color). By using only one of the visual properties (i.e., color), we provide an overview of several design problems and, at the same time, allow the user to complement the mapping with other software metrics that may contribute to obtaining interesting insights, such as the ones we presented in the case studies.

To our best knowledge, all the previous work depicts software artifacts in terms of such low-level metrics and does not address the visual representation of such high-level design problems. To enable an efficient visualization of a system's design problems, we drew inspiration from disease maps and built on top of the previous techniques revolving around our city metaphor a technique called disharmony maps.

However, a key part of the success of applying the city metaphor to design quality assessment is owed to the detection strategies that we relied on for computing the precise design data we have access to in our approach. The detection strategies [Mar04a] were introduced by Marinescu as a mechanism to formulate complex rules using the composition of metrics-based filters, and extended later by Lanza and Marinescu [LM06] by formalizing the detection strategies and providing solutions for recovering from detected problems. The 2D polymetric views provided as means to visualize the systems do not explicitly illustrate the disharmonious artifacts, nor do they provide an overview and distribution of the disharmonies within the observed systems. In the context of detection strategies, Raţiu et al. aimed at further improving the design flaw detection by taking into account information from the history of the candidate classes to compute their persistency in exhibiting a particular design flaw during their lifetime [RDGM04].

5.6 Summary

The third application context for our city metaphor was design quality assessment. We presented a novel visualization called disharmony map, which is a code city whose color denotes the spread and distribution of design problems. By integrating the design problem data obtained by running detection strategies on software systems with the city metaphor, we obtain a visual approach able to provide an overview from the perspective of the quality of the system's design. This third extension of the city metaphor supports the claim of our thesis related to the versatility of our city metaphor.

We validated this third application of our city metaphor by means of several case studies. Using our approach, we learned about false appearances (e.g., small classes suffering from design problems that are typical for large classes), we saw how a bad organization of the package structure is accompanied by many disharmonies of its classes and how disharmonies can conquer a system in the absence of reengineering. All these insights are indications of another useful application of our city metaphor.

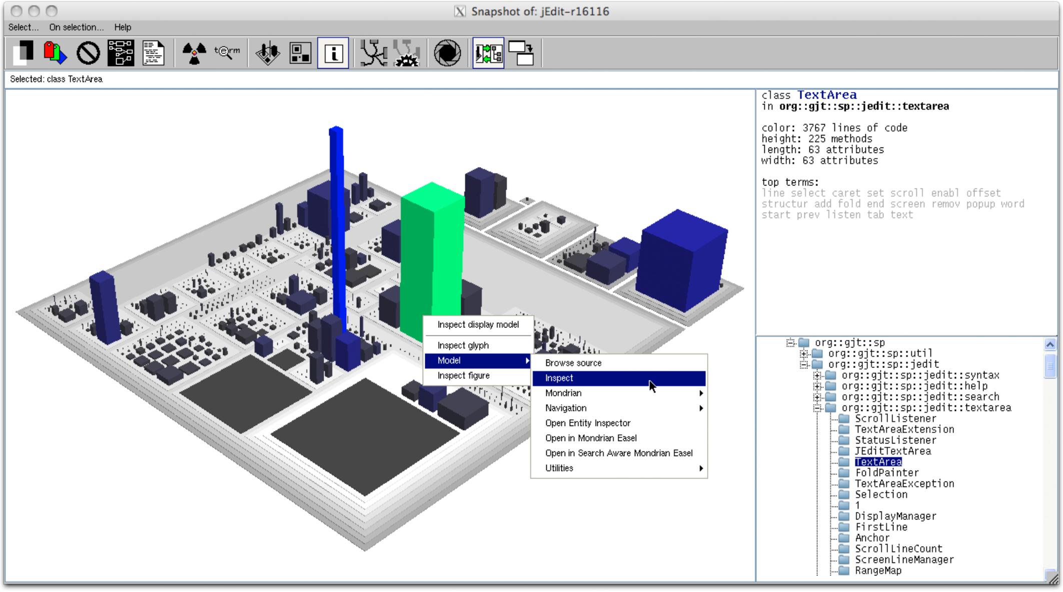
We believe that, with the three applications of our city metaphor, we demonstrated the first claim of our thesis, which refers to the versatility of the city metaphor for software visualization. The second part, which is yet to be proven, refers to the efficiency of the approach.

However, both the application of the city metaphor in different contexts and the empirical validation of our approach—presented later in this thesis—strongly rely on the tool support presented in Chapter 6.

Chapter 6

Tool Support

To support our approach we implemented a software visualization tool called CodeCity, with an essential role in both demonstrating the versatility of our metaphor for reverse engineering, by enabling us to apply our approach on real software systems, and supporting our empirical evaluation aimed at proving the efficiency of our approach.



Selected: class TextArea

```
class TextArea
in org::gjt::sp::jedit::textarea

color: 3767 lines of code
height: 225 methods
length: 63 attributes
width: 63 attributes

top terms:
line select caret set scroll enabl offset
structur add fold end screen remov popup word
start prev listen tab text
```

```
org::gjt::sp
  org::gjt::sp::util
  org::gjt::sp::jedit
    org::gjt::sp::jedit::syntax
    org::gjt::sp::jedit::help
    org::gjt::sp::jedit::search
    org::gjt::sp::jedit::textarea
      ScrollListener
      TextAreaExtension
      StatusListener
      JEditTextArea
      TextArea
      FoldPainter
      TextAreaException
      Selection
      1
      DisplayManager
      FirstLine
      Anchor
      ScrollLineCount
      ScreenLineManager
      RangeMap
```

Figure 6.1. CodeCity's main window

CodeCity is a sovereign visualization tool, i.e., an application that tends to occupy the entire screen estate [CR03]. Its user interface, presented in Figure 6.1, is composed of three panels:

- main panel (left), which enables visualizing, interacting with, and navigating code cities,
- information panel (top right), which shows contextual data on the inspected artifacts, and
- structure panel (bottom right), which shows the system's structure in a tree view.

6.1 The Process of Visualizing Software Systems as Cities

Visualizing software systems as code cities requires three steps:

1. **Extracting the model.** It is believed that the use of frameworks as underlying technology for software tools leads to faster tool innovation, as less time is spent on reinventing the wheel [Sto06]. CodeCity is implemented in Smalltalk and built on top of Moose[1], a reengineering platform whose energetic community has produced over 150 publications in its fourteen years of existence [NDG05, DGKR09].

   Moose provides an implementation of both the language-independent FAMIX [DTD01] meta-model, which is able to model software systems, regardless of the programming languages they have been written in (currently, it supports Java, Smalltalk, C++, and C#) and an implementation of a meta-model for history, called Hismo [Gîr05]. In Hismo, a history is a sequence of versions of the same kind of entity (e.g., class history, package history, etc.), where a version is a snapshot of an entity at a certain point in time. By building on top of the Moose technology, we get access to large amounts of rich, structured information—including a extensive set of software metrics—contained in the FAMIX models of software systems.

   The model extraction step implies parsing the software system's source code to produce a FAMIX model of the system. Depending on the programming language that the system is written in, we use different tools to parse the system: For Smalltalk, we use the parsing technology implemented in Moose. For Java and C++ systems, we use iPlasma [MMM+05], an analysis platform courtesy of the Loose Research Group[2]. For C# systems we use the PMCS (i.e., Parsing and Modeling C# Systems) tool [Dak09].

   Both external tools, i.e., iPlasma and PMCS allow exporting the FAMIX model using the MSE[3] exchange format. After importing the FAMIX models, CodeCity handles software systems uniformly, unconstrained by programming language boundaries. Moreover, if the models represent different versions of the same system, they can be merged in a model of the system's history.

2. **Building the visualization model.** Based on a FAMIX model of a software system, CodeCity builds a visual model of the software system, according to the user's preferences, in terms of property mapping strategies, layouts, granularity of the representation.

3. **Rendering the code city.** We built our rendering layer on top of the Jun graphics library [AHK+01], which enables us to render, using OpenGL[4] technology, the visual model of the software system as an interactive code city visualization.

[1] http://www.moosetechnology.org

[2] http://loose.upt.ro/iplasma

[3] http://scg.unibe.ch/wiki/projects/fame/mse

[4] http://www.opengl.org

6.2 CodeCity's Architecture

The module-level architecture of CodeCity, depicted in Figure 6.2, describes the system in terms of four modules, discussed next.

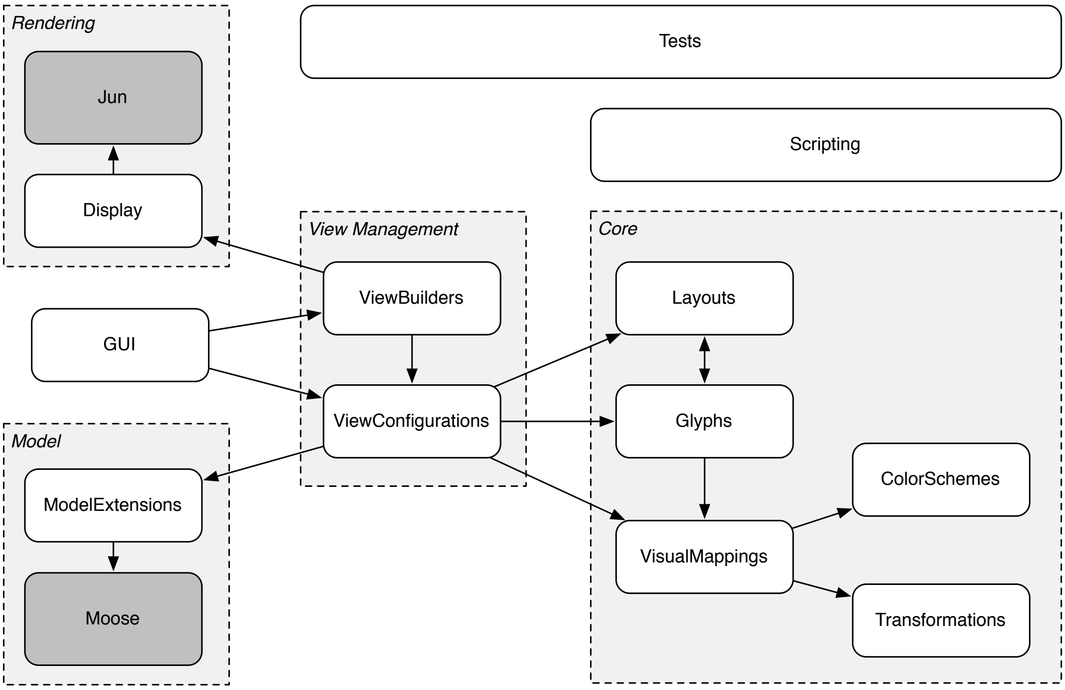


Figure 6.2. CodeCity's module-level architecture

Model deals with modeling the software systems we visualize with CodeCity. We built our tool on top of the `Moose` framework and extended the FAMIX meta-model using Smalltalk's class extension mechanism, in package `MooseExtensions`.

Core handles the visual model, which is made of figures (`Glyphs`), layouts (`Layouts`), and the mapping mechanism (`VisualMappings`). The latter, provides the means to map model properties (i.e., software metrics) onto glyph properties (e.g., size, color), and involves the computational parts of the packages `Transformations` and `ColorSchemes`.

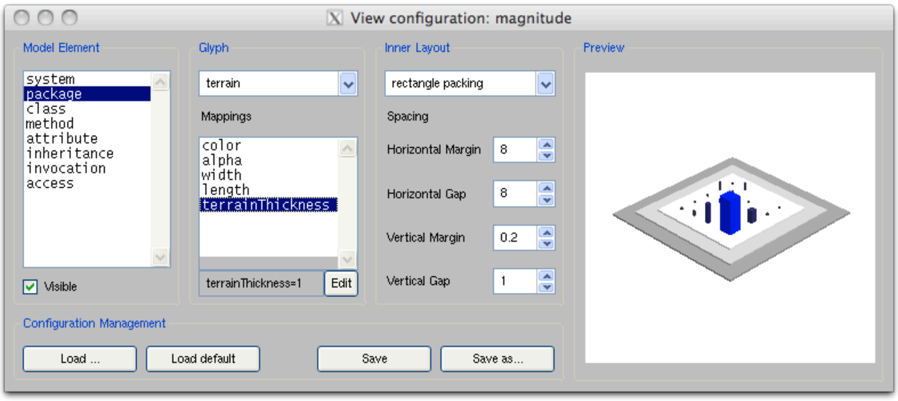
View Management handles the process of building views (`ViewBuilders`) and the view configuration mechanism (`ViewConfigurations`), presented next.

Rendering deals with making the visual model visible on the screen. We currently use the `Jun` framework as an OpenGL implementation, on top of which we built the `Display` package, which serves as an interface between CodeCity and Jun, and deals with the rendering, navigation, and interaction.

6.3 Flexibility through View Configurations

Measuring complex “organisms” such as software systems involves a broad range of different metrics. The display is a precious, yet limited resource, because the amount of information a view is able to present at any moment is limited. Moreover, displaying too many details can overwhelm the users. Therefore, it is crucial to allow the users to chose the software characteristics that are important for the task at hand. To provide assistance for a broad range of tasks, we strived for extensive configurability of CodeCity, achieved mainly through the view configuration mechanism. A *view configuration* is a specification defining for each model element type:

1. the *visibility*, i.e., whether to incorporate it in the visualization,
2. the associated *glyph* type,
3. the *layout* to use for its components (e.g., the layout for packages will be used to place sub-packages and classes), and
4. the visual *mappers* associated with each property of the chosen glyph.


View configuration: magnitude

Model Element: system, package, class, method, attribute, inheritance, invocation, access

☑ Visible

Glyph: terrain

Mappings: color, alpha, width, length, terrainThickness

terrainThickness=1

Inner Layout: rectangle packing

Spacing

Horizontal Margin: 8

Horizontal Gap: 8

Vertical Margin: 0.2

Vertical Gap: 1

Figure 6.3. User interface to the view configuration

View configuration tuning is done visually, using the graphical user interface in Figure 6.3, which enables the modification of every view configuration parameter. The preview panel shows how a very small code city would look like with the current view configuration, which allows one to quickly understand the effect of each configuration parameter on the visualization.

In terms of view configuration management, CodeCity allows both saving a useful configuration and loading a saved configuration, either for direct use or as a starting point to building new configurations. The view configuration objects are able to construct a building script, i.e., a string which, once evaluated by the Smalltalk compiler, returns a `ViewConfiguration` object equivalent to the original one. This mechanism, which relies on the reflection supported by Smalltalk, allows us to store a view configuration as source code and therefore manage it with the versioning system. The major advantage of this approach over a text-based one (e.g., saving the view configuration in an XML file) is that whenever a class that is part of the view configuration (i.e., any `Glyph`, `Layout`, or `Mapper`) is subject to a renaming refactoring [FBB$^+$99], the view configurations are updated automatically. The most complex components of a view configuration are the mappers, described next.

The Property Mapping Mechanism

The mapper class hierarchy, presented in Figure 6.4, provides access to various mapping strategies, including the three examples detailed in Section 3.3.2, i.e., identity mapping, box plot based mapping, and threshold based mapping. Each concrete mapper type has a corresponding user interface component, presented in Figure 6.5.

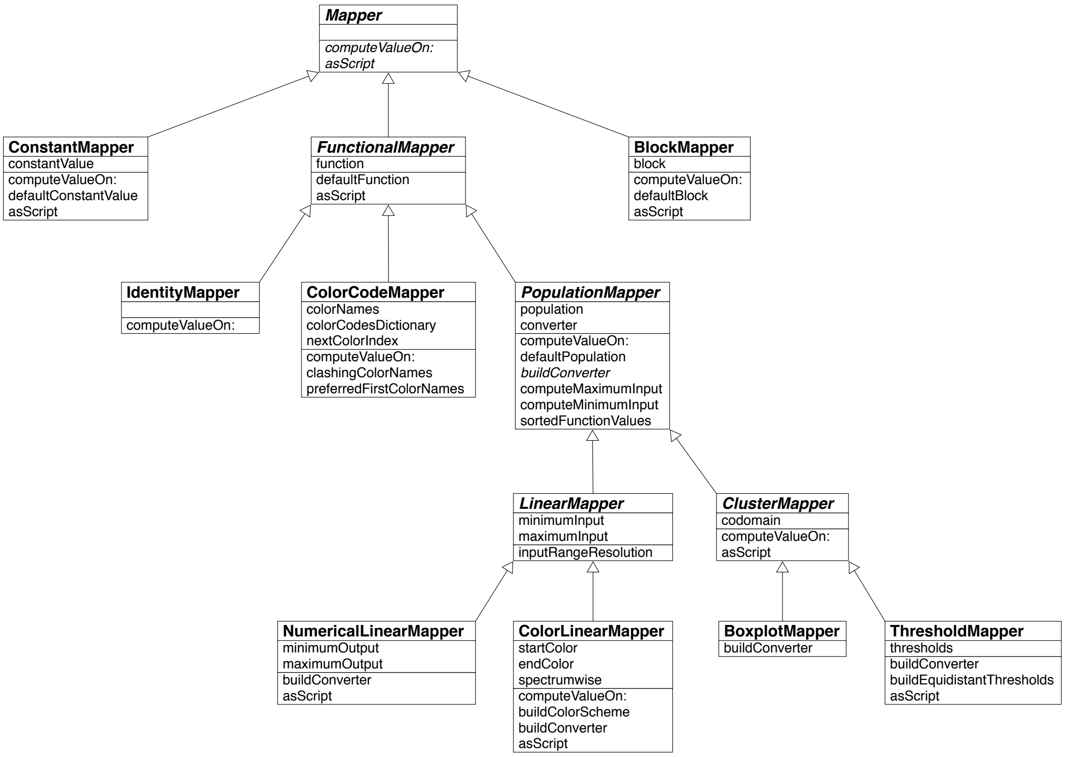


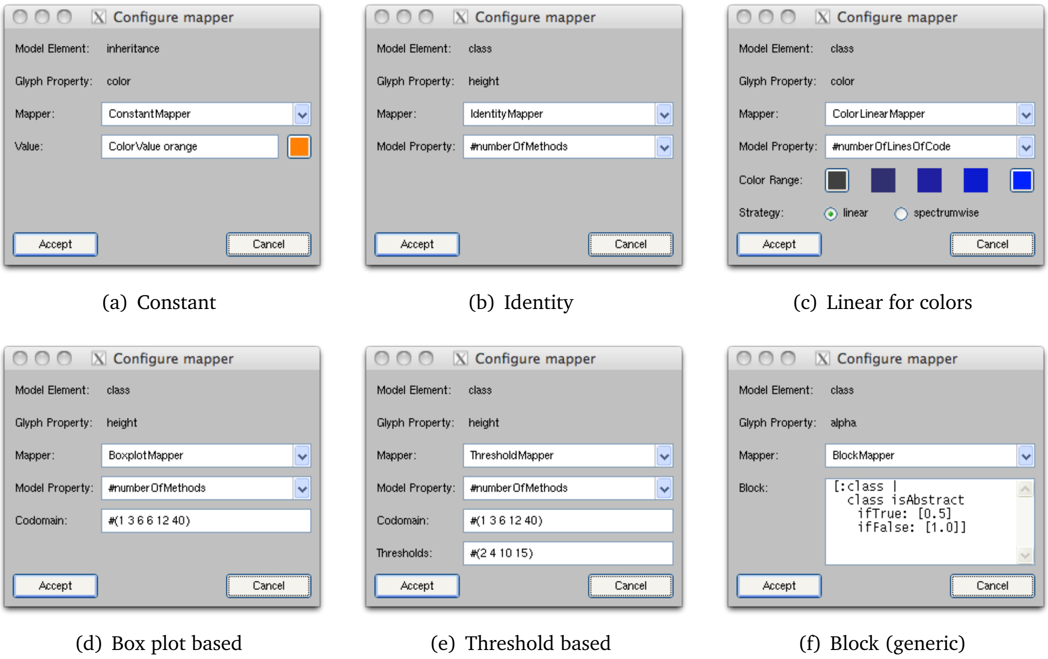
Figure 6.4. Class diagram of CodeCity's mappers

The key method of the mappers is `computeValueOn`, which is defined in the abstract class `Mapper`. This method takes a model element as parameter and returns the value of the corresponding visual property. For example, it takes a `FAMIXClass` object (i.e., the model of a class) as input and returns the numerical value corresponding to the height of the building representing that class.

A `ConstantMapper` returns the same value regardless of the input and is therefore used to map a common value to all elements of a particular type, e.g., all the edges representing inheritance are colored orange (See Figure 6.5(a)).

An `IdentityMapper` provides the most accurate representation of a metric. For example, we use an identity mapper to map the number of methods of a class on the building's height, as shown in Figure 6.5(b)).

For more complex mappings, we need to consider the metric values of all the elements—the population—of a particular model type. The simplest type of `PopulationMapper` is the `LinearMapper`, which scales the input values to a specified output range.



Model Element: inheritance
Glyph Property: color
Mapper: ConstantMapper
Value: ColorValue orange 🟧

(a) Constant

Model Element: class
Glyph Property: height
Mapper: IdentityMapper
Model Property: #numberOfMethods

(b) Identity

Model Element: class
Glyph Property: color
Mapper: ColorLinearMapper
Model Property: #numberOfLinesOfCode
Color Range: ⬛ 🟦 🟦 🟦 🟦
Strategy: ● linear ○ spectrumwise

(c) Linear for colors

Model Element: class
Glyph Property: height
Mapper: BoxplotMapper
Model Property: #numberOfMethods
Codomain: #(1 3 6 6 12 40)

(d) Box plot based

Model Element: class
Glyph Property: height
Mapper: ThresholdMapper
Model Property: #numberOfMethods
Codomain: #(1 3 6 12 40)
Thresholds: #(2 4 10 15)

(e) Threshold based

Model Element: class
Glyph Property: alpha
Mapper: BlockMapper
Block:

```
[:class |
  class isAbstract
    ifTrue: [0.5]
    ifFalse: [1.0]]
```

(f) Block (generic)

Figure 6.5. User interface widgets for the various mapping strategies

For example, the `ColorLinearMapper` uses a color scheme to convert the input property value into a color included in a specified range of the continuos optical spectrum. Figure 6.5(c) shows the color mapper we use for classes: the number of lines of code mapped on a color range from dark gray to intense blue.

The next mapper hierarchy, whose root class is `ClusterMapper`, divides the input range into five or six sub-ranges and assigns a single output value for each of them. The `BoxplotMapper` uses the box plot based technique to compute the boundaries between the sub-ranges corresponding to the *extremely low*, *low*, *average*, *high*, and *extremely high* categories. The box plot based mapper in Figure 6.5(d) maps a value $h \in \{1, 3, 6, 12, 40\}$ on the building's height, according to the cluster to which its number of methods metric value belongs to. Each output value corresponds to a building type in Figure 3.5.

The `ThresholdMapper` also divides the input range into sub-ranges, whose boundaries are based on statistical data. The threshold mapper in Figure 6.5(e) maps on the building's height a value $h \in \{1, 3, 6, 12, 40\}$, according to the cluster to which the number of methods metric value of its class belongs to. This example's input clusters, corresponding to the number of methods metric for Java classes, are: $[0, 2)$, $[2, 4)$, $[4, 10)$, $[10, 15)$, and $[15, \infty)$.

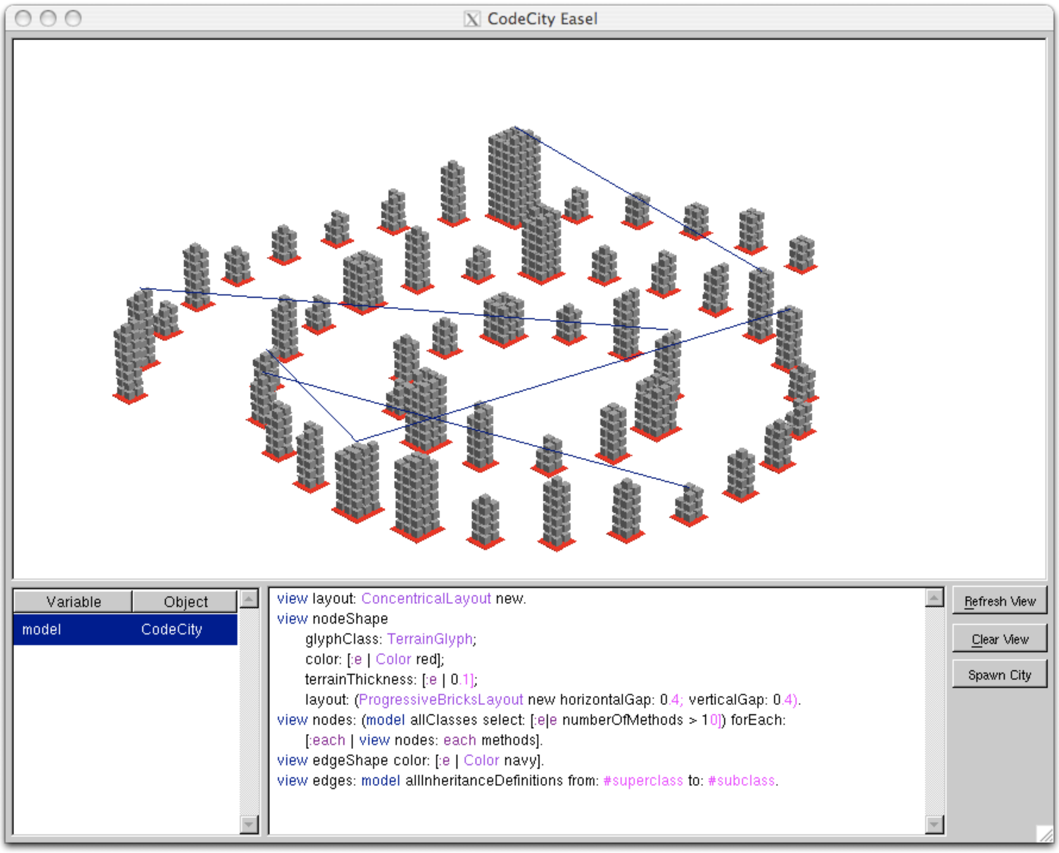
Finally, the `BlockMapper` is a generic mapper which requires Smalltalk skills[5] to "program" new ad-hoc mappers. Figure 6.5(f) presents a mapper which makes abstract classes semi-transparent ($\alpha = 0.5$) and concrete classes opaque ($\alpha = 1$).

[5] A block closure in Smalltalk is a self-contained piece of code which can be evaluated.

6.4 Prototyping Visualizations with Scripting

In spite of the extended flexibility of CodeCity, the graphical user interface does not grant full access to CodeCity's core, such as the one a programmer could benefit from. Moreover, the tool can only be used to visualize software system models based on a particular view configuration. However, by replacing the view building part, it is possible to visualize any type of structured information.

Therefore, inspired by Mondrian [MGL06] we implemented basic scripting support for building ad-hoc visualizations [Wet08]. This allowed us to experiment the feasibility of new visualizations before fully embedding them in CodeCity. Although in the meanwhile it would be possible to write the view building part of CodeCity using scripts, for performance reasons we preferred to keep it as it was, i.e., using domain knowledge to optimize the building step. An example of a simple script applied to the model of CodeCity itself is presented in Figure 6.6.



Variable	Object
model	CodeCity

```
view layout: ConcentricalLayout new.
view nodeShape
    glyphClass: TerrainGlyph;
    color: [:e | Color red];
    terrainThickness: [:e | 0.1];
    layout: (ProgressiveBricksLayout new horizontalGap: 0.4; verticalGap: 0.4).
view nodes: (model allClasses select: [:e|e numberOfMethods > 10]) forEach:
    [:each | view nodes: each methods].
view edgeShape color: [:e | Color navy].
view edges: model allInheritanceDefinitions from: #superclass to: #subclass.
```

Figure 6.6. Scripting example (bottom) and the produced output (top)

6.5 Interaction & Navigation

Interactivity is a key feature that pushes visualization beyond beautiful pictures. In the context of 3D visualization, navigability is a major feature, because it puts the users in control over the environment. CodeCity supports the following types of user interaction:

- *Examine.* To get a brief explanation on the physical properties of a city artifact, the user can hover the mouse cursor over the artifact and get a description of the underlying model and the values of the software metrics reflected by every visual property of the city artifact.
- *Select.* To interact with a city artifact in any way other than a quick examination, the user must first select it, by left-clicking on the element. With the current selection one can then perform operations, such as adding to or removing from the selection, clearing the selection, and inverting the selection.
- *Spawn.* Spawning a complementary view, i.e., isolating a group of elements in a new visualization is useful whenever the user needs to focus solely on a particular part of the system.
- *Mark.* During a longer exploration of a code city, color and transparency can be used as a marking mechanism. For example, after a first visual examination of a code city, one could assign a flashy color, such as red, to the buildings that make good starting points in the exploration of the system. Another example is using transparency to make the uninteresting artifacts less visible.
- *Query.* CodeCity provides the means to perform automated searches by indicating a set of criteria, which enable searching for artifacts that match a particular string, or type (e.g., all packages), or terms (i.e., vocabulary), or artifacts related to the current selection (e.g., all classes that invoke any of the methods of the selected classes). To enable the fast prototyping of new queries, we implemented a query engine and provided a generic query in CodeCity that allows us to build new queries, by writing short Smalltalk code snippets.
- *View contextual dependencies.* Using an opportunistic approach, CodeCity allows the user to visualize the dependencies of only the selected artifacts. The user can choose the types of dependencies (i.e., inheritance, invocation, or access), their direction (i.e., incoming or outgoing), and the color of the visual representation. The selection mechanism in this case is crucial, for it allows the user to change the scope of the dependency query. For example if a package is selected, the dependency context is made of all the classes defined in the package, while if only a class is selected, the context is reduced to that class only.
- *Navigate.* As opposed to some 3D visualization systems, where the viewer can rotate or move elements, our approach bears more similarity to video games, where the player is placed *within* an environment, and assigned limited capabilities. CodeCity allows a whole range of available movements: looking around, moving forward or backward, moving laterally, and orbiting horizontally or vertically.
- *Explore evolution.* The viewer can apply the *age map* color scheme and perform *time travels* through the history of a software system, and also generate *timelines* for any city artifacts.
- *Open disharmony maps.* The *disharmony map* functionality enables the users to get an overview of the design problems in the system.

6.6 Usability

In 2008, in the context of a Master's course on Software Design at the University of Lugano, we performed a usability study for CodeCity (and for other tools developed in our research group), with six students as participants. Prior to the study, CodeCity has been introduced to the subjects during a tool demonstration. After the subjects performed an analysis of the JHotDraw system in the context of software evolution, they were asked to answer a set of four questions aimed at informally evaluating its usability and its capability to support real software engineering tasks. The answers from our subjects are presented in Table 6.1, using a five-point Likert scale.

	Statement	Strongly disagree	Disagree	Undecided	Agree	Strongly agree
S1	CodeCity is easy to use.			2	4	
S2	CodeCity helps in building a first impression of the analyzed system.				4	2
S3	CodeCity's interactivity helps in analyzing systems beyond the first impression.			1	4	1
S4	CodeCity would increase your efficiency in solving software engineering tasks.				4	2

Table 6.1. The results from the questionnaire on CodeCity's usability

For each of the four questions, the mode of the data was *Agree*, which is an indication that most of the participants of our study agreed with all our four statements about CodeCity. Notably, we performed this study almost two years ago, when CodeCity was far from its current maturity. This study showed that early investments in the tool's usability do pay off.

6.7 Language-Independence, Scalability, and Performance

We illustrate the language-independence and the scalability of our approach by means of eight example systems, presented in Table 6.2, along with the time required to build the visualization.

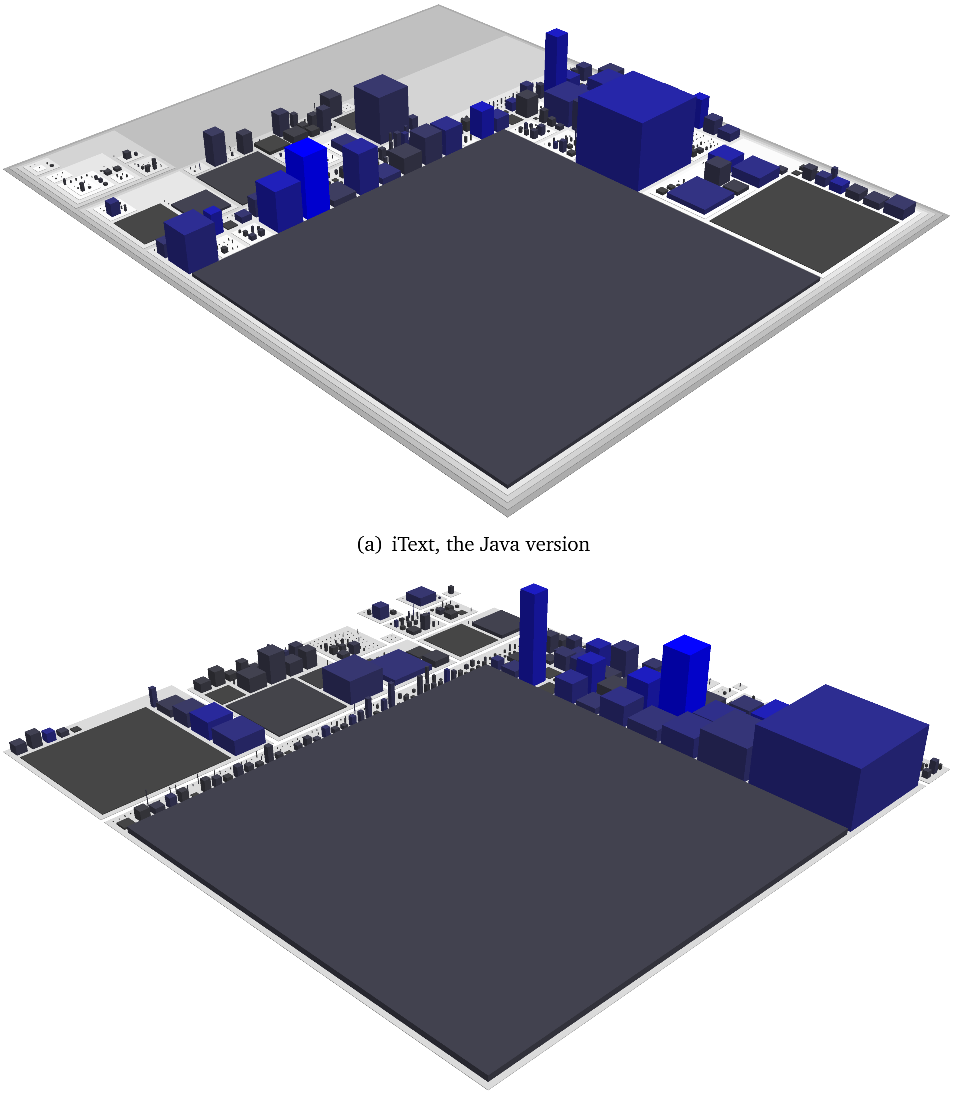
Name	Version	Date	Language	NOP	NOC	LOC	time (s)
iText	5.0.2	Apr 2009	Java	36	566	59,346	2.9
iTextSharp	5.0.2	Apr 2010	C#	22	485	57,917	2.7
Moose	3.2.171	Jun 2009	Smalltalk	48	474	35,555	2.8
ScummVM	1.1.1	May 2010	C++	141	3,117	304,815	7.5
GWT	rev. 25	Oct 2009	Java	302	4,372	211,858	8.5
JBoss	5.1.0.GA	May 2009	Java	1,507	7,881	434,943	10.4
JDK	1.5	Sep 2004	Java	664	12,888	1,084,606	31.6
Eclipse	3.5	Jun 2009	Java	1,800	27,900	2,871,016	34.4

Table 6.2. Visualization build time for a sample of our battery of visualized systems

Language-independence

The language-independence of our underlying meta-model (i.e., FAMIX), enables us to take our visual analyses beyond programming language barriers. We demonstrate this trait by visualizing software systems written in different programming language, i.e., Java, C#, C++, and Smalltalk.

First, we look at the code cities of the two implementations of iText, presented in Figure 6.7. iText[6] is a PDF library, started as a Java project in 2000, and ported to C# starting with 2003. In spite of the language differences, the two cities are very similar.



(a) iText, the Java version

(b) iTextSharp, the C# version

Figure 6.7. iText, implemented in both Java and C#

[6] `http://itextpdf.com`

Figure 6.8 presents the code city of Moose, a rather small system written in Smalltalk. Moose is the platform for software analysis used in our approach as underlying technology.

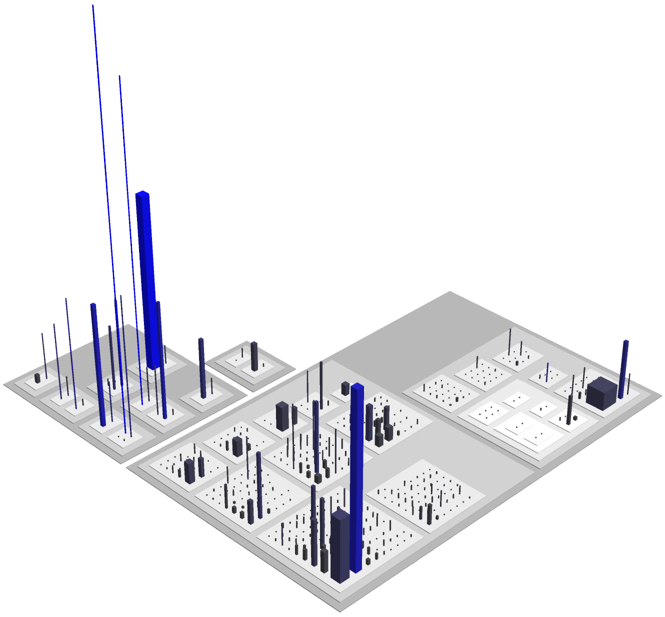


Figure 6.8. Moose, a Smalltalk system

In contrast with this Smalltalk code city, Figure 6.9 shows a bulky code city representing a C++ software system, called ScummVM[7]. ScummVM is a cross-platform interpreter for several point-and-click adventure engines.

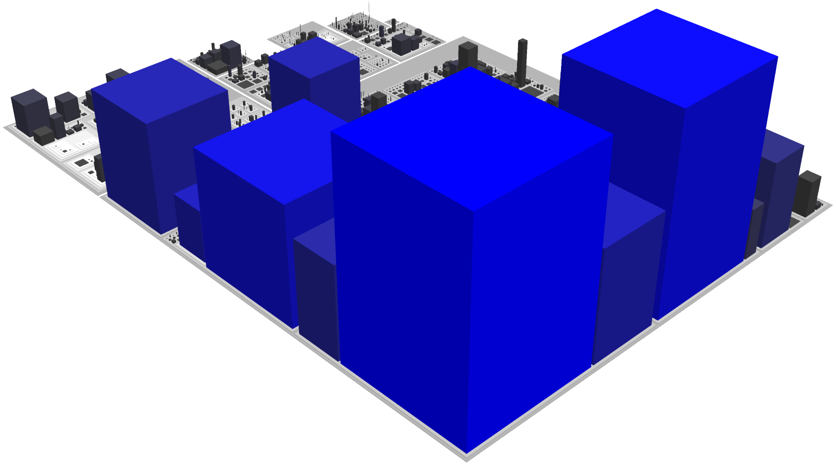


Figure 6.9. ScummVM, a C++ system

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

Scalability

Scalability has been acknowledged as one of the challenges of software visualization [Kos03, Sto06]. To illustrate the scalability of our approach, we present the code city visualizations of four other software systems. The first is GWT[8] (Google Web Toolkit), a development toolkit for building and optimizing complex browser-based applications. The source code of the GWT system has over 200,000 lines. The code city of GWT is presented in Figure 6.10.

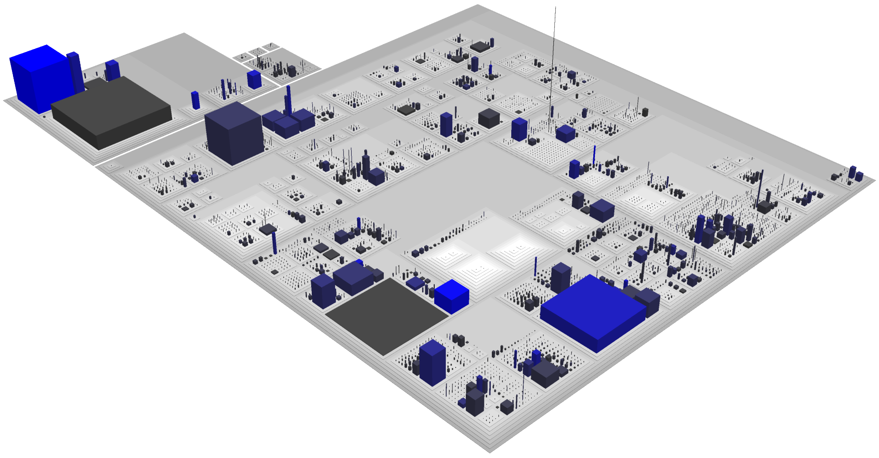


Figure 6.10. Google Web Toolkit (GWT), a system of 200+ KLOC

A software system more than twice the size of GWT is JBoss Application Server, with over 400,000 lines of code. Figure 6.11 presents the code city of the JBoss Application Server[9], which is an open source Java EE-based application server.

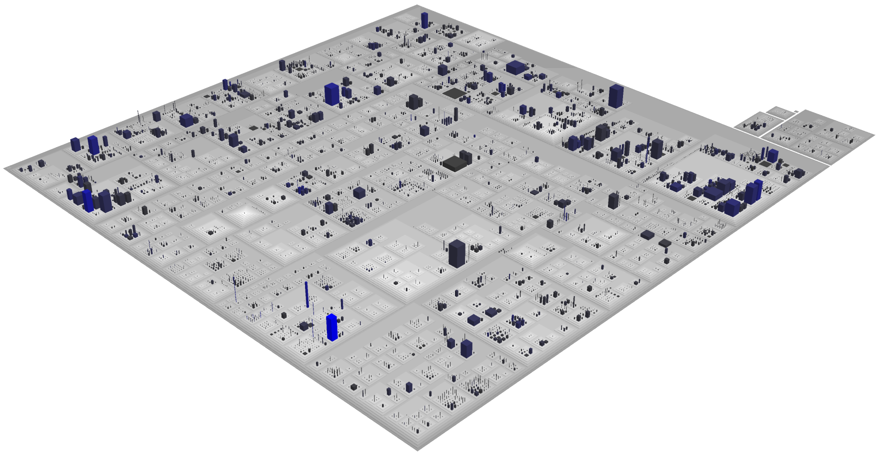


Figure 6.11. JBoss Application Server, a system of 400+ KLOC

[8] `http://code.google.com/webtoolkit`

[9] `http://www.jboss.org/jbossas.html`

The third code city, presented in Figure 6.12, represents JDK, a system with over one million lines of code. The Java Development Kit (JDK) is a Java platform[10], consisting of the API classes, a Java compiler, and the Java Virtual Machine interpreter.

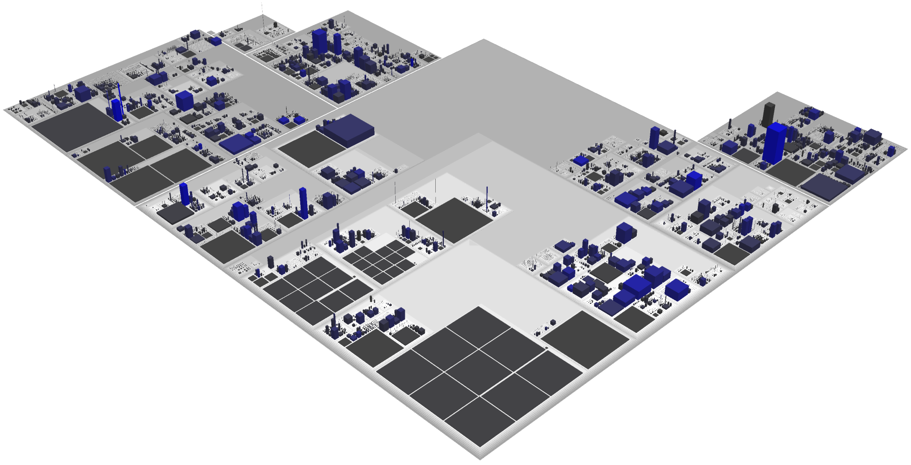


Figure 6.12. JDK, a system of 1+ MLOC

The largest system we have visualized so far is Eclipse, a system which totals almost three million lines of code. Eclipse[11] is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system.

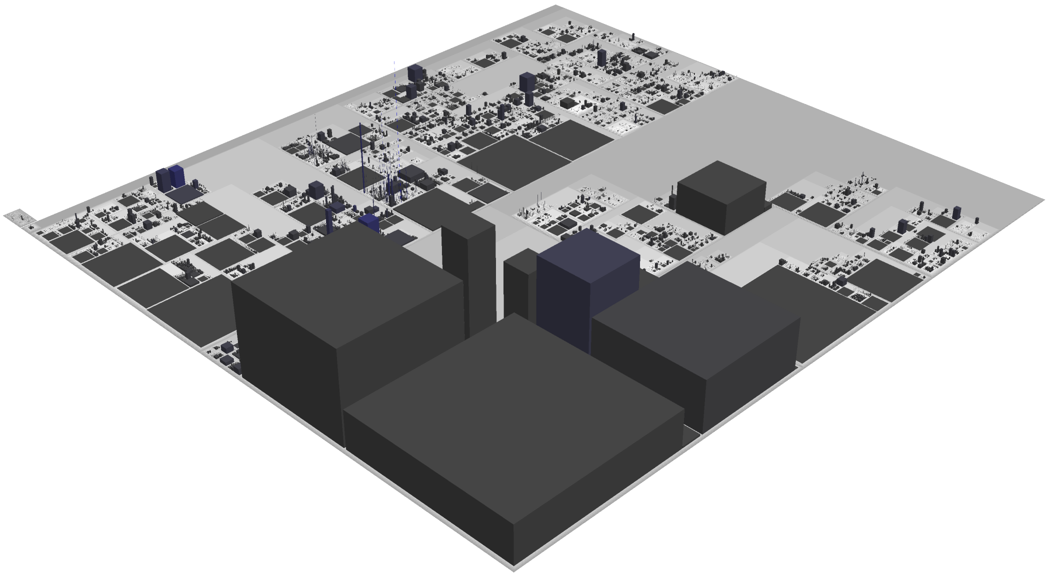


Figure 6.13. Eclipse, a system of nearly 3 MLOC

[10] http://java.sun.com/javase

[11] http://www.eclipse.org

In the context of scalability and language-independence, we built—as an exercise for the scripting support of our approach—a visualization which allows cross-language analyses, presented in Figure 6.14. The visualization presents ten of our case studies presented so far, each written in one of the four programming language supported so far by our fact extractors.

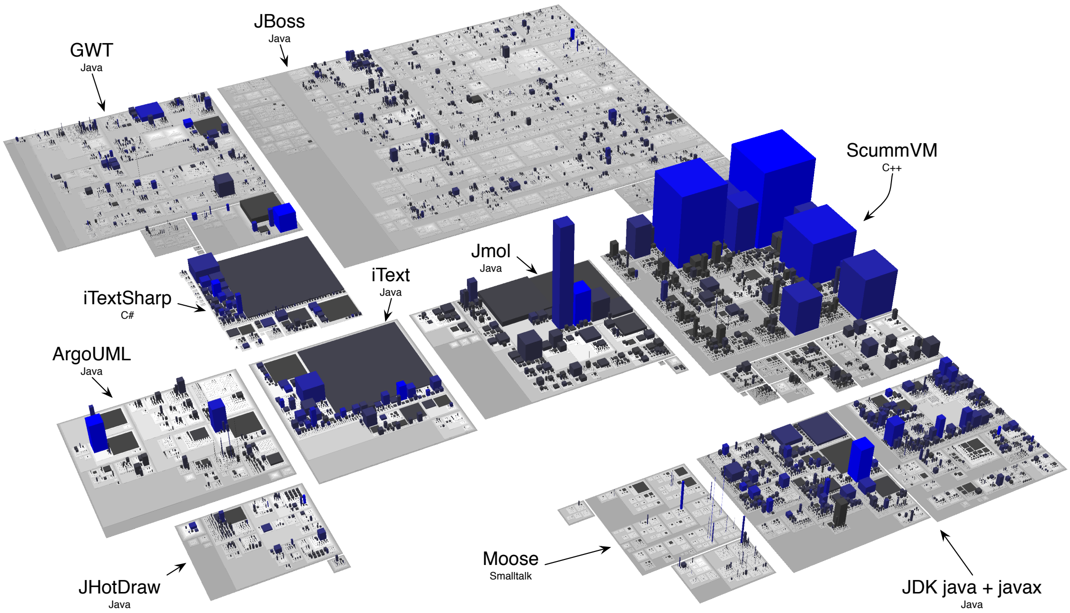


Figure 6.14. A cross-language visualization of ten systems, totaling 1.7+ MLOC

For this visualization we used the scripting support, because the visualization of several software systems simultaneously is something for which CodeCity has not been intended. However, the scripting allows us to experiment with ideas and evaluate a direction before implementing it in the tool. A potential materialization of the idea illustrated in Figure 6.14 is the visualization software ecosystems, i.e., collections of software projects which co-evolve in the same environment [Lun09].

Performance

To assess the time performance, we measured the average visualization build time over ten runs, based on the following hardware configuration: MacBook Pro with one 2.4 GHz Intel Core 2 Duo processor and 4GB RAM, running Mac OS X version 10.6.4.

The performance times presented in Table 6.2 show that, even for a large system such as Eclipse, building its code city which comprises around 30,000 artifacts takes less than one minute. This is an encouraging result in the context of scalability, i.e., a challenge that every software visualization tool needs to face.

The scalability of CodeCity is owed to an appropriate granularity level, to the underlying meta-model, and to the numerous optimizations that our tool was subject to over time.

6.8 Availability

Although the city metaphor has been explored in the past in an academic context, the software industry had practically no contact with such visualizations. For we believe in the value of tool building and in the credibility that a robust tool can bring to the scientific research, from the very beginning we backed our research activity with the development of CodeCity.

With each new application of our city metaphor to a new context (i.e., program comprehension, software evolution, and design quality assessment), CodeCity accordingly enlarged its application realm.

In March, 2008, we made CodeCity publicly available[12] for download, with support for every major platform (i.e., Mac OS X, Windows, Linux). Making CodeCity public has been a great opportunity for improvement, because it allowed users everywhere to experience our approach and consequently to send us feedback. Our users have given us several ideas, of which some ended up being integrated in the tool, such as exporting CodeCity visualizations to high-resolution images, a feature added by popular request.

The web site features a "hall of fame" page, which is a collection of various code cities. During our work on software evolution analysis, discussed in Chapter 4, we asked several developers to confirm several hypotheses we proposed about some insights we gained on the systems they were developing. Using the 'hall of fame" we were able to present preliminary results to these developers, one of whom was quite enthusiastic to look at their system from a new perspective: "I had a look at your city of JHotDraw and it looks like a city I would like to live in".

To support our users, we created a suite of video tutorials, which by means of hands-on examples help new users learn the basic functionality of CodeCity. Later, when we designed and conducted a controlled experiment for the validation of our approach, we used the video tutorials as learning material for the participants.

We have evidence that CodeCity is being used in both academic research and industry. Over the last two years, the tool which has reached its fifth release, has been downloaded 5,000+ times.

6.9 Summary

In the software visualization field, research and tool building intertwine, in a perpetual play between ideas and feasible solutions. In spite of the lack of attention from a large part of the community, the tool is often the supporting actor in promoting the research ideas behind it. A visualization approach is perceived at most as good as its implementation, because the tool is the means to present the approach. A brilliant idea demonstrated with a poor implementation may never reach the audience. Second, a tool provides the means to validate a new approach on case studies and find out whether it meets the expectations early on. Finally, a solid tool is mandatory for a controlled experiment for the empirical evaluation of the approach, in particular with subjects from industry.

In this chapter, we presented CodeCity, a tool that supports our city metaphor in all the three application contexts we designed it for, i.e., program comprehension, software evolution analysis, and design quality assessment. CodeCity allowed us to apply our approach based on the city metaphor on several open-source software systems and incrementally build confidence in our approach.

[12] http://codecity.inf.usi.ch

We aimed for the increased configurability of our tool, which we acquired by means of both a flexible design and scripting support. As opposed to the typical research prototypes, CodeCity has been improved in terms of usability, which made the tool more prepared for the use in an industrial context and easier to use, as witnessed by the subjects of our usability study. Furthermore, we performed several optimizations of CodeCity, which led to better time performance and increased scalability.

Since we can only perform the evaluation of our approach by means of the tool, the outcome of such an evaluation depends not only on the soundness of the approach, but also on the efficiency and usability of its implementation. Once again, the role of the tool is essential because it enables us to perform a controlled experiment for the evaluation of the city metaphor, presented in Part III.

Part III

Evaluation

Preview

By instantiating our city metaphor in the contexts of program comprehension, software evolution analysis, and software design quality assessment, we demonstrated the versatility of the city metaphor for reverse engineering. Moreover, by means of several case studies we showed that our approach supports the users in performing various analyses, which led to a number of interesting insights in the analyzed systems.

In this part we address the second claim of our thesis, which states that the city metaphor enables the building of efficient software visualizations. To assess the efficiency of our approach in supporting various reverse engineering analyses, we planned and conducted a controlled experiment whose main goal was to discover whether, under which circumstances, and to whom our approach is useful.

In this part we describe the controlled experiment in terms of design, operation, and results.

Chapter 7 deals with the design of our experiment, which is partly inspired from the strengths and weaknesses identified during an extensive survey of the work related to the empirical validation of visualization approaches.

In Chapter 8, we describe both the operation of the experiment and the results of the various analyses we performed based on the collected data.

Chapter 7

Experimental Design

7.1 Introduction

A successful experiment is one that reveals the facts, regardless whether it supports or rejects the tested hypothesis. However, there are many factors that can negatively affect the success of a controlled experiment. The first key factor is the set of decisions known as the experiment's design. Poor experimental designs may cause the rejection of true hypotheses or the acceptance of false hypotheses. To minimize this risk, we invested a great effort into the design of our experiment, which started from a study of the related work, presented next.

7.2 Learning from Related Work

There is a rich body of research on empirical evaluation by means of controlled experiments. To identify both good practices and commonly occurring mistakes, we first conducted an extensive study of the literature. The lessons extracted from this study are synthesized in a list of *desiderata* for the design of our experiment. Given the wide span of related work, we limit the discussion of the related work to the contributions that influenced the design of our experiment.

7.2.1 Guidelines for Information Visualization Evaluation

Software visualization is rooted in information visualization. Therefore, we start our study with the empirical evaluations of information visualization approaches.

Plaisant acknowledges the challenge of information visualization evaluation, but also its major role in increasing the credibility of tools towards industry adoption [Pla04]. Two important matters emphasized in this work are the use of real datasets and the demonstration of realistic tasks. Moreover, based on several reviewed experiments, Plaisant observed that tools perform differently for different tasks and, consequently, the composition of tasks can favor one tool over another when measuring overall performances. Therefore, to allow potential adopters to match tools with their own tasks, Plaisant recommends reporting on individual tasks rather than overall. The author also signals the urgent need for both task taxonomies and benchmark repositories of datasets and tasks.

In their analysis of user evaluation studies in information visualization [ED06], Ellis and Dix identified a set of problems that occur in user evaluation studies and discussed a number of solutions to these problems, which can be applied when designing and conducting evaluation studies in information visualization. The authors claim that empirical evaluation of visualizations is methodologically unsound, because of the generative nature of visualizations. It turns out that we cannot find perfect justifications of the observed results, because reasoning based on our incomplete knowledge of human perception is flawed. The authors do not advocate against empirical evaluations, but rather plead for a restrained interpretation of their results. Another issue they discussed was finding a balance between good science and "publishability": On the one hand, when evaluating solely aspects that are questionable, one is more likely to find problems in the visualization. On the other hand, when evaluating aspects that are on the safe side, it is practically impossible to learn something from the experiment, in spite of the potentially significant results. An interesting observation was that in open-ended tasks, the time a user spent on a task does not necessarily reflect the actual time required to finish it, but may also show how much they enjoyed themselves solving it.

Zhu proposed a framework for the definition and measurement of effective data visualization [Zhu07], according to three principles: accuracy (i.e., the attributes and structure of a visual element should match the ones of the represented data item), utility (i.e., an effective visualization should help users achieve the goal of specific tasks), and efficiency (i.e., an effective visualization should reduce the cognitive load for a specific task over non-visual representations). However, the great challenge in this context, which is finding concrete means to measure these effectiveness metrics, has unfortunately not been solved by the author with concrete solutions.

7.2.2 Empirical Evaluation in Information Visualization

In information visualization there are many controlled experiments which compare the efficiency of several tools presenting the same data. Since information visualization tools are more general than software visualization tools, the evaluations are not always task-centered, and even if they are, the tasks tend to be less focused than the ones in software visualization.

Petre shares some timeless insights which, although aimed at visual programming, are valid for software visualization as well [Pet95]. In this work, the author focused mostly on the differences between novice users and experts, briefly discussed here. First, the expert knows where to look, which is not so obvious for a novice. Second, there is a major difference in the strategies employed by experts and novices in using a graphical environment. While reading a textual representation is straightforward—due to the sequential nature of text—reading a graphical representation in two or three dimensions requires the reader to identify an appropriate reading strategy. Finally, an expert knows how to exploit cues outside of what is formally defined—information invisible to a novice. To support her claim that "looking isn't always seeing", Petre distinguishes experts by their ability to "see", which allows them to both perceive as important the information relevant to solve a task and to filter out inessential information. We support this observation and take it into account in the design of our experiment, by using blocking—distributing our subjects in groups featuring similar characteristics—based on the experience level of our subjects.

An early work in evaluating 3D visualization designs is the one of Wiss et al. [WCJ98]. The authors tried to isolate the design from the implementation and to evaluate it in isolation. For this, they implemented three existing 3D information visualization designs: the Cam Tree, the Information Cube, and the Information Landscape. The object system was a data set with 30 leaves and 8 internal nodes and an electronic newspaper's table-of-contents with 56 leaves and 14 internal nodes. The authors compared the three designs and concluded that each approach encountered problems with different data sets and that there was no absolute winner. At the end, however, the authors acknowledged that, by evaluating any information visualization design in isolation, one can only look at whether it can be used for implementing a task or not. This conclusion strengthens our belief that, in order to test a visual approach, one needs to test its implementation.

Stasko et al. [Sta00] presented the results of two empirical studies of two visualization tools for depicting hierarchies, implementing two space-filling layout methodologies, i.e., Treemap and Sunburst. The authors, who have developed certain assumptions about the strengths and weaknesses of each of the two approaches, used the empirical studies to test these assumptions. The experiment had 32 students as participants and 16 short tasks (i.e., with a maximum time limit of 1 minute), typical of operations that people perform on file systems. Besides correctness, the authors also analyzed average completion time per task, but only on correct tasks. An interesting fact about this work is that the authors analyzed the strategies taken (in terms of basic operations upon the tools) by the users to solve each task.

Kobsa presented the results from an empirical study, in which he compared three commercial information visualization systems (i.e., Eureka, InfoZoom, and Spotfire), based on tasks performed on three different databases [Kob01]. There were 82 student participants, and they had to solve 26 tasks (i.e., small tasks that can be solved in 1-2 minutes each) in three blocks of 30 minutes. Kobsa acknowledges that the more complex the tasks are, more factors may influence the outcome of the study, such as the ability of the subjects to understand the tasks and to translate them into available visualizations and operations upon these visualizations.

In another work, Kobsa compared five tree visualization systems (i.e., Treemap, Sequoia View, BeamTrees, Star Tree, and Tree Viewer) to the Windows Explorer baseline[Kob04]. There were 15 tasks and the object system was a hierarchy representing a subset of a taxonomy of items on eBay. The participants were 48 students with at least one year of experience working with computers, and the design of the experiment was between-subjects. The subjects were allowed a maximum of 5 minutes per task and were recorded with screen recording software. This allowed the experimenters to perform a post-experiment analysis in order to try to explain the differences in performance, and to observe interesting insights in relation to each tool. Interestingly, the most preferred tool turned out to be the non-visual, yet popular, Windows Explorer.

Kosara et al. [KHI$^+$03] addressed a set of questions around user studies, drawing attention upon the importance of studying a technique in an application setting, since one cannot assume that low-level results automatically apply to more complex contexts. The authors remark that the comments from participants are often more important than the other data an experimenter collects and that observing how professionals use a tool or technique is vital. They also acknowledge that, unfortunately, researchers are discouraged from publishing null results (i.e., original hypotheses not supported by the data), in spite of the intrinsic value of such shared experiences.

We applied several lessons we learned from this work. First, we designed tasks that are not trivial, but rather close in complexity to realistic tasks, and yet solvable in a limited amount of time. During our experiment runs, we gathered many observations from our subjects, both formally, via questionnaires, and informally, by verbal communication. Moreover, on one occasion, we had the chance to watch professionals using our tool in their own after-work environment, i.e., during a user group meeting.

O'Donnell et al. [ODB06] present an evaluation experiment for their PieTree visualization. Before the formal evaluation (i.e., the actual experiment run) the authors performed two rounds of informal evaluation to discover usability problems. For the informal evaluation the subjects were eight postgraduate students and the objects two fictional data hierarchies of 12 and 125 nodes. The formal evaluation was conducted with 16 students, most of them postgraduate. While in the informal evaluation they compared PieTree in conjunction to a TreeView with TreeMap, in the formal experiment they compared the use of PieTree in conjunction with a TreeView to the use of the PieTree or of the TreeView alone. This comparison chosen by the authors is poor, because it shows at best that the two approaches are better than any one of them taken separately, instead of trying to prove the usefulness of the PieTree approach created by the authors. The experiment took place with one subject at a time, which allowed the authors to observe a number of common strategies used by the subjects to solve the tasks and discuss how these strategies influenced the results. The main lesson that the authors learned with their experiment is that the results depend not only on the support provided by the tool, but also on the users and on their capability to translate the tasks into interactions with the visualization. The results of the comparison, which indicated that the combination of the two tools was outperformed by the use of one of the tools in every task, show that more is *not* always better.

7.2.3 The Challenges of Software Visualization

Since tool support is a key factor for the evaluation of software visualization approaches, the challenges of software visualization tools are important for empirical evaluations.

In the context of the theories, tools, and research methods used in program comprehension, Storey places an important emphasis on visualization [Sto06], whose challenges include dealing with scalability, choosing the right level of abstraction, and selecting which views to show—all problems we needed to handle to provide a tool that can stand the test of evaluation.

Koschke performed a research survey on the use of software visualization in the fields of software maintenance, reverse engineering and re-engineering and synthesized the perspectives of 82 researchers [Kos03]. According to this survey, the vast majority of the researchers believe that visualization is absolutely necessary or at least very important to their domain, a result considered overrated by the author of the survey. Koschke brings up a set of observations, pointing out the space for improvement. Despite the fact that visualization has come to be perceived as particularly appropriate to give an overview of a large information space, several researchers stated that it is only suited for small-to-medium-sized systems, and one of the participants indicated that for large systems or for systems with an overwhelming number of dependencies, queries are preferred over visualization.

From the perspective of existing representations for software visualization, graphs are by far the dominant one, while metaphors are covered by only 1% of the approaches. This insight gives a clear indication of the quantity of research invested in each of these directions and strengthens our belief that we are investigating a less uncovered, and thus potentially valuable direction.

Some of the challenges of visualization mentioned by Koschke are scalability and complexity, source code proximity (i.e., maintaining a link with source code), and integrability of visualization in processes and tools for maintenance, reverse engineering, re-engineering, and forward engineering. Finally, an interesting aspect is the subjectivity of most researchers, who consider the appropriateness of their own visualization as a given, without any empirical evidence whatsoever. However, the justified expectation of the research community for evaluation through controlled experiments is challenged not only by the creators' subjectivity, but also by the cognitive nature of the tasks supported by software visualization.

7.2.4 Program Comprehension Tasks

Differently from the information visualization field, where the focus is more on perception, the evaluations of software visualization approaches are based on task solving. Therefore, finding the tasks for the experiments is of major importance. We looked at the existing frameworks and at the tasks used in controlled experiments for the validation of reverse engineering and program comprehension approaches.

Based on qualitative studies performed with participants from both industry and academia, Sillito et al. defined a set of questions that developers ask during a programming change task [SMDV06]. However, this valuable framework focuses on the source code level and supports mainly developers. Therefore, it is not appropriate for the evaluation of our approach, which supports not only developers, but also architects, designers, and to a certain extent project managers, in solving high-level reverse engineering and comprehension tasks.

Pacione et al. [PRW04] proposed a model for evaluating the ability of software visualization tools to support software comprehension. According to their model, a tool or approach is characterized by three dimensions: level of abstraction (i.e., the granularity of the visualized data), facets of software (i.e., structure, behavior, data), and type of analyzed data (i.e., static or dynamic). The authors defined a set of comprehension activities that should be supported by visualization tools and a set of tasks which are mapped on the comprehension activities. However, in spite of its apparent generality, this model is heavily biased towards dynamic data (e.g., execution traces) visualizations, and therefore is not usable for the evaluation of our approach, which relies solely on static information. The authors themselves acknowledged the fact that none of their tasks can be solved in the absence of dynamic information.

7.2.5 Guidelines for Software Visualization Evaluation

Kitchenham et al. [KPP$^+$02] proposed a set of guidelines for designing, conducting, and evaluating empirical research in the more general context of software engineering. Some of these are applicable to empirical research in software visualization, in particular the ones related to the presentation of the results. An observation mentioned in this work is that in a validation experiment one can compare two defined technologies, one against the other, but "it is usually not valid to compare using a technology with not using it". Although this sounds like a reasonable observation, we found this anti-pattern in the designs of several of the controlled experiments for the evaluation of software visualization approaches discussed later.

One of the problems in designing and performing evaluations of software visualization approaches is the lack of software visualization benchmarks. Maletic and Marcus acknowledged this problem and launched a call for such contributions, to raise the awareness of the scientific community [MM03].

Di Penta et al. synthesized a set of guidelines for designing empirical studies in the field of program comprehension [PSK07]. Some of the pitfalls of breaking these guidelines are severe, such as data that fails to support even true hypotheses, or conclusions that are not statistically significant due to insufficient data points. A major concern raised by the authors was the replicability of the empirical studies. They proposed a "recipe" for presenting the results and making materials available to facilitate replication and evaluation. We used this recipe to present our controlled experiment in a replicable way.

After performing several experiments for the evaluation of visualization approaches, Sensalire et al. [SOT09] share a number of lessons learned during the process. One of these lesson refers to the risk of involving participants covering a wide range of experience levels, which could bias the results of the study. To address this issue in our experiment, we use blocking based on the experience level, which allows us to perform separate analyses on the different blocks. Following the authors' advice against exposing the participants to the tool for just a few minutes before the experiment, we planned to perform a session to present our approach before each experimental run.

With respect to tasks, Sensalire et al. make a distinction between tasks aiming at program discovery and tasks aiming at program maintenance, and admit that in case of the former, it is harder to quantify the effectiveness of the tool. CodeCity is a tool that supports mainly program discovery and only indirectly maintenance. We demonstrated its usefulness in finding unexpected facts only by means of our case studies. However, testing its effectiveness in performing precise tasks can give a complementary measure of its practical value. The authors suggested that professionals are interested in tools supporting maintenance, rather than program discovery. The positive feedback we received on CodeCity support our somewhat different viewpoint: Lower-level maintenance tools and higher-level analysis tools (e.g., visualizations) are not reciprocally exclusive, but rather complementary in supporting the understanding of today's software systems. Some of the more experienced industry practitioners that participated in our experiment or attended a presentation specifically mentioned the lack and need of overview tools, such as CodeCity.

Another concern raised by the authors of this work relates to the motivation of participants, in particular professionals, who may require a measurable return to invest time in learning to use a tool. To this end, we precede each experimental session with a presentation session. The opening presentation session includes a description of the approach and a tool demonstration, which provide motivation for both professional interested in new tools, and academics active in software visualization, reverse engineering, or program comprehension.

7.2.6 Empirical Evaluation in Software Visualization

Koschke states that the lack of proper evaluation to demonstrate the effectiveness of tools is detrimental to the development in the field [Kos03]. Consequently, there is a growing request for empirical evaluations in the software visualization field.

Storey and Müller count among the first researchers to have performed empirical evaluations for their visualization tools (i.e., SHriMP and Rigi, respectively) by means of controlled experiments. In a first step, the authors drafted a controlled experiment for the evaluation of reverse engineering tools, and reported on preliminary results obtained from a pilot study [SMW96]. In a second step, Storey et al. performed the actual experiment [SWM97], in which they compared their two tools to a baseline (i.e., SNIFF+). Based on the experiment, performed with 30 students, of which 5 graduates and 25 undergrads, the authors compared the support provided by their tools in solving a number of program comprehension tasks. The authors focused on identifying both the types of tasks that are best supported by their tools and the tools' limitations, which is also one of the goals of our controlled experiment. However, the tasks of their user study are more oriented towards code change and lower-level comprehension, while the aim of our approach and therefore, of the tasks in our experiment, is on higher-level analyses and overall comprehension of the system's structure.

Apart from the positive lessons we could extract from this work, we identified a couple of issues with this user study. First, in spite of the practical nature of the tasks (i.e., maintenance and program understanding), the subjects were exclusively students and therefore might not have been a representative sample for the tasks' target population, namely industry practitioners. Second, the two experimental treatments required a decomposition of the object system manually built by the authors of the tools (i.e., a sub-system hierarchy, based on the modularization of the source code into files), which turned out to be a key factor on the outcome of these groups. Although semi-automatic approaches are common in program comprehension, this intervention may have influenced the results of the experiment.

Marcus et al. [MCS05] described a study aimed at testing the support provided by their tool called sv3D in answering a set of program comprehension questions. To this purpose, the authors compared the performances obtained by using their tool to the ones obtained by exploring a text file containing all the metrics data and of source code in an IDE. We consider this to be a more spartan version of our choice for a baseline, i.e., we provided a spreadsheet with all the metric data, which is structured and allows advanced operations, such as sorting or filtering. The questions that the subjects were supposed to address mostly related to the metrics represented by the tool (i.e., number lines of text, number of lines of comments, complexity measures, number of control structures).

The object system of their experiment was a small Java application of only 27 classes and 42 kLOC, which is not a representative size for typical software systems. Moreover, the fact that all the participants (i.e., 24 in the experiment and 12 in the pilot study) were students raises the threat of representativeness of the subject sample. In the pilot, the authors performed the training session just before the test, while for the experiment they decided to schedule the training session few days prior to the test. The authors developed additional software to capture statistics, such as the amount of time needed to answer a question or the number of times a participant changed an answer. A surprising result of the experiment was that from the viewpoint of completion time, the text group performed better than the visualization group. From the accuracy point of view, the experimental group performed slightly better, but the difference was not statistically significant. An important lesson shared by the authors is that their subjects would have required several hours of training to get to use the tool in a similar manner as the authors themselves.

One fundamental threat to internal validity we see in the design of this experiment is the fact that the authors changed too many elements (i.e., level of experience of the subjects, the amount of time passed between the training and the test) between the two phases of the experiment and thus were not able to determine which of these confounding factors was the real cause of the difference between the results of the two runs.

Arisholm et al. [ABHL06] performed an experiment to validate the impact of UML documentation on software maintenance. Although documentation does not have much in common with interactive visualization— and yet, so many people consider ULM as visualization, rather than a visual language—there is a number of interesting insights about the design of evaluation experiments and the presentation of the results. The independent variable was the use of UML, which goes against one of the guidelines of Kitchenham et al. [KPP$^+$02], because it compares using a technology to not using it. Moreover, providing the experimental group with the same toolset as the control group, in addition to the new tool, opened the possibility for the subjects in the experimental group to use only the baseline tools, a fact the authors found out from the debriefing interviews. Apart from the questionable validity of such a comparison, the presence of this confounding factor is another reason to avoid such a design. The two objects of this experiment were very small systems: a simple ATM system of 7 classes and 338 LOC and a software system controlling a vending machine with 12 classes and 293 LOC. The UML documents provided were a use case diagram, sequence diagrams, and a class diagram.

Although the authors were mainly interested in demonstrating the usefulness of UML documentation in practice, the size of the two object systems is not comparable with the size of software systems in industry and the few UML diagrams used in the experiment do not reflect the huge amount of UML diagrams present in a system documented using this modeling language. We claim that anything that is demonstrated under such artificial conditions can hardly be generalized for a real setting, such as an industrial context. Moreover, all 98 subjects of the experiment were students, which is another external threat to validity.

A positive characteristic of this experiment's design was the realism of the tasks, reflected also by the significant amount of time required for an experiment run (8–12 hours). The authors used blocking to ensure comparable skills across the two student groups corresponding to the two treatments. We also use blocking in our experiment, not only based on the experience level, but also on the background (i.e., industry or academia), since we had a large share of industry practitioners. The experiment of Arisholm et al. took place on two sites, i.e., Oslo (Norway) and Ottawa (Ontario, Canada). In a similar vein, our experiment had eleven runs over four sites in three different countries. For the analysis, Arisholm et al. considered each task separately, since different results were observed due to the variation in complexity. For the same reason, we also consider each task separately, complementary to the overall results. The authors concluded that although in terms of time UML documentation did not seem to provide an advantage when considering the additional time needed to modify models, in terms of correctness, for the most complex tasks, the subjects who used UML documentation performed significantly better than those who did not.

Lange et al. presented the results of a controlled experiment in which they evaluated the usefulness of four enriched UML views they have devised, by comparing them with traditional UML diagrams [LC07]. The experiment was conducted over two runs, in which the second was a replication of the first. There were 29 multiple-choice questions divided in four categories. The subjects of this experiment, conducted within a course on software architecture, were 100 master students unaware of the goal and research questions of the experiment. The baseline of the experiment was composed of a UML tool and a metric analysis tool. Similarly to this approach, we compose a baseline from several tools in order to cover the part of our tool's functionality that we were able to evaluate. Probably due to the lack of scalability of UML in general, the size of the object systems in this experiment was rather modest (i.e., 40 classes) compared to our object systems, which are up to two orders of magnitude larger (i.e., 4,656 classes in the case of Azureus). The measured dependent variables in the experiment of Lange et al. were the total time and the correctness, which is defined as the ratio between the number of correct answers and the total number of questions. This form of recall allows direct comparison in terms of correctness between the experiment and its future replications, even if the number of questions varies.

For our experiment, we considered that measuring the total time alone was a rather imprecise measure of performance, given the variety in difficulty of the tasks, and therefore we decided to complement the overall time analysis with a task-by-task time analysis.

Quante performed a controlled experiment for the evaluation of his dynamic object process graphs in supporting program understanding [Qua08]. The experiment had 25 computer science students as subjects, a homogeneous composition lacking any representation from industry. An interesting choice of the author was the use of not one, but two object systems. The tool was introduced using slides and an experimenter's handbook, followed by a set of training tasks for both the experimental and the control group, of which the first half performed in parallel with the experimenter. For each of the two systems, the author devised three tasks and allowed the participants to take as much time as needed to finish each task, to avoid placing any time pressure on the subjects. The participants were not told how many tasks there were, yet after two hours, they were stopped. The lack of time constraints led to several participants using the entire allotted time in solving the first task. For this reason, only the first task for each object system had complete data.

A very interesting outcome of this experiment is the fact that the two object systems led to significantly different results. The improvement of the experimental group could only be detected in the case of one of the object systems, while in the case of the other, the performances of the participants were not significantly better. This work gave us the valuable insight that relying on solely one subject system is unsound. Another lesson we learned from Quante's work is that an experiment that failed with respect to the expected results is *not* necessarily a failure.

Knodel et al. presented the results of a controlled experiment for the evaluation of the role of graphical elements in visualization of software architecture [KMN08]. In a preliminary step, the authors verified the soundness of the tasks with the help of two experts in the object system (i.e., Tomcat). The participants of the experiment were 12 experienced researchers and 17 students from Fraunhofer in Kaiserslautern (Germany) and Maryland (United States). The tested hypotheses were either about the impact of the configuration of the visualization on the results of different types of tasks, or about the difference in performance between experienced researchers and students in solving the tasks. In our experiment, we use blocking based on the experience level to identify the type of user best supported by our approach. Interestingly, the authors asked for results in the form of both written answers and screenshots created with the tool. From the debriefing questionnaire the authors found out that the configuration of the visualization does make a difference when solving tasks and that an "optimal configuration" does not exist, because the results depend on both the user and the task. Moreover, they were able to identify the more efficient of the two configurations they tested. Knodel et al. consider configurability to be a key requirement and recommend the visualization tool developers to invest effort into it — a point we fully support.

Cornelissen et al. [CZRvD09, CZvDVR09] performed a controlled experiment for the evaluation of EXTRAVIS, an execution trace visualization tool. The experiment consisted of solving four tasks, divided into a total of eight sub-tasks, which—as the authors claimed—cover all the activities in Pacione's model [PRW04]. The choice of the model fits the nature of their approach, i.e., the analysis of dynamic data. The purpose of the experiment was to evaluate how the availability of EXTRAVIS influences the correctness and the time spent by the participants in solving the tasks. The subject population was composed of 23 participants from academia and only one participant from industry, which the authors claimed to mitigate the concern of unrepresentative population. However, as the authors themselves admit in the discussion on the threats to validity, one single subject from industry cannot generate any statistically relevant insights that holds for industry practitioners in general.

We drew inspiration from this experiment in some of the organizational aspects, such as the pre-experiment questionnaire (i.e., a self-assessment of the participant candidates on a set of fields of expertise) or the debriefing questionnaire (i.e., a set of questions related to the difficulty and the time pressure experienced while solving the tasks). We also learned that training sessions of 10 minutes are probably too short, something acknowledged even by some of their participants. The authors designed the treatments as follows: The control group gets Eclipse, while the experimental group gets Eclipse, EXTRAVIS, and the execution traces.

We found two issues with this design. First, the two groups do not benefit from the same data, since only the experimental group has the execution traces. Under these circumstances, it is not clear whether the observed effect is owed to the availability of the data, of the tool, or of both. Second, in addition to the evaluated tool (i.e., EXTRAVIS), the experimental group also had the tool of the control group, which goes back to the problem signaled by Kitchenham et al., who question the validity of comparing using a technology with not using it [KPP^{+}02]. Nevertheless, the work has inspired us from many points of view, such as organization, the questionnaire, or the amount of details in which they presented the experiment's design and procedure, which makes it replicable.

7.3 Wish List Extracted from the Literature

The literature survey we conducted allowed us to build the following list of desiderata for our experiment, extracted from both the strengths and the weaknesses of the current body of research:

1. **Avoid comparing using a technique against not using it.** Although in their guidelines for empirical research in software engineering Kitchenham et al. characterized this practice as invalid, many of the recent controlled experiments are based on such a design, which tends to become an anti-pattern. To be able to perform a head-to-head comparison with a reasonable baseline, we invested effort into finding a good combination of state-of-the-practice tools to compare our approach to.

2. **Involve participants from industry**. Our approach, which we devised to support practitioners in analyzing their software systems, should be evaluated by a subject population that includes a fair share of software practitioners. Moreover, professionals are less likely to provide a positive evaluation of the tool if it does not actually support them in solving their tasks [SOT09]. Unfortunately, the literature study we performed showed that most evaluations of software visualization approaches have been performed with academics, in particular students.

3. **Provide a not-so-short tutorial of the experimental tool to the participants**. It is important for the participants to choose an appropriate visualization and to translate the tasks into actions upon the visualization. On the one hand, for a fair comparison of a new tool with the state-of-the-practice, the experimental group would require many hours of intensive training, to even come close to the skills of the control group acquired in years of operation. On the other hand, the most an experimenter can hope for from any participant, in particular from professionals in both research and industry, is a very limited amount of time. Therefore, the experimenter needs to design an interesting yet concise tutorial which is broad enough to cover all the features required by the experiment, yet is not limited to solely those features.

4. **Avoid, whenever possible, to give the tutorial right before the test.** One of the lessons learned from the experiment of Marcus et al. [MCS05] is that allowing the subjects to get in contact with the tool in advance (i.e., performing the training a few days before the test) is quite useful. Although we tried to give the tutorial in advance, sometimes this was just impossible due mostly to the limited amount of time we could afford to get from our subjects. To compensate for this, we provided a set of online video tutorials that the subjects could consult in advance.

5. **Use the tutorial to cover both the research behind the approach and the implementation.** Different types of subjects have different incentives to participate in the experiment. Practitioners are probably more interested in what the tool is able to support them with, while academics are more likely to be interested in the research behind the tool. If the experiment is designed to have both categories of subjects, dividing the tutorial in two distinctive parts can be helpful. Each of our experimental sessions is preceded by a presentation and tool demonstration of about 45 minutes.

6. **Find a set of relevant tasks**. In task-based evaluations, the relevance of the results depends directly on the relevance of the tasks with respect to the purpose of the approach. In the absence of a definitive set of typical higher-level software comprehension tasks in the literature, we naturally look for the tasks among the ones that we had in mind when devising our approach. However, for objectivity, we placed the tasks in the context of their rationale and of their target users. Moreover, we avoided very basic or trivial tasks and chose tasks close in complexity to the real tasks performed by practitioners. This alleviates the concern about performing the study in an artificially simple environment, raised by Kosara et al. [KHI$^+$03].

7. **Choose real object systems that are relevant for the tasks.** Many of the experiments in the literature used very small systems as objects and therefore, led to results that cannot be generalized for the case of real systems, such as the ones in industry. Since with our research we aim at supporting the understanding and analysis of medium-to-large software systems, for our experiment we consider only real systems of relevant size, a decision that goes along the guidelines of Plaisant et al. [Pla04].

8. **Include more than one subject system in the experimental design**. The experiment of Quante [Qua08] showed that performing the same experiment on two different systems can lead to significantly different results. Therefore, in our experiment, we consider two systems, different in both scale and application domain.

9. **Provide the same data to all participants**. No matter which groups the participants belong to, they should have access to the same data. Thus, the observed effect of the experiment is more likely to be due to the independent variables.

10. **Limit the amount of time allowed for solving each task**. Allowing unbounded time for a task to avoid time pressure may lead to participants spending the entire allotted time for the experiment solving a single task. Moreover, in an open-ended task setup, a long time does not necessarily reflect the difficulty of the task, but also the fun one has solving it. Our solution for this was to provide a maximum time per task and to check with an expert for each of the tools whether the time window is feasible for the task.

11. **Provide all the details needed to make the experiment replicable**. We followed the guidelines of Di Penta et al. [PSK07] and made available the materials and results to facilitate its evaluation and replication:

    - subject selection criteria and justification
    - subject screening materials and results, with subject names replaced by identifiers
    - pre-test questions, and results keyed to the unique subject identifiers, as well as explanation of the skills that the questions are designed to evaluate

- control and treatment groups (i.e., sets of subject identifiers)
- post-test design and control/treatment group materials, as well as an explanation of the knowledge the post-test questions are designed to evaluate
- if different instructions are given to the control and treatment groups, some summary of the contents of these instructions

12. **Report results on individual tasks**. A precise identification of the types of tasks that mostly benefit from the evaluated tool or approach allows a more informed decision for potential adopters. Moreover, due to the variation in complexity, differences in time performances from one task to another are expected [ABHL06].

13. **Include tasks on which the expected result is not always to the advantage of the tool being evaluated.** This allows the experimenter to actually learn something during the experiment, including shortcomings of the approach. However, given the limited amount of time—and thus tasks—participants have, these tasks should be a minority with respect to the tasks for which superiority from the evaluated tool is expected.

14. **Take into account the possible wide range of experience level of the participants.** To allow an analysis based on the experience level which is supposed to influence the participants' performance in solving the given tasks [Pet95, KMN08], we use blocking, which implies dividing the subjects of each treatment into blocks based on their experience and skills.

Guided by this list, we conceived the design of our controlled experiment, described next.

7.4 Experimental Design

The purpose of the experiment is to provide a quantitative evaluation of the effectiveness and efficiency of our approach when compared to state-of-the-practice exploration approaches.

7.4.1 Research Questions & Hypotheses

The research questions underlying our experiment are:

Q1 : Does the use of CodeCity increase the *correctness* of the solutions to program comprehension tasks, compared to non-visual exploration tools, regardless of the object system size?

Q2 : Does the use of CodeCity reduce the *time* needed to solve program comprehension tasks, compared to non-visual exploration tools, regardless of the object system size?

Q3 : Which are the *task types* for which using CodeCity makes a difference in either *correctness* or *completion time* over non-visual exploration tools?

Q4 : Do the potential benefits of using CodeCity in terms of *correctness* and *time* depend on the user's *background* (i.e., academic versus industry practitioner)?

Q5 : Do the potential benefits of using CodeCity in terms of *correctness* and *time* depend on the user's *experience* level (i.e., novice versus advanced)?

The null and alternative hypotheses corresponding to the research questions *Q1* and *Q2* are synthesized in Table 7.1. The remaining three questions, although secondary, allow us to search for more precise insights about our approach. To address the question *Q3*, we perform a separate analysis of correctness and completion time for each of the tasks, while for the questions *Q4* and *Q5* we perform an analysis of the data within blocks.

Null hypothesis		Alternative hypothesis	
$H1_0$:	The *tool* does not impact the correctness of the solutions to program comprehension tasks.	$H1$:	The *tool* impacts the correctness of the solutions to program comprehension tasks.
$H2_0$:	The *tool* does not impact the time required to complete program comprehension tasks.	$H2$:	The *tool* impacts the time required to complete program comprehension tasks.

Table 7.1. Null and alternative hypotheses

7.4.2 Dependent & Independent Variables

Similarly to other empirical evaluations of software visualization approaches [LC07, Qua08, CZRvD09], the dependent variables of our experiment are the *correctness* of the task solution and the *completion time*. While the correctness of the task solutions is a measure of the effectiveness of the approach, the completion time represents a measure of the efficiency of the approach.

The purpose of the experiment is to show whether CodeCity's 3D visualizations provide better support to software practitioners in solving program comprehension tasks than state-of-the-practice non-visual exploration tools. Additionally, we want to see how well CodeCity performs compared to the baseline when analyzing systems of different magnitudes, given that one of the goals of our approach was to provide support in analyzing large-scale systems.

Hence, our experiment has two independent variables: the *tool* used to solve the tasks and *object system size*. The tool variable has two treatments, i.e., CodeCity and a baseline, while the object system size has two treatments, i.e., medium and large, because visualization starts to become useful only when the analyzed system has a reasonable size. The baseline and the objects systems, as well as the criteria based on which we chose them, are presented next.

Finding a Baseline. There is a subtle interdependency between the baseline and the set of tasks for the experiment. In an ideal world, we would have devised tasks for each of the three contexts in which we applied our approach: software understanding, evolution analysis, and design quality assessment. Instead, we had to find a reasonable trade-off. In our search for an appropriate baseline, we looked for two characteristics: data & feature compatibility with CodeCity and recognition from the community (i.e., a state-of-the-practice tool).

Unfortunately we could not find a single tool satisfying both criteria. The first candidate was a highly configurable text-based reverse engineering tool called MooseBrowser [NDG05], built on top of the Moose reengineering platform[1]. MooseBrowser is data-compatible with CodeCity, for it uses the same underlying meta-model for object-oriented software (i.e., FAMIX [DTD01])

[1] `http://www.moosetechnology.org`

and is able to cover the features of CodeCity we wanted to test. However, in spite of the enthusiastic Moose community, MooseBrowser is not yet state-of-the-practice in reverse engineering.

To allow a fair comparison, without having to excessively limit the task range, we opted to build a baseline from several tools. The baseline needed to provide exploration and querying functionality, support for the presenting at least the most important software metrics, support for design problems exploration, and if possible support for evolutionary analysis.

In spite of the many existing software analysis approaches, software understanding is still mainly performed at the source code level. Since the most common source code exploration tools are integrated development environments (IDEs), we chose Eclipse[2], an IDE popular in both academia and industry. The next step was finding support for exploring meta-data, such as software metrics and design problem data, since they were not available in Eclipse. We looked for a convenient Eclipse plugin for metrics or even an external tool (such as Sonar[3]) that would either include the metrics we needed for the tasks, or would provide support for entering user-defined metrics, or would even allow us to hard-code the data we had in CodeCity. Again, none of the tools we found enabled us to do so. Since we did not want to confer an unfair data advantage to the subjects in the experimental group, we chose to provide the control group with a table containing both the metrics and the design problem data, and the popular Excel spreadsheet application for exploring the data.

Finally, due to Eclipse's lack of support for multiple versions, we decided to exclude the evolution analysis from our evaluation, although we consider it one of the strong points of our approach. We felt that providing the users in the control group with several projects in Eclipse representing different versions of the same system, with no relation among them, would have been unfair.

Objects. We chose two Java systems, both large enough to potentially benefit from visualization, yet of different size, so that we can reason about this independent variable. The smaller of the two systems is FindBugs[4], a tool using static analysis to find bugs in Java code, developed as an academic project at the University of Maryland [HP04], while the larger system is Azureus[5], a popular P2P file sharing client and one of the most active open-source projects hosted at SourceForge. In Table 7.2, we present the main characteristics of the two systems related to the tasks of the experiment.

	medium	**large**
Name	FindBugs	Azureus
Lines of code	93,310	454,387
Packages	53	520
Classes	1,320	4,656
God classes	62	111
Brain classes	9	55
Data classes	67	256

Table 7.2. The object systems corresponding to the two levels of system size

[2] http://www.eclipse.org

[3] http://www.sonarsource.org

[4] http://findbugs.sourceforge.net

[5] http://azureus.sourceforge.net

7.4.3 Controlled Variables

For our controlled experiment we identified two factors that can have an influence on the performance, i.e., the background and the experience level of the participants.

The *background* represents the working context of a subject, i.e., the context in which they are currently conducting their work. The background factor has two levels: industry and academia. The background information is directly extracted from the personal information provided by the participants at the time of their enrollment. Given that some of the participants were active in both academia and industry, we considered them as practitioners.

The second factor is *experience level*, which represents the domain expertise gained by each of the participants. To keep things simple, we limited the experience level also to two levels: beginner and advanced. The level of experience of the participants is also derived from the information provided at the time of their enrollment. For the participants from academia, students (i.e., bachelor and master) are considered beginner, while researchers (i.e., PhD students, post-docs and professors) are considered advanced. For industry, we considered as beginners the participants with up to three years of experience, and as advanced the remaining ones.

We used a *randomized block design*, with *background* and *experience level* as blocking factors. We assigned each participant—based on personal information collected before the experiment—to one of the four categories (i.e., academia-beginner, academia-advanced, industry-beginner, and industry-advanced). We then randomly assigned one of the four treatments (i.e., combinations of tool and system size) to the participants in each category. The outcome of this procedure is described in Section 8.5, in the context of the subjects' analysis.

7.4.4 Tasks

Our approach, implemented in CodeCity, provides aid in comprehension tasks supporting adaptive and perfective maintenance. We considered using a previously-defined maintenance task definition framework to design the tasks of our evaluation. However, the existing framework proved ill-suited. Due to the fact that CodeCity relies exclusively on static information extracted from the source code, it was not realistic to map our tasks over the model of Pacione et al. [PRW04], which is biased towards dynamic information visualization. On the other hand, the set of questions asked by developers, synthesized by Sillito et al. [SMDV06], although partially compatible with our tasks refers to developers exploring source code only. Our approach supports software architects, designers, quality-assurance engineers, and project managers, in addition to developers. These additional roles assess software systems at higher levels of abstraction not covered by the the framework proposed by Sillito et al.

In spite of the lack of frameworks and task models for higher-level assessments of software systems, we describe each task in terms of its concern and rationale, which illustrate operation scenarios and identify the targeted user types. The questions in the tasks were designed to fit in one of the following categories: structural understanding, concept location, impact analysis, metric-based analysis, and design problem assessment.

The questionnaires corresponding to the four treatments are specific to each combination of toolset and object system, but conceptually equal. In the following, we present the conceptual set of tasks, while in Section A.1 we include the full questionnaire with all the variations corresponding to the four treatments. In the handed questionnaires, apart from the tasks themselves, we included spots for the participants to log the begin and end times, as well as the split times between each two consecutive tasks.

The task set is split in two parts, i.e., part *A* concerned with program comprehension and part *B* concerned with the design quality assessment.

A1 **Task.** Locate all the unit test classes of the system and identify the convention (or lack of convention) used by the system's developers to organize the unit tests.
Concern. Structural understanding.
Rationale. Unit testing is a fundamental part of quality software development. For object-oriented systems, the unit tests are defined in test classes. Typically, the test classes are defined in packages according to a project-specific convention. Before integrating their work (which ideally includes unit tests) in the structure of the system, *developers* need to understand how the test classes are organized. Software *architects* design the high-level structure of the system (which may include the convention by which test classes are organized), while *quality assurance engineers* monitor the consistency of applying these rules throughout the evolution of the system.

A2.1 **Task.** Look for the term *T1* in the names of the classes and their attributes and methods, and describe the spread of these classes in the system.
Concern. Concept Location.
Rationale. Assessing how the domain knowledge is encapsulated in the source code is important for several practitioner roles. To understand a system they are not familiar with, *developers* often start by locating familiar concepts in the source code, based on their knowledge of the application domain [HM08]. From a different perspective, *maintainers* use concept location on terms extracted from bug reports and change requests to identify the parts of the system where changes need to be performed [MRB+05]. And finally, at a higher level, software *architects* are interested in maintaining a consistent mapping between the static structure and the domain knowledge. The dispersion provides an indication of the logical modularization of a program [RMJ09]. For each of these tasks, an initial step is locating a particular term or set of terms in the system and assessing its dispersion.

A2.2 **Task.** Look for the term *T2* in the names of the classes and their attributes and methods, and describe the spread of these classes in the system.
Concern & Rationale. See task A2.1.
Note. The term *T2* was chosen such that it had a different type of spread than *T1*.

A3 **Task.** Evaluate the change impact of class *C* defined in package *P*, by considering its caller classes (classes invoking any of its methods). The assessment is done in terms of both intensity (number of potentially affected classes) and dispersion (how these classes are distributed in the package structure).
Concern. Impact Analysis.
Rationale. Impact analysis provides the means to estimate how a change to a restricted part of the system would impact the rest of the system. Although extensively used in *maintenance* activities, impact analysis may also be performed by *developers* when estimating the effort needed to perform a change. It also gives an idea of the *quality* of the system: A part of the system which requires a large effort to change may be a good candidate for refactoring.

A4.1 **Task.** Find the three classes with the highest number of methods (NOM) in the system.
Concern. Metric Analysis.

Rationale. Classes in object-oriented systems ideally encapsulate one single responsibility. Given that the method represents the class's unit of functionality, the number of methods metric is a measure of the amount of functionality of a class. Classes with an exceptionally large number of methods make good candidates for refactoring (e.g., split class), and therefore are of interest to practitioners involved in *maintenance* or *quality assurance*.

A4.2 **Task.** Find the three classes with the highest average number of lines of code per method in the system.
Concern. Metric Analysis.
Rationale. The number of lines of code (LOC) is a popular and easily accessible software metric for the size of source code artifacts (e.g., methods, classes, modules, system). Moreover, it has been shown to be one of the best metrics for fault prediction [GFS05]. A method, as a unit of functionality, should encapsulate only one function and should therefore have a reasonable size. Classes with a large ratio of lines of code per method (i.e., classes containing long and complex methods) represent candidates for refactoring (e.g., extract method), and therefore are of interest to practitioners involved in either *maintenance* activities or *quality assurance*.

B1.1 **Task.** Identify the package with the highest percentage of god classes in the system. Write down the full name of the package, the number of god classes in this package, and the total number of classes in the package.
Concern. Focused Design Problem Analysis.
Rationale. God class is a design problem first described by Riel [Rie96] to characterize classes that tend to incorporate an overly large amount of intelligence. The size and complexity of god classes makes them a maintainer's nightmare. To enable the detection of design problems in source code, Marinescu provide a formalization called detection strategies [Mar04b]. In spite of the fact that the presence alone of this design problem does not qualify the affected class as harmful [RDGM04], keeping these potentially problematic classes under control is important for the sanity of the system. We raise our analysis at the package level, because of its logical grouping role in the system. By maintaining the ratio of god classes in packages to the minimum, the *quality assurance engineer* keeps this problem at a manageable level. For a *project manager*, in the context of the software process, packages represent work units assigned to the developers. Assessing the magnitude of this problem allows him to take informed decisions in assigning resources.

B1.2 **Task.** Identify the god class containing the largest number of methods in the system.
Concern. Focused Design Problem Analysis.
Rationale. God classes are characterized by a large amount of encapsulated functionality, and thus, by a large number of methods. The fact that the result of applying the god class strategy on a class is a boolean indicating that a class is either a god class or not, makes it difficult to prioritize refactoring candidates in a list of god classes. In the absence of other criteria (such as the stability of a god class over its entire evolution), the number of methods can be used as a measure of the amount of functionality for solving this problem related to *maintenance* and *quality assurance*. For the participants of the experiment, this task is an opportunity to experience how a large amount of functionality encapsulated in a class is often related to the god class design problem.

B2.1 **Task.** Identify the dominant class-level design problem (the design problem that affects the largest number of classes) in the system.

Concern. Holistic Design Problem Analysis.
Rationale. God class is only one of the design problems that can affect a class. A similar design problem is the brain class, which accumulates an excessive amount of intelligence, usually in the form of brain methods (i.e., methods that tend to centralize the intelligence of their containing class). Finally, data classes are just "dumb" data holders without complex functionality, but with other classes strongly relying on them. Gaining a "big picture" of the design problems in the system would benefit *maintainers*, *quality assurance engineers*, and *project managers*.

B2.2 **Task.** Write an overview of the class-level design problems in the system. Are the design problems affecting many of the classes? Are the different design problems affecting the system in an equal measure? Are there packages of the system affected exclusively by only one design problem? Are there packages entirely unaffected by any design problem? Or packages with all classes affected? Describe your most interesting or unexpected observations about the design problems.
Concern. Holistic Design Problem Analysis.
Rationale. The rationale and targeted user roles are the same as for task B2.1. However, while the previous one gives an overview of design problems in figures, this task provides qualitative details and has the potential to reveal the types of additional insights obtained with visualization over raw data.

Our set of tasks maps partially on the program comprehension framework described by Pacione et al. [PRW04], which consists of 9 activities and 14 tasks (of which 8 general comprehension tasks and 6 specific reverse engineering tasks). We found this framework to be biased towards dynamic analysis: four of the activities (A1, A5, A6, and A7) and six of the tasks (G2, G5, G6, S1, S2, and S6) rely heavily or completely on information gathered at runtime. Moreover, the authors themselves acknowledge the fact that the activities cannot be performed without dynamic information.

In spite of this bias, it is interesting to see the extent to which these activities are covered by our task set (See Table 7.3).

ID	Activity Description	Compatible tasks (IDs)
A1	Investigating the functionality of the system	-
A2	Adding to or changing the system's functionality	A1
A3	Investigating the internal structure of an artifact	A2.1, A2.2
A4	Investigating dependencies between artifacts	A3
A5	Investigating runtime interactions in the system	-
A6	Investigating how much an artifact is used	-
A7	Investigating patterns in the system's execution	-
A8	Assessing the quality of the system's design	A4.1, A4.2, B1.1, B1.2, B2.1, B2.2
A9	Understanding the domain of the system	A2.1, A2.2

Table 7.3. The relation between our tasks and the activities defined by Pacione et al.

7.4.5 Treatments

By combining the two levels of each of the two independent variables we obtain four treatment combinations, illustrated in Table 7.4.

	Azureus	**FindBugs**
CodeCity	T1	T2
Ecl+Excl	T3	T4

Table 7.4. Independent variables and the resulting treatment combinations

We provided the treatments as virtual images for VirtualBox[6], which was the only piece of software required to be installed by each participant. Each virtual image contained only the necessary pieces of software (i.e., either CodeCity or Eclipse+Excel), installed on a Windows XP operating system with Service Pack 2 (SP2) installed.

The two images corresponding to the experimental groups (i.e., T1 and T2) contained:

1. an installation of CodeCity,
2. the FAMIX model of the object system loaded in CodeCity, and
3. the source code of the object system, directly accessible from the visualizations (i.e., CodeCity allows the user to view the source code of any visualized class).

The two images corresponding to the control groups (i.e., T3 and T4) contained:

1. an Eclipse installation with all default development tools,
2. an Eclipse workspace containing the entire source code of the object system in one compilable Eclipse project, and
3. an Excel installation and a sheet containing all the metrics and design problem data required for solving the tasks and available to the experimental groups.

The design of our experiment is a between-subjects design, i.e., a subject is part of either the control group or of the experimental group.

7.5 Summary

The design of the experiment is a key factor to the success of the experiment. Therefore we carefully designed our experiment, taking into account a list of guidelines we built based on the strengths and weaknesses we encountered in the related work.

We believe that our experimental design, although far from perfect, represents a good trade-off between objectivity, relevance, and attainability.

The contribution of this reliable design to the overall success of the experiment is twofold, as demonstrated in the next chapter. On the one hand, the elaborate design allowed us to conduct the operation phase of the experiment with high confidence. On the other hand, the design permitted us to collect rich data, which enabled us to perform a broad range of analyses which led to a number of promising results.

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

Chapter 8

Experimental Operation and Results

8.1 Introduction

Apart from the experimental design, a key factor that contributes to the success of a controlled experiment is the operation, which boils down to the struggle of the experimenters to conduct the experiment along the lines of the experimental design, in the presence of unexpected perturbations. While the design phase is an iterative, error-tolerant process, the operation phase is in general unrepeatable, because it "uses up" the subjects—the most expensive resource of an experiment—by exposing them to the treatments. Therefore, the experimental operation is a critical phase which requires a thorough preparation, because in a controlled experiment, as in real life—to paraphrase Murphy's law—anything that can go wrong, *will* go wrong.

8.2 Operation

Our experiment's operation covered a time span of six months, between Nov 2009 and Apr 2010, and can be divided in two phases, i.e., the pilot phase and the experiment phase, each consisting of a series of experimental runs.

An experimental run consists of a presentation session of about one hour, followed by one or more experimental sessions of up to two hours each. A presentation session consists of a talk in which the experimenter presents our approach, concluded with a CodeCity tool demonstration. During the experimental sessions following the presentation session, the subjects solve the tasks with the assigned tool on the assigned object system, under the experimenter's observation.

Although in an ideal world, the experiment would shortly follow the presentation, due to logistical reasons (i.e., it is already very difficult to be granted four hours of someone's spare time, let alone four consecutive hours), these two phases of the experimental session were separated by time windows whose lengths range from 20 minutes to four weeks.

Figure 8.1 shows the timeline of our experiment. Apart from the dates and locations of the different experiment runs, the timeline also shows the succession between presentation sessions and experimental sessions and the distribution of experimental and control units in the experimental session. The numbers in the figure reflect only the participants whose data points were taken into account during the analysis and do not include the four exceptional conditions excluded from the analysis, as explained in Section 8.4.2.

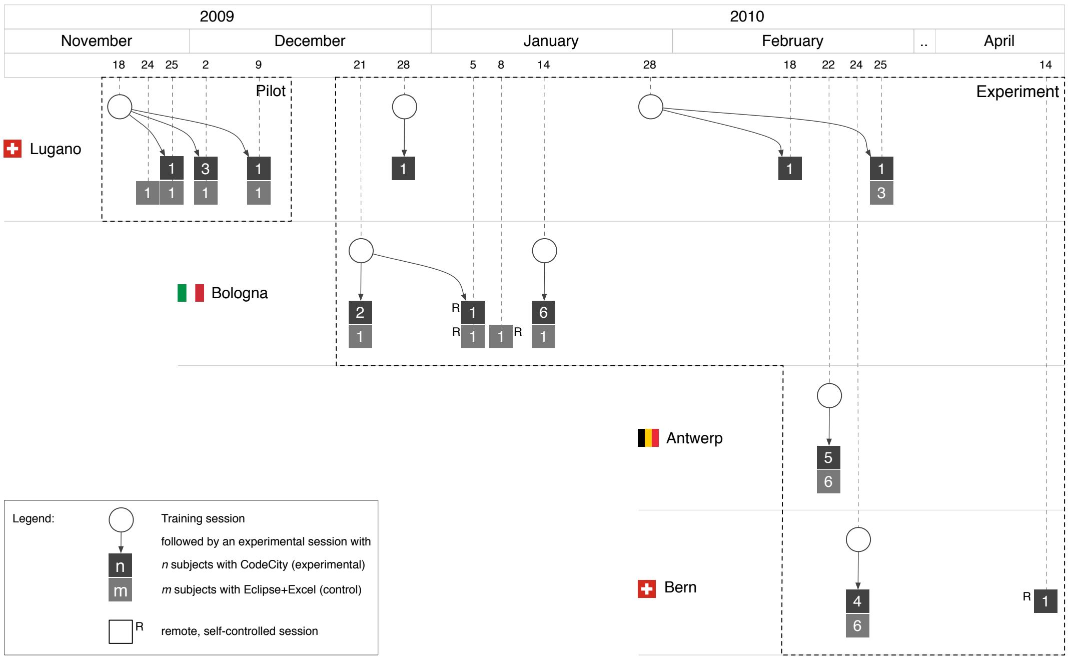


Figure 8.1. The timeline of the experiment

8.2.1 The Pilot Study

Before taking our experiment design to industry practitioners, we wanted to make it reliable, if not foolproof. With this goal in mind we designed a pilot study with the Master students of the University of Lugano (Switzerland) enrolled in a course of Software Design and Evolution. Improving the questionnaire and solving problems as they emerged required several iterations. Since we wanted to make the most of our resources, of which the most important one consisted of the participants, we assigned only two participants per experiment session.

The study was conducted from the 25th of November to the 9th of December 2009, in the window of four academic hours assigned weekly for the course's laboratory work. In the first lab session, we presented the approach and gave a tool demonstration, followed in the next three weeks by experimental sessions. Before the first of these sessions, we conducted the experiment with a Ph.D. student from our group, who has extensive experience with Eclipse, to make sure the tasks for the control group are doable in the allotted time. During these three weeks we managed to obtain a stable form for the questionnaires, to incrementally fix the encountered problems, and to come up with a reliable and scalable timing solution (i.e., before creating the timing web application, we used third-party timing software, which did not give enough configurability and scalability).

Unfortunately, although the design of this study was exactly the same with the one of the experimental phase, we could not include these data points in our analysis, due to the changes in the questionnaire's form, which made the first drafts incompatible with the final version.

8.2.2 The Experimental Runs

At this point in time, we were confident enough to start our experiment. We had the luck to benefit from the industry contacts of one of the members of our research group, who acted as a mediator between us and two groups of industry practitioners from Bologna (Italy). Each of these groups were meeting regularly to discuss various technology-related topics. Many of the practitioners of these two groups were quite enthusiastic about our invitation to attend a CodeCity presentation and volunteered to participate in our experiment.

Bologna I. The first group was composed of practitioners working for several companies from and around Bologna, including Cineca[1], an Italian consortium between several large Italian universities (i.e., Bologna, Florence, Padova, and Venice), founded to support the transfer of technology from academical research to industry. The subjects of this run were eight practitioners (i.e., developers, software engineers, and software architects) with 4–10 years of experience in object-oriented software development.

During this experimental run, conducted on the Dec 21 2009, we encountered the first scalability challenges, with its eight subjects and two experimenters. First, due to some OS-related issues of the virtualization software, three of the participants could not import the virtual images containing their treatments. We provided our only extra machine to one of them, but unfortunately this subject eventually gave up the experiment, due to fatigue and the late time (i.e., the experimental session started at 10 PM). The two remaining subjects offered to perform the experiment remotely and to send us the results later. Given the reliability of the persons and the value of their data points (i.e., one of them was probably the most experienced of the group), we were happy to accept their offer, despite the lack of “control”. In the end, we got the data points from these experiment runs performed remotely. Moreover, the more experienced practitioner performed the experiment two times, once with CodeCity and once with Ecl+Excl, but every time with a different object system, to avoid any learning effect on the participant. Of the five subjects that performed the experiment in controlled conditions, we had to exclude two from the analysis because of the reasons detailed in Section 8.4.2. In spite of all these, the practitioners reacted quite positively, found CodeCity useful, and were looking forward to use it on their own systems.

Bologna II. The second group of software practitioners we conducted our experiment with was part of the eXtreme Programming User Group (XPUG) in Bologna[2]. This rather heterogeneous group included eight practitioners covering a wide range of roles at their working places (i.e., developer, consultant, architect, trainer, project manager, system manager/analyst, CTO) and one student. The practitioners had 7–20 years of experience in object-oriented programming and up to 10 years in Java programming. During this run, performed in the evening of Jan 14 2010, an interesting event took place.

After the presentation, almost the entire audience remained for the experiment, including not only the enrolled volunteers we were counting on, but also other practitioners who wished to assist as spectators. To our surprise, very likely in the vein of their group meetings, these spectators soon formed small groups around a couple of the subjects, mostly working with CodeCity. Although this unplanned situation was not part of the design of our controlled experiment (i.e., where the unit of the experiment was the individual), we did not wish to intervene and break the

[1] http://www.cineca.it

[2] http://bo-xpug.homeip.net

ad-hoc dynamics of the group. Instead, we chose to observe these enthusiastic clusters performing pair-analysis and enjoying every moment, which was one of the most gratifying experiences throughout the experiment.

Lugano I. In between the two experiment sessions in Bologna, we received in Lugano the visit of a fellow post-doctoral researcher from Bern (Switzerland), who is also development leader in a successful small company, and we performed an experiment session with him.

Lugano II & III. The third group of industry practitioners we approached was the Java User Group (JUG) in Lugano[3]. First, we gave a presentation at the end of January 2010 and made contact with potential participants. Later, we performed two experimental runs (i.e., on the 18th and the 25th of January 2010, respectively) with five practitioners in total, all Java experts with 10 or more years of experience in both object-oriented and Java programming, occupying various positions (i.e., architect, head of IT, developer, project manager).

In the week between the two experiment sessions in Lugano, we performed a tour-de-force with stops in Antwerp (Belgium) and Bern (Switzerland).

Antwerp. First, we went to Antwerp, where we were hosted by Prof. Serge Demeyer and his research group LORE[4]. We performed the experiment session during the Software Reengineering course with both Master students enrolled in the course and Ph.D. students from our hosting research group.

The first problem we had to deal with was that the low amount of RAM memory on the workstations would not allow us to run the virtual machines. To solve this problem, during the presentation session, one of our hosts copied the content of the virtual machines directly on the workstations hard drives, which allowed running the tools on the host operating system of the workstations. Later on, some of our subject signaled us another problem, this time with the spreadsheet. While the data in the spreadsheet has been entered with a '.' separator for decimals, in Belgium the correct separator is ','. Due to this incompatibility, some of the numeric data was by default considered string and interfered with the sorting operations. The problem was solved by the participants either by modifying the regional settings in their operating system or by performing a search and replace operation. Most of the participants of this experimental run were very well prepared with operating CodeCity, which showed that they have studied the video tutorials in advance.

Bern. Only two days after Antwerp, we went to Bern, where we were hosted by Prof. Oscar Nierstrasz and his research group SCG[5]. We performed the experiment session with mostly Ph.D. students and a couple of Master students from the research group. During this experimental run we did not encounter any problem.

Some of the participants had already seen CodeCity earlier, given that this was the research group that built Moose, the reverse-engineering platform underlying CodeCity. With this occasion, we asked the main developer behind Moose, who is currently working as a consultant and who was not available for that afternoon, to perform the experiment remotely and to send us the result. With this last data point, received three weeks later, we concluded our data collection.

[3] http://www.juglugano.ch

[4] http://lore.ua.ac.be

[5] http://scg.unibe.ch

8.3 Data Collection and Marking

Using different mechanisms, we collected several types of information at different moments in time: before, during the experiment, and after the experiment. We used blind marking for grading the correctness of the participants' solutions.

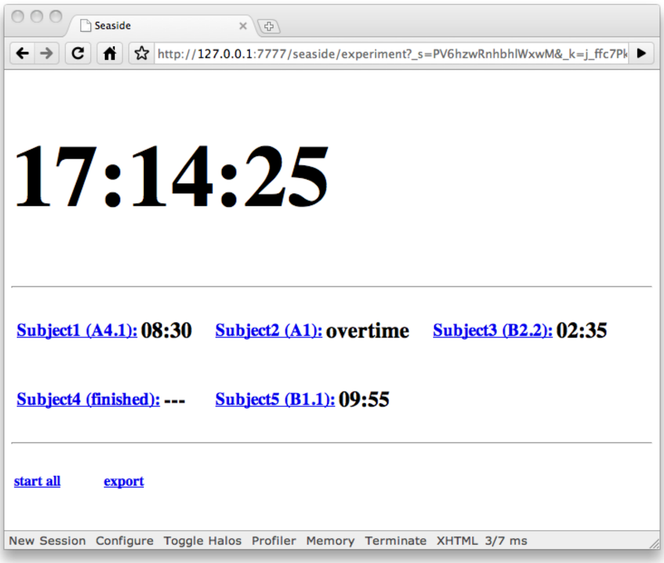
8.3.1 Personal Information

Before the experiment, we collected both personal information (e.g., gender, age, nationality) used for statistics and professional information (e.g., current job position and experience levels in a set of four skills identified as important for the experiment) used for blocking, by means of an online questionnaire presented in Section A.1. The collected data allowed us to know at all times the number of participants that we can rely on and to plan our experimental runs.

8.3.2 Timing Data

To measure the completion time, we asked the participants to log the start time, split times, and end time. However, we learned that this measure alone did not represent a reliable solution, on an occasion in which several participants, excited by the upcoming task, simply forgot to write down the split times. Moreover, we needed to make sure that none of them would use more than ten minutes per task, which was not something we could ask them to watch for.

To tackle this issue, we developed a timing web application in Smalltalk using the Seaside framework[6]. During the experimental sessions, the timing application would run on the experimenter's computer and project the current time (See Figure 8.2).


Seaside
http://127.0.0.1:7777/seaside/experiment?_s=PV6hzwRnhbhlWxwM&_k=j_ffc7Pk
start all
export
New Session Configure Toggle Halos Profiler Memory Terminate XHTML 3/7 ms

17:14:25

Subject1 (A4.1): 08:30 Subject2 (A1): overtime Subject3 (B2.2): 02:35

Subject4 (finished): --- Subject5 (B1.1): 09:55

Figure 8.2. The output of our timing web application

[6] http://www.seaside.st

The displayed time was used as common reference by the participants whenever they were required in the questionnaire to log the time. In addition, the application displayed the following information for each participant: the name, the current task, and the maximum remaining time for the current task.

The subjects were asked to notify the experimenter every time they log the time, so that the experimenter could reset their personal timer by clicking on the hyperlink marked with the name of the subject. Whenever a subject was unable to finish a task in the allotted time, the message "overtime" would appear beside his name, which would make the experimenter ask the subject to immediately pass to the next task, before resetting his timer.

Since most of the times the experimenter would only get to meet the subjects just before the experiment, associating names with the persons by memory was not something we wanted to rely on. Therefore, the experimenter would always bring with him a set of name tags which would be placed near the corresponding subject and would thus help the experimenter identify the subjects in a timely manner.

At the end of an experimental session, the experimenter would export the recorded times of every participant. This apparently redundant measure allowed us to recover the times of participants who either forgot to log the time, or logged it with insufficient details (i.e., only hours and minutes) in spite of the clear guidelines.

Conversely, relying completely on the timing application would have also been suboptimal: On one occasion, the timing application froze and the only timing information available for the particular tasks the participants were solving were their own logs. On another isolated occasion, a participant raised a hand to ask a question and the experimenter assumed that the participant announced his move to the next task and reset his timer. In this case, the incident was noted by the experimenter and the time was later recovered from the participant's questionnaire. The timing data we collected is presented in Table A.4 in Section A.4.

8.3.3 Correctness Data

The first step towards obtaining the correctness data points was to collect the solutions from our subjects using the questionnaires presented in detail in Section A.1.

Then, to convert task solutions into quantitative information, we needed an oracle set, which would provide both the superset of correct answers and the grading scheme for each task. However, given the complexity of our experiment's design, one single oracle set was not enough. On the one hand we needed a separate oracle for each of the two object systems. On the other hand, we needed separate oracles for the solutions obtained by analyzing an object system with different tools. This happens because for the first few tasks, the data resource of the control groups is source code, while the one of the experimental groups is a FAMIX model of the object system extracted from the source code, which in practice is never 100% accurate [JS03].

To obtain the four oracle sets, three experimenters independently solved the tasks with each combination of treatments (i.e., on the assigned object system, using the assigned tool) and came up with a grading scheme for the tasks. In addition, for the two experimental groups, they computed the results using queries on the model of the object system, to make sure that they did not miss any information caused by limitations of the visualization (e.g., occlusion, too small buildings, etc.). Eventually, by merging the solutions and after discussing the divergences, we obtained the four oracle sets, presented in Section A.4.

Finally, we needed to grade the solution of the subjects. To remove subjectivity when grading, we employed blinding, which implies that, when grading a solution, the experimenter is not aware whether the subject that provided the solution has used an experimental treatment or a control treatment. For this, one of the experimenters created four code names for the groups and created a mapping between groups and code names, known only by him. Then he provided the other two experimenters with the encoded data, along with the encoded corresponding oracle, which allowed them to perform blind grading. In addition, the experimenter who encoded the data performed his grading unblinded. Eventually, the experimenters discussed the differences and converged to the final grading, presented in Table A.3 of Section A.4.

8.3.4 Participants' Feedback

The questionnaire handout ends with a debriefing section, in which the participants are asked to assess the level of difficulty for each task and the overall time pressure, to give us feedback that could potentially help us improve the experiment, and optionally, to share with us any interesting insights they encountered during the analysis.

8.4 Data Analysis

Before performing the hypothesis testing, we had a preliminary look at the collected data, presented in Section 8.4.1. In addition, we looked for outliers in our data, caused by exceptional conditions, which could compromise the soundness of our hypothesis testing results. The outliers analysis is described in Section 8.4.2.

8.4.1 Preliminary Data Analysis

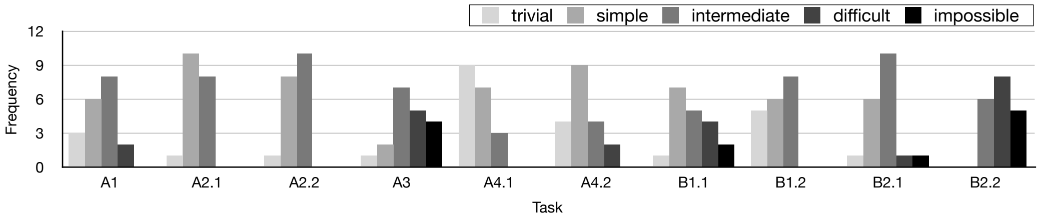
On a first look at the data, we observed an exceptional condition related to task *A4.2*. The data points for this task showed that the experimental group was not able to solve this task, while the control group was quite successful at solving it.

The experimental groups had an average correctness score of 0.06. Out of 22 solutions of the experimental group only one subject achieved a perfect score for this task, while 19 achieved a null score (see the individual scores in Table A.3), in spite of the fact that most of the participants used up the entire ten minutes window allotted for the task (The data on completion time is presented in Figure 8.9). It turned out that the only perfect score was provided by a participant who had used CodeCity on several previous occasions. Moreover, as a central figure in the Moose community, he had a deep knowledge of CodeCity's underlying meta-model. Therefore, he was the only one in the experimental group able to access CodeCity functionality beyond the features presented during the tutorial sessions.

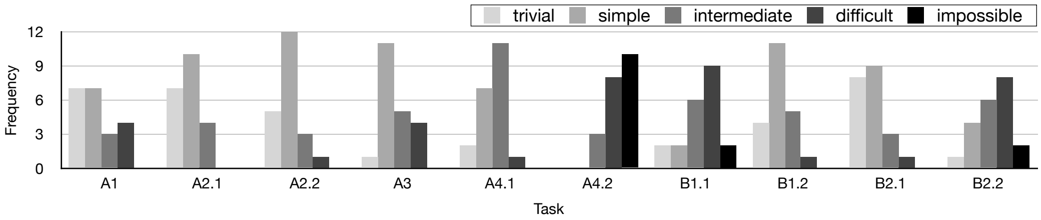
The control groups, on the other hand, had an average correctness score of 0.86, with 15 perfect scores and only 2 null scores. Moreover, the subjects of the control groups were able to complete the task in roughly half the time, on average, allotted for the task.

This task had the highest discrepancy in correctness between control and experimental groups and the participants also perceived its difficulty accordingly. According to the subjects' feedback, most of the subjects in the experimental group described the task as "impossible", while the majority of subjects in the control group described it as "simple" (See Figure 8.3).

The reason for this is that we underestimated the knowledge of CodeCity required to perform this task. Solving this task with CodeCity implies the use of its customization features, which requires a deep knowledge of CodeCity and of the underlying Smalltalk programming language,



(a) Ecl+Excl



(b) CodeCity

Figure 8.3. Histograms of perceived difficulty per task

as demonstrated by the only subject that managed to solve the task in the experimental group. These were unreasonable requirements to expect from the experimental subjects. To eliminate this unusually large discrepancy between the two groups, we excluded the task from the analysis.

8.4.2 Outlier Analysis

Before performing our statistical test, we also followed the suggestion of Wohlin et al. [WRH$^+$00] regarding the removal of outliers caused by exceptional conditions, in order to allow us to draw valid conclusions from our data. During the first experiment run in Bologna, one of the participants experienced serious performance slowdowns, due to the relative low performance of the computer. One of the experimenters made a note about this fact during the experiment and the participant himself complained about it in the debriefing questionnaire. Although this participant was not the only one reporting performance slowdowns, he was by far the slowest as measured by the completion time and, since this represented an exceptional condition, we excluded his data from the analysis. In the same session, another participant got assigned to a Ecl+Excl treatment by mistake, although he specified he did not have any experience with Eclipse, but with another IDE. For this reason, this subject took more time in the first tasks than the others, because of his complete lack of experience with Eclipse. Since we did not want to compromise the analysis by disfavoring any of the groups (i.e., this data point provided the highest completion time and would have biased the analysis by disadvantaging one of the control groups using Ecl+Excl), we excluded also this data point from the analysis.

During the Bologna II run, two participants had compatibility problems with the virtualization software installed on their machines. After unsuccessfully trying for a while to make it work on their machines, they eventually were given our two replacement machines. However, due to these delays and to the tight schedule of the meeting room, we were not able to wait for them to finish the last couple of tasks. We decided to also exclude these two data points from our analysis, for we consider these to be conditions that are unlikely to happen again.

8.5 Subject Analysis

We first performed a pilot study with nine participants, followed by the experiment with 45 participants in several runs. After removing four data points during the outlier analysis, based on the criteria presented Section 8.4.2, we were left with 41 subjects, described next.

All 41 subjects are male, and represent 7 countries: Italy (18 subjects), Belgium (12), Switzerland (7), and Argentina, Canada, Germany, and Romania (1 participant each).

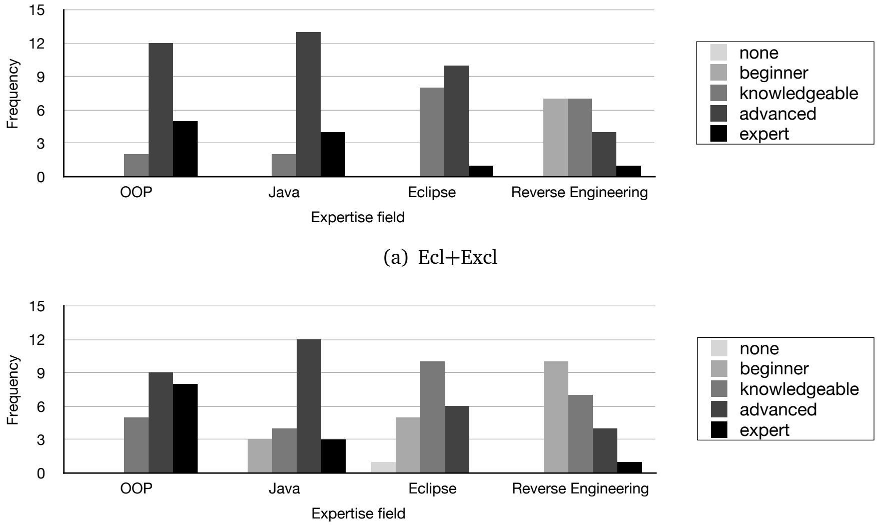
With respect to professional background, our aim was to involve both industry practitioners and people in academia. We managed to obtain valid data for 41 subjects, of which 20 industry practitioners (all advanced), and 21 from academia (of which 9 beginners and 12 advanced). For each of the 4 treatment combinations, we have 8–12 data points.

For each of the four treatment combinations, we obtained a fair distribution of subjects within the remaining three blocks, described in Table 8.1.

			Treatment				
			T1	**T2**	**T3**	**T4**	**Total**
Block	**Academia**	**Beginner**	2	3	2	2	9
		Advanced	2	2	3	5	12
	Industry	**Beginner**	0	0	0	0	0
		Advanced	6	7	3	4	20
		Total	10	12	8	11	41

Table 8.1. Subject distribution

Moreover, the random assignments of treatment within blocks led to a fair distribution of the subjects' expertise among treatment combinations, as seen in Figure 8.4.



(a) Ecl+Excl

(b) CodeCity

Figure 8.4. Histograms of the subjects' expertise level

In only a few cases we intervened in the assignment process. First, when one researcher expressed his wish to be part of a control group, we allowed him to do so. This kept him motivated and he proved to be the fastest subject from a control group. Second, in one of the early experimental runs, we randomly assigned a subject with no Eclipse experience to a Ecl+Excl group. Later, we had to exclude his data point from the analysis (See Section 8.4.2). We learned our lesson from this and later assigned the few subjects with no experience with Eclipse to one of the experimental groups in order not to penalize the control group. However, even in these cases we did not assign them manually, but we randomized the other independent variable, i.e., the object system size. As Figure 8.4 shows, while some of the subjects assigned with CodeCity have little or no experience with Eclipse, every subject assigned with Ecl+Excl is at least *knowledgeable* in using this IDE.

In spite of the fact that we completely lacked subjects in the industry-beginner group, our rich set of data points and the design of our experiment allowed us to perform the complementary analyses presented in Section 8.6.7 (i.e., academia versus industry) and Section 8.6.6 (i.e., beginners versus advanced) .

The age of the participants covers the range 21–52, with an average of 29.8, a median of 29 and the interquartile range of 24–34. The box plots in Figure 8.5 show the age of our participants in each of the three blocks: academia-beginner, academia-advanced, and industry-advanced.

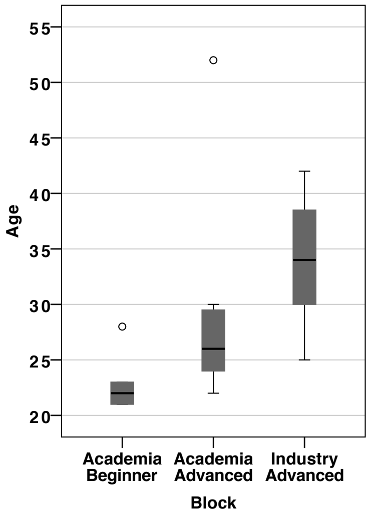


Figure 8.5. The participants' age for each of the three blocks

The age of the academia-beginners has a median of 22 and is fully enclosed in the 21–23 interval (representative for this category, covered almost exclusively by master students), with the exception of one outlier at 28, representing a Ph.D. student with less experience in the required skills.

The age of the academia-advanced group has a median of 26 and an interquartile range of 24–29.5 (also representative for the category, made almost entirely of Ph.D. students), with an outlier representing a university professor.

Finally, the age of the advanced-industry group described by a minimum of 25, a maximum of 42, a median of 34, and an interquartile range of 30–38, shows that industry population is also well represented.

8.6 Experimental Results

Based on the design of our experiment, i.e., a between-subjects design with two independent variables, the suitable parametric test for hypothesis testing is a two-way ANalysis Of VAriance (ANOVA). We performed this test for both *correctness* and *completion time*, using the SPSS[7] statistical package. Before looking at the results of our analysis, we made sure that our data fulfills the three assumptions of the ANOVA test:

1. Homogeneity of variances of the dependent variables. We tested our data for homogeneity of both *correctness* and *completion time*, using Levene's test [Lev60] and in both cases the assumption was met.

2. Normality of the dependent variable across levels of the independent variables. We tested the normality of *correctness* and *completion time* across the two levels of *tool* and *object system size* using the Shapiro-Wilk test for normality [SW65], and also this assumption was met in all cases.

3. Independence of observations. This assumption is implicitly met through the choice of a between-subjects design.

We chose a significance level of .05 ($\alpha = .05$), which corresponds to a 95% confidence interval.

Next, we present the results of the analysis, separately for each of the two dependent variables, i.e., correctness and completion time. Apart from the effect of the main factors, i.e., tool and system size, the ANOVA test allows one to test the interaction between the two factors.

8.6.1 Analysis Results on Correctness

Interaction effect between tool and system size on correctness. First, it is important that there is no interaction between the two factors, that could have affected the correctness. The interaction effect of tool and system size on correctness was not significant, $F(1,37) = .034$, $p = .862$. According to the data, there is no evidence that the variation in correctness between CodeCity and Ecl+Excl depends on the size of the system, which strengthens any observed effect of the tool factor on the correctness.

The effect of tool on correctness. There was a significant main effect of the tool on the correctness of the solutions, $F(1,37) = 14.722$, $p = .001$, indicating that the mean correctness score for CodeCity users was significantly higher than the one for Ecl+Excl users, regardless of the object system's size.

Overall, there was an increase in correctness of 24.26% for CodeCity users ($M = 5.968$, $SD = 1.294$) over Ecl+Excl users ($M = 4.803$, $SD = 1.349$). In the case of the medium size system, there was a 23.27% increase in correctness of CodeCity users ($M = 6.733$, $SD = .959$) over Ecl+Excl users ($M = 5.462$, $SD = 1.147$), while in the case of the large size system, the increase in correctness was 29.62% for CodeCity users ($M = 5.050$, $SD = 1.031$) over Ecl+Excl users ($M = 3.896$, $SD = 1.085$). The data shows that the increase in correctness for CodeCity over Ecl+Excl was higher for the larger system.

[7] http://www.spss.com

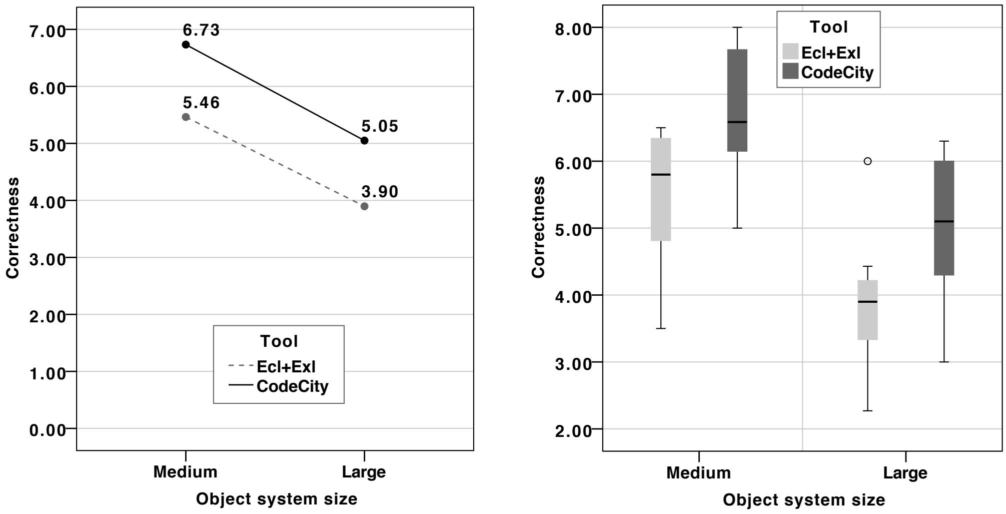
The effect of system size on correctness. Although not the object of the experiment, an expected significant main effect of system size on the correctness of the solutions was observed, $F(1,37) = 26.453$, $p < .001$, indicating that the correctness score was significantly higher for users performing the analysis on the medium size system than for users performing the analysis on the large size system, regardless of the tool they used to solve the tasks.

A detailed description of the statistics related to correctness is given in Table 8.2.

System size	Medium		Large		Overall	
Tool	Ecl+Excl	CodeCity	Ecl+Excl	CodeCity	Ecl+Excl	CodeCity
mean	5.462	6.733	3.896	5.050	4.803	5.968
difference		+23.27%		+29.62%		+24.26%
min	3.50	5.00	2.27	3.00	2.27	3.00
max	6.50	8.00	6.00	6.30	6.50	8.00
median	5.800	6.585	3.900	5.100	4.430	6.065
stdev	1.147	.959	1.085	1.031	1.349	1.294

Table 8.2. Descriptive statistics related to correctness

The main effect of both tool and object system size on correctness, as well as the lack of the effect of interaction between tool and object system size on correctness, are illustrated in Figure 8.6(a), while the correctness box plots across treatments are shown in Figure 8.6(b).



(a) Means, overall

(b) Box plots, across treatments

Figure 8.6. Graphs for correctness

Result Summary for Correctness. The analyzed data allows us to reject the first null hypothesis $H1_0$ in favor of the alternative hypothesis $H1$, which states that the tool impacts the correctness of the solutions to program comprehension tasks. Overall, CodeCity enabled an increase in correctness of 24.26% over Ecl+Excl. This result is statistically significant.

8.6.2 Analysis Results on Completion Time

Interaction effect between tool and system size on completion time. Similarly, it is important that there is no interaction between the two factors, that could have affected the completion time. The interaction effect of tool and system size on completion time was not significant, $F(1,37) = .057$, $p = .813$. According to the data, there is no evidence that the variation in completion time between CodeCity and Ecl+Excl depends on the size of the system, which strengthens any observed effect of the tool factor on the completion time.

The effect of tool on completion time. There was a significant main effect of the tool on the completion time $F(1,37) = 4.392$, $p = .043$, indicating that the mean completion time, expressed in seconds, was significantly lower for CodeCity users than for Ecl+Excl users.

Overall, there was a decrease in completion time of 12.01% for CodeCity users ($M = 36.117$, $SD = 6.910$) over Ecl+Excl users ($M = 41.048$, $SD = 9.174$). In the case of the medium size system, there was a 14.51% decrease in completion time of CodeCity users ($M = 33.178$, $SD = 5.545$) over Ecl+Excl users ($M = 38.809$, $SD = 6.789$), while in the case of the large size system, there is a 10.16% decrease in completion time for CodeCity users ($M = 39.644$, $SD = 6.963$) over Ecl+Excl users ($M = 44.128$, $SD = 11.483$). The data shows that the time decrease for CodeCity users over Ecl+Excl users is only slightly lower in the case of the larger system compared to the time decrease obtained on the medium sized one.

The effect of system size on completion time. Although not the object of the experiment, an expected significant main effect of system size on the completion time was observed, $F(1,37) = 5.962$, $p = .020$, indicating that the completion time was significantly lower for the users performing the analysis on the medium size system than for users performing the analysis on the large size system.

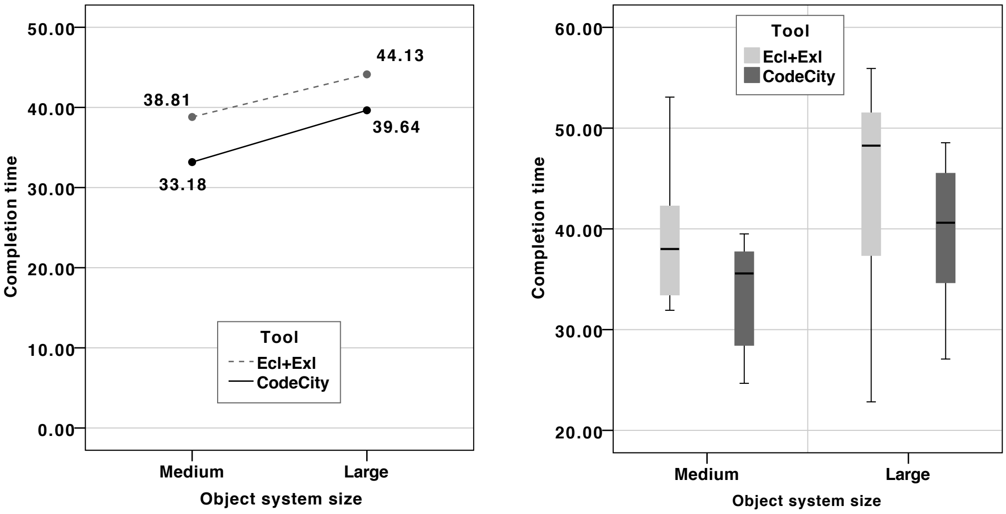
A detailed description of the statistics related to completion time is given in Table 8.3.

System size	**Medium**		**Large**		**Overall**	
Tool	**Ecl+Excl**	**CodeCity**	**Ecl+Excl**	**CodeCity**	**Ecl+Excl**	**CodeCity**
mean	38.809	33.178	44.128	39.644	41.048	36.117
difference		**-14.51%**		**-10.16%**		**-12.01%**
min	31.92	24.67	22.83	27.08	22.83	24.67
max	53.08	39.50	55.92	48.55	55.92	48.55
median	38.000	35.575	48.260	40.610	40.080	36.125
stdev	6.789	5.545	11.483	6.963	9.174	6.910

Table 8.3. Descriptive statistics related to completion time, in minutes

The main effect of both tool and object system size on completion time, as well as the lack of the effect of interaction between tool and object system size on completion time, are illustrated in Figure 8.7(a), while the completion time box plots across treatments are shown in Figure 8.7(b).

Result Summary for Completion Time. The analyzed data allows us to reject the second null hypothesis $H2_0$ in favor of the alternative hypothesis $H2$, which states that the tool impacts the



(a) Means, overall

(b) Box plots, across treatment combinations

Figure 8.7. Graphs for completion time, in minutes

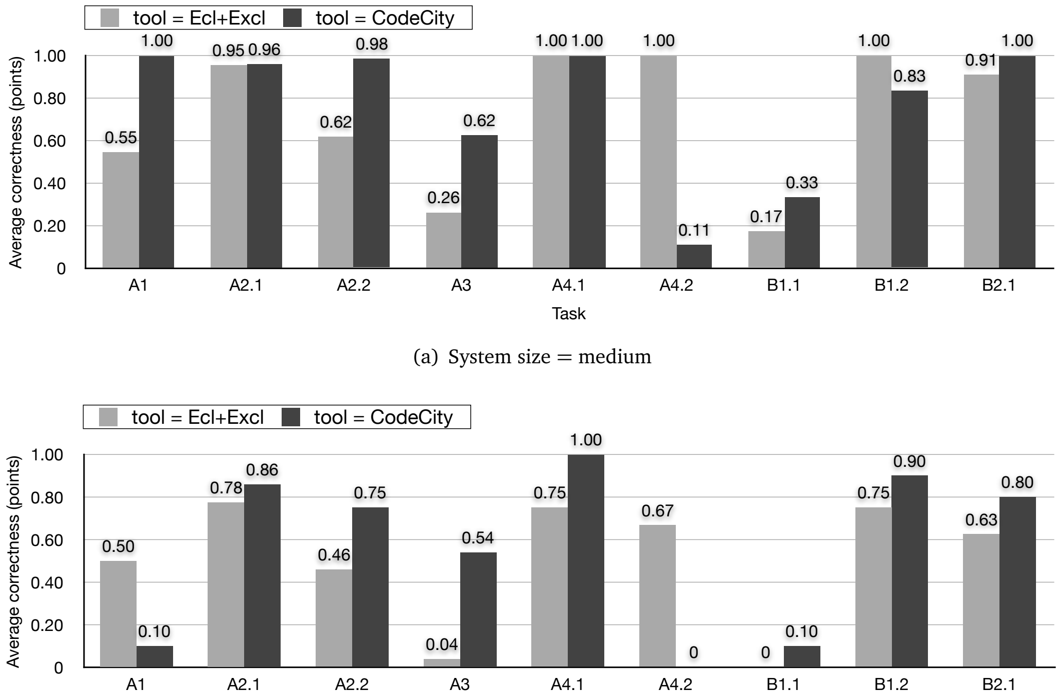
time required to complete program comprehension tasks. Overall, CodeCity enabled a reduction of the completion time of 12.01% over Ecl+Excl. This result is also significant.

8.6.3 Task Analysis

One of the research goals of our experiment was to identify the types of tasks for which CodeCity provides an advantage over Ecl+Excl. To this end, we compared for each task described in Section 7.4.4 the performances (i.e., in terms of correctness and time) of the two levels of the *tool* and reasoned about the potential causes behind the differences. See Figure 8.8 and Figure 8.9 for a graphical overview supporting our task analysis.

A1 - Identifying the convention used to organize unit tests with respect to the tested classes. While Ecl+Excl performed constantly, CodeCity clearly outperformed it on the medium system and underperformed it on the large system. The difference in performance is partially owed to the lack of unit tests in the large system, in spite of the existence of a number of classes named *Test. Only a small number of CodeCity users examined closer the inheritance relations; the majority relied only on the name. The completion time is slightly better for the CodeCity subjects, because they could look at the overview of the system, while in the case of Eclipse, the subjects needed to scroll through the package structure, which rarely fits into one screen.

A2.1 - Determining the spread of a term among the classes. CodeCity performed marginally better than Eclipse in both correctness and completion time. In CodeCity once the search for the term is completed, finding any kind of spread is straightforward with the overview. In Eclipse, the search for a term produces a list of the containing classes, including the packages where these are defined. Given that in this case the list showed many packages belonging to different hierarchies, a dispersed spread is a safe guess.



(a) System size = medium

(b) System size = large

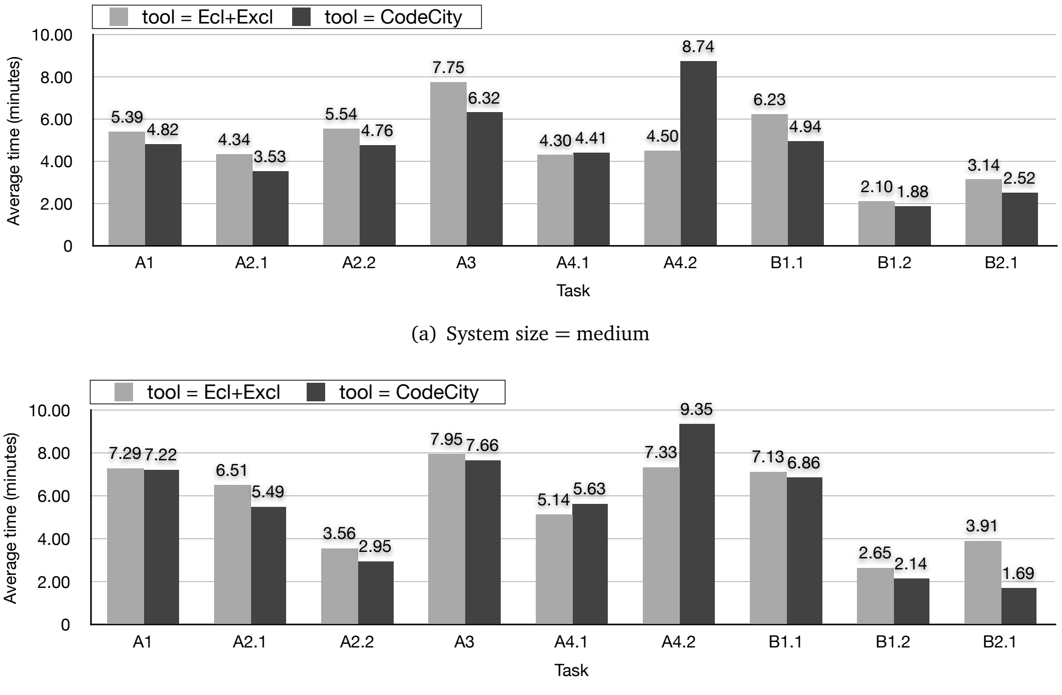
Figure 8.8. Average correctness per task

A2.2 - Determining the spread of a term among the classes. Although the task is similar to the previous, the results in correctness are quite different: CodeCity outperformed Eclipse by 29–38%. The list of classes and packages in Eclipse, without the context provided by an overview (i.e., How many other packages are there in the system?) deceived some of the subjects into believing that the spread of the term is dispersed, while the CodeCity users took advantage of the "big picture" and correctly identified the localized spread of this term.

A3 - Estimating impact. CodeCity outperformed Eclipse in correctness by 40–50%, while for completion time CodeCity was again slightly faster than Eclipse. Finding the caller classes of a given class in Eclipse, as opposed to CodeCity, is not straightforward and the result list provides no overview.

A4.1 - Identifying the classes with the highest number of methods. In terms of correctness, CodeCity was on a par with Excel for the medium size and slightly better than it for the large size. In terms of completion time, the spreadsheet was slightly faster than CodeCity. We learned that, while CodeCity is faster at building an approximate overview of systems, a spreadsheet is faster at finding precise answers in large data sets.

A4.2 - Identifying the classes with the highest ratio of lines of code per method. This task was discarded from the analysis based on the impartiality criteria detailed in Section 8.4.1. We



(a) System size = medium

(b) System size = large

Figure 8.9. Average completion time per task

failed to provide the subjects in the experimental groups the knowledge required to solve this task. The task could not be solved visually, because this would have implied performing an imaginary division between two metrics, i.e., one depicted by size and another by color. Although CodeCity provides a mechanism which allows advanced users to define complex mappings programmatically, by writing Smalltalk code snippets (i.e., this is what subject IE13 did to get his perfect score), we did not cover this feature in our tutorial.

B1.1 - Identifying the package with the highest percentage of god classes. In both correctness and completion time, CodeCity slightly outperformed Excel on this task. The low correctness scores of both tools shows that none of them is good enough to solve the problem alone, although they would complement each other: CodeCity lacks Excel's precision, while Excel would benefit from CodeCity's overview abilities.

B1.2 - Identifying the god class with the highest number of methods. Both tools obtain very good correctness scores, i.e., over 75% in average. Excel is slightly better than CodeCity in the case of the medium size system, while CodeCity outperforms Excel in the case of the large system. While CodeCity's performance is consistent across systems with different sizes, Excel's support is slightly more error-prone in the case of a larger system, which implies the handling of more data.

B2.1 - Identifying the dominant class-level design. In terms of correctness, CodeCity outperforms Excel regardless of the system size. The aggregated information found in CodeCity's disharmony map was less error-prone than counting rows in Excel. In terms of correctness, CodeCity slightly outperforms Excel and the difference is probably owed to the scrolling required for solving the task with Excel.

As expected, at focused tasks such as *A4.1*, *A4.2*, or *B1.1* CodeCity does not perform better than the baseline, because Excel is extremely efficient in finding precise answers (e.g., the largest, the top N, etc.). However, it is surprising that, in most of these tasks, CodeCity managed to be on a par with the baseline. At tasks that benefit from an overview, such as *A2.1*, *A3*, or *B1.2*, CodeCity constantly outperformed Ecl+Excl, in particular in terms of correctness, mainly because the overview allowed for a more confident and quicker answer in the case of the experimental group compared to the control group.

8.6.4 Qualitative Analysis

Task *B2.2*, which dealt with a high-level design problem assessment, is the only qualitative task. The task was excluded from the quantitative analysis upfront, given the difficulty of modeling a right answer, let alone to grade the participants' solutions.The qualitative task was the last one to solve in the experiment, to avoid any side-effects it could place (i.e., fatigue, frustration) on the quantitative tasks. The goal we had in mind when designing the task was to compare the solutions obtained with CodeCity to the ones obtained with the baseline and see whether we can spot some insights exclusively supported by our visual approach.

Although we provided a set of guidance questions for the subjects that needed a starting point (See Section 7.4), we encouraged the subjects to share with us the most interesting or unexpected observations regarding the design problems, in this open-ended task.

Ecl+Excl

As expected, many of the subjects working with Ecl+Excl, limited by the lack of overview, could not provide any insights. Some subjects used the guiding questions as a starting point, and were able to address them partially, as the following examples illustrate:

- "Many packages suffer only of data class and also of god class." (IA01)
- "The design problems do not affect many of the classes. Many god classes are also brain classes. It's hard to get a clear overview by using the spreadsheet. So I don't have any interesting observation to report." (AB07)
- "data classes: 65, god classes: 60, brain classes: 9, on total: 1208." (AA14)
- "Relatively few classes are affected: 64 data classes, 60 god classes, 9 brain classes, out of 1208 classes." (AB09)
- "The majority of the problems seems concentrated in a few packages. Package `findbugs.detect` has a large number of god classes and data classes. 15% of the classes in this package have one of the two problems (30 classes)." (IA20)
- "Only a few classes are affected by any design problem (10%). The design problems affect the system in specific packages; some parts of the system do not show design problems.

There are packages without any design problems. Could not find a package of which all classes are affected." (AA06)

Only very few of the subjects in an experimental group managed to build some insights, either by using advanced functionality of the spreadsheet or by using experience to overcome the limitations of the tool:

- "Most of the god class and data class problems appear in the `findbugs` and `findbugs.detect` packages. Probably the detection algorithms are complex and procedural." (AA12)
- "High correlation of God and Brain class, low correlation of Data class." (AA07) *Observation*. The participant enriched his observations with graphs, probably synthesizing the ones he produced with Excel.
- "There are many design problems. I can't really say how big the problems are, because I don't know the purpose of the specific class or if the class is in USE. `detect` seems to be a HOTSPOT for possible defect search." (IA19)

CodeCity

Similarly, many of the subjects in the experimental groups followed the guiding questions. However, they were able to address most of the questions:

- "Almost all the packages are affected by design problems. The only package that seems to be well built is `org.gudy.azureus2.plugins`. The god class and brain class problems are very spread." (IA01)
- "brain classes: 9, god classes: 20, data classes: 67. Most problems seem to occur in the GUI classes, which is not really a surprise. The `detect` and `workflow` classes are also affected, these packages seem to be core packages. There's only 1 brain class located in the `detect` core package. The following packages are not affected: `jaif`, `xml`, `bcel`, `bcp`." (AA04)
- "About 10% of the classes have design problems. Data classes are the most frequent problem, but those classes are not very big. Packages not affected by this are `findbugs.detect`, `findbugs.workflow`, and findbugs.classfile.analysis. I think the god classes are a bigger problem, 62 god classes is a lot, and most packages are affected." (AB05)
- "The biggest problem according to the method are god classes and brain classes. There are 110 god classes and 54 brain classes. The problems affect most of the system, but not all. Notably, the packages `org.gudy.azureus2` and `org.bouncy.castle` aren't affected. Of the infected packages, none really stands out (I think, not sure). The design problem is near ubiquitous!" (AA01)
- "Brain classes are only 16 and mostly limited to a few packages, and only 1 or 2 per package. God classes: 72; also spread out. More god+brain in `az1` than `az2`; in `az2` in `peer.impl` and `disk.impl`. Packages `org.gudy.az2.plugins` and `edu.harvard...` are mostly unaffected. `org.bouncycastle` has mostly only data classes" (AA02)
- "The biggest part of the classes (>90%) are not affected by any design problem. The most relevant design problem is the huge percentage of data classes (256). There are packages in the system affected by only data class problem." (IA06)

Many of the subjects in the experimental groups provided interesting insights into the analyzed system's design problems. The different approaches to gain the insights (i.e., semantics, dependencies), often revealed within the answers, lead to a wide range of points of view:

- "Data classes are far more apparent than god/brain classes. There's about 256 data classes, ca. 55 brain classes and 111 god classes. Most data classes can be found in the `ui.swt.views.stats` package, which isn't very surprising, considering the nature of stats. However, the number of classes using these data suppliers is quite limited (15). The `org.gudy.core3.peer` and `org.gudy.core3.download` packages contain a high concentration of god classes. The packages `org.gudy.azureus2.platform` and `org.plugins.*` seem to be mostly free of problem classes." (AB02)

- "The three types of problems are distributed in all the packages of the project. In particular, data classes are uniformly distributed, while the god classes, having a presence, are being identified as the largest classes of the main packages. There are no packages with all the classes affected by problems, but there are packages with no design problems. As an observation about the project, I observed that the largest and most problematic classes are those which implement the GUI, but also the access to the DB and command-line, hence the parts of the system interfaced with other external software." (IA07)

- "`MainBugFrame` and `MainFrame` are obviously god classes that would be worth refactoring. The detect package seems to be a data package, but it's ok. `DBCloud` seems odd, could not understand what it does based on outgoing/incoming calls. `anttask` could be improved. `BugInstance` has lots of incoming calls, and is yet a god class which can introduce fragility in the design." (AA04)

- "As the name says, package `detect` is the central package with most classes. It also concentrates the most design problems and it manages to feature all of these: GodClasses, BrainClasses, DataClasses. The most problematic BrainClass is `DBCloud`. The rest 7 BrainClasses are either in the UI, which is partly expected, or in the `detect` package, which should define mostly standalone detection components. The most interesting DataClass is `ba.vna.ValueNumber` because it is accessed by many classes also from the outside of the `ba.vna` package. It looks like the most important packages feature one Brain Class. Only small/marginal packages are unaffected by design problems." (IA13)

Quite as expected, the lack of the overview in the control group is strongly felt. The answers in the experimental groups are visibly richer and contain more insights, while the ones in the control groups, with few exceptions, only prove that having the raw data is far from "seeing".

8.6.5 Debriefing Questionnaire

In the following we briefly summarize the formal and informal feedback obtained during the debriefing.

Four subjects in the experimental group complained about the fact that they were not shown how to access the advanced mapping customization features in CodeCity, which caused their frustration in front of the task *A4.2*, which was one failure of our design. One subject in the experimental group suggested a shortcomings of the tool, i.e., "small buildings are barely visible".

Eight subjects in the control group complained about the fact that the pre-experiment assessment did not contain a question about the skills with Excel. We discuss this threat to validity in

Section 8.7. Two other subjects in the control group said they hated the search functionality in Eclipse.

Two subjects praised the setup and the organization of the experimental runs. One subject found the experiment very stimulating. Several industry developers expressed their interest in using CodeCity in their line of work after the experiment.

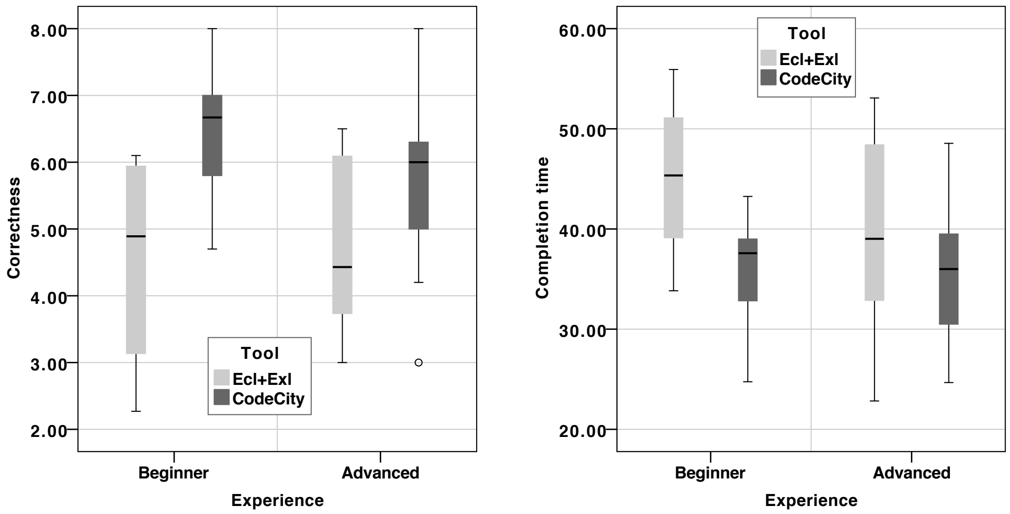
Two participants, one in the experimental group and one in the control group, expressed their concern about the fact that Eclipse was not an appropriate baseline for our high-level analysis and suggested Sonar or a UML tool as alternative. One other participant wondered about the practical relevance of the results.

A subject in the experimental group suggested another debriefing question: “What have you learned about the system?”. He shared with us that: “I gave the stats, but learned 0”.

One subject in the experimental group had several suggestions: “The experiment does not evaluate CodeCity in the presence of deeper knowledge about the system. However, I believe it can prove useful in the more advanced steps of analysis, because a map becomes more useful as we get more comfortable with the larger parts[...]”

8.6.6 Experience Level

We compared the correctness and time scores across the two levels of experience, i.e., beginner and advanced. The data shows that CodeCity outperforms Ecl+Excl in both correctness and completion time, regardless of the experience level, as shown in Figure 8.10.



(a) Correctness

(b) Completion time

Figure 8.10. Performance comparison between experience levels: Beginner vs Advanced

An interesting observation is that CodeCity users have much less variability in performance than the users of Ecl+Excl, which shows a more consistent performance of CodeCity compared to Ecl+Excl. This can be assessed visually, as the boxes of the box plots for CodeCity are much smaller than the one for the baseline.

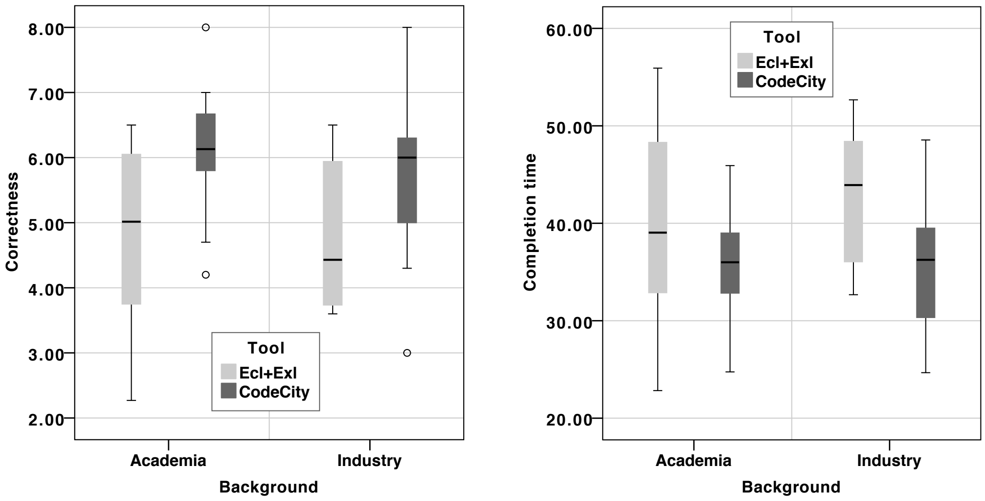
The correctness data shows that the difference with which CodeCity outperforms Ecl+Excl is slightly higher for beginners than for advanced users. Moreover, among CodeCity users, the beginners slightly outperform the advanced. One possible explanation is that our only beginners were the students from Antwerp, which have used the video tutorials prior to the experiment and were therefore very well prepared in using CodeCity.

The time data shows that the difference with which CodeCity outperforms Ecl+Excl is higher for beginners than for advanced. While among CodeCity users the time performance is almost constant across experience levels, among Ecl+Excl users the advanced outperform the beginners.

These results are an indication of the ease of use and the usability of CodeCity, which enables its users to obtain better results than with conventional, non-visual approaches, even without extensive training.

8.6.7 Background

We also compared the correctness and time scores across the two levels of background, i.e., academia and industry. The performance for CodeCity is better in both correctness and completion time, regardless of the background, as shown in Figure 8.11.



(a) Correctness (b) Completion time

Figure 8.11. Performance comparison between background: Academia vs Industry

Again, the box plots of CodeCity users (in particular the ones from academia) have shorter boxes than the ones of Ecl+Excl, which show a more consistent performance of CodeCity, compared to Ecl+Excl.

In terms of correctness, the difference with which CodeCity outperforms Ecl+Excl is only slightly higher for academia than for industry.

In terms of completion time, the difference between CodeCity and Ecl+Excl is minimal in the case of academia and more consistent for industry practitioners.

The results show that in terms of correctness the benefits of CodeCity over the non-visual approach are visible for both academics and industry practitioners, while in terms of completion time CodeCity provides a boost in particular to industry practitioners.

8.7 Threats to Validity

In this section, we discuss the threats to our experiment's validity. For experiments that validate applied research, the categories are—in decreasing order of importance: internal, external, construct and conclusion validity [WRH$^+$00].

8.7.1 Internal Validity

The internal validity refers to uncontrolled factors that may influence the effect of the treatments on the dependent variables.

Subjects. Several threats to internal validity refer to the subjects. One threat is that subjects may not have been competent enough. To reduce this threat, before the experiment we analyzed the subjects' competence in several relevant fields and made sure that they had at least a minimal knowledge of object-oriented programming, Java, and for the subjects assigned with Ecl+Excl, of Eclipse. A second threat was that the expertise of the subjects may not have been fairly distributed across the control and experimental groups. We mitigate this threat by using blocking and randomization when assigning treatments to subjects. A third internal threat is that the subjects may not have been properly motivated. This treatment is diminished by the fact that all the subjects' volunteered to participate in the experiment, by filling out the online questionnaire.

Tasks. First, the choice of tasks may have been biased to the advantage of CodeCity. We alleviate this threat by presenting the tasks in context, with rationale and targeted users. Moreover, we tried to include tasks which clearly do not advantage CodeCity (e.g., any task which focuses on precision, rather than on locality), which is visible from the per-task results and from the perceived difficulty of the subjects in the experimental groups. Another threat is that the tasks may have been too difficult or that not enough time was allotted for them. To alleviate this threat we performed a pilot study and we collected feedback about the perceived task difficulty and time pressure. As a consequence, we excluded one task which was extremely difficult for one group and trivial for the other. In addition, this task was the only one that showed a ceiling effect (i.e., most subjects used up the entire time) for the affected group.

Baseline. The baseline was composed of two different tools (i.e., Eclipse and Excel), while CodeCity is one tool, and this might have affected the performance of the control group. We attenuate this threat by designing the task such that no task requires the use of both tools. Moreover, all the tasks that were to be solved with Eclipse were grouped in the first half of the experiment, while all the tasks that were to be solved with Excel were grouped in the second half of the experiment. This allowed us to minimize the effect of switching between tools to only one time, between tasks A3 and A4.1. The good scores obtained by the Ecl+Excl subjects on task A4.1, in both correctness and time, do not provide any indication of such a negative effect.

Data differences. CodeCity relies on FAMIX models of the systems, while Eclipse works with the source code. These data differences might have an effect on the results of the two groups and this represents a threat to internal validity. To alleviate it, we accurately produced the answer model based on the available artifact, i.e., source code or FAMIX model, and made sure that the slight differences between the two data sources do not lead to incompatible answers.

Session differences. There were seven sessions and the differences among them may have influenced the result. To mitigate this threat, we performed four different sessions with nine subjects in total during a pre-experiment pilot phase and obtained a stable and reliable experimental setup (i.e., instrumentation, questionnaires, experimental kit, logistics). Even so, there were some inconsistencies among sessions. For instance the fact that some of the participants in the Bologna XPUG paired to perform the experiment was an unexpected factor, but watching them in a real work-like situation was more valuable for us than imposing the experiment's constraints at all costs. Moreover, there were four industry practitioners who performed the experiment remotely, controlled merely by their conscience. Given the value of data points from these practitioners and the reliability of these particular persons (i.e., one of the experimenters knew them personally), we trusted them without reservation.

Training. The fact that we only trained the subjects with the experimental treatment may have influenced the result of the experiment. We afforded to do so because we chose a strong baseline tool set, composed of two state-of-the-practice tools, and we made sure that the control subjects had a minimum of knowledge with Eclipse. Although many of the Ecl+Excl subjects remarked the fact that we should have included Excel among the assessed competencies, they scored well on the tasks with Excel, due to the rather simple operations (i.e., sorting, arithmetic operations between two columns) required to solve the tasks. As many of the CodeCity subjects observed, one hour of demonstration of a new and mostly unknown tool will never leverage years of use, even if sparse, of popular tools such as Eclipse or Excel.

Paper support. From our experience and from the feedback of some of our subjects, we have indications that the fact that the answers had to be written on paper may have influenced the results. The influence of this threat is not changing the result, but it reduces the effect of the tool, since for some of the tasks (i.e., the one requiring writing down some package or class names) the writing part takes longer than the operations required to reach the solution. If this effect does exist, it affects all subjects regardless of the tool treatment. Removing it would only increase the difference with which CodeCity outperformed Ecl+Excl.

8.7.2 External Validity

The external validity refers to the generalizability of the experiment's results.

Subjects. One threat to external validity is the representativeness of the subjects for the targeted population. To mitigate this threat, we categorized our subjects in four categories along two axes (i.e., background and experience level) and strived to cover all the categories. We obtained a balanced mix of academics (both beginners and advanced) and industry practitioners (only advanced). The lack of industry beginners may have had an influence on the results. However, our analysis of the performances across experience levels indicates that CodeCity supported beginner users well, who were better than the advanced at outperforming Ecl+Excl. Therefore, we believe that the presence of industry beginners would only strengthen these results.

Tasks. Another external validity threat is the representativeness of the tasks, i.e., that the tasks may not reflect real reverse engineering situations. We could not match our analysis with any of the existing frameworks, because they do not support design problem assessment and, in addition, they are either too low-level (e.g., the questions asked by practitioners during a programming change task by Sillito et al. [SMDV06]), or biased towards dynamic analysis tools (e.g., the framework of comprehension activities by Pacione et al. [PRW04]). To alleviate this threat, we complemented our tasks with usage scenarios and targeted users.

Object systems. The representativeness of the object systems is another threat. In spite of the increased complexity in organizing the experiment and analyzing the data, introduced by a second independent variable, we chose to perform the experiment with two different object systems. Besides our interest in analyzing the effect of the object system size on the performance of CodeCity's users, we also applied the lessons learned from Quante's experiment [Qua08] that the results obtained on a single object system are not reliable. The two object systems we opted for are well-known open-source systems of different, realistic sizes (see Table 7.2) and of orthogonal application domains. It is not known how appropriate these systems are for the reverse-engineering tasks we designed, but the variation in the solutions to the same task shows that the systems are quite different.

Experimenter effect. One of the experimenters is also the author of the approach and of the tool. This may have influenced any subjective aspect of the experiment. Although, we tried to mitigate this threat in several ways (e.g., objective oracle set, blinded marking), we cannot exclude all the possible influences of this factor on the results of the experiment.

8.7.3 Construct Validity

The construct validity concerns generalizing the result of the experiment to the concepts or theories behind the experiment.

Hypothesis guessing. Another threat to internal validity is that the subjects were aware that the author of CodeCity was among the experimenters and that the purpose of the experiment was to compare the performance of CodeCity with a baseline. To alleviate this threat, we clearly explained to them before each experiment session that it was the tool's support that was being measured and *not* the subjects' performances and we asked them to do their best in solving the tasks, regardless of the tool they have been assigned with. An indication that this was clearly understood by the participants is that the absolute best completion time and one of the best correctness scores in the case of the large object system were obtained by subjects in control groups (i.e., AA07 and AA05, respectively).

8.7.4 Conclusion Validity

The conclusion validity refers to the ability to draw the correct conclusions about the relation between the treatment and the experiment's outcome.

Fishing for results. Searching for a specific result is a threat to conclusion validity, for it may influence the result. In this context, a threat is that task solutions may not have been graded correctly. To mitigate this threat, three experimenters independently built a model of the answers and a grading scheme and then reached consensus. Moreover, the grading was performed in a similar manner and two of the three experimenters graded the solutions blinded, i.e., without knowing the treatments (e.g., tool) used to obtain the solutions.

8.8 Summary

In this part, we addressed our claim that the city metaphor enables the creation of efficient software visualizations. To demonstrate this claim, we designed and performed an extensive experiment which spanned over six months of time. We managed to engage large samples of our target population, which covered both academia and industry, and included both beginner and advanced participants.

By performing various analyses of the data we gathered from our controlled experiment, we learned several facts about our approach. The main result of our experiment is the fact that our approach outperforms in both correctness and completion time the combination of two state-of-the-practice exploration tools. This result is statistically significant, which is a solid indication that, at least for the program comprehension and design quality assessment, our city metaphor enables the creation of efficient software visualizations.

Apart from an aggregated analysis, we performed a detailed analysis of each task, which provided a number of insights on the type of tasks that our approach best supports. Unsurprisingly, in the case of focused tasks, i.e., tasks which require very precise answers, CodeCity did not perform better than Excel. However, for most of these tasks, our approach managed to be on a par with the baseline, which is an unexpected result. As for the tasks that benefit from an overview of the system, CodeCity constantly outperformed the baseline, in particular in terms of correctness. The qualitative task confirmed our view that our code city visualizations present the data such that it provides an advantage over conventional presentations, such as a spreadsheet.

The last type of analysis we performed was to try to find evidence on whether the benefits of our approach are received differently, depending on the background or experience level. An interesting find was that the beginners performed more consistently in terms of correctness than the advanced, which shows our approach does not have a steep learning curve and supports the user in obtaining good results even without much training. Another interesting insight was that in terms of completion time, our approach provides a boost in particular to the industry practitioners. This is a very positive result in the context of a potential adoption of our visualization approach in industry.

We believe that researchers interested in evaluating their tools should benefit from our experience. Therefore, we provided the complete raw and processed data (i.e., the pre-experiment questionnaire, the experiment questionnaires, solution oracles and grading systems, correction scores and measured completion time) to allow reviewers to evaluate the experiment more thoroughly and fellow researchers replicate the experiment or start from its design, as a base for their own experiment.

After successfully addressing both claims of our thesis, we look back at the "big picture" of our research and reflect on the meaning and consequences of our findings, in Part IV.

Part IV

Epilogue

Chapter 9

Conclusions

We built our thesis starting from the observation that, in spite of the relatively many instances of the city metaphor in software visualization, there is no evidence on the value of this metaphor for reverse engineering. The research goal that guided our work was finding out whether the city metaphor is valuable for reverse engineering through software visualization.

At the beginning of this dissertation we claimed that the city metaphor is versatile and that it enables the building of efficient visualizations for reverse engineering. From that point on, we presented evidence which supports our claim. At this point, we take a step back and reflect on the results of our research in the context of the thesis.

9.1 Reflections

In the following, we discuss the achievements and limitations of our work in the light of the thesis, and share a number of insights related to people and tools.

9.1.1 Versatility

The first claim of the thesis was that the city metaphor is versatile. To demonstrate this aspect of the metaphor, we chose to apply it in three different contexts related to reverse engineering, i.e., program comprehension, software evolution analysis, and design quality assessment. For each of the three application contexts, we devised a number of visualizations aimed at revealing certain aspects of the software systems which are important for that particular context.

With the visualizations in place, we illustrated our approach by means of case studies. For each case study, we found interesting insights, i.e., a form of subjective evidence that our approach works. In the case of the evolution analysis application, several key developers of the software systems we used as case studies confirmed the correctness of our findings regarding the history of their systems.

The approach we built around the city metaphor has several limitations.

First, our search for meaningful visual representations for the various software artifacts was to a certain extent restricted by the central metaphor. Regardless of how appealing an idea was for a visualization technique, it *needed* to fit the city metaphor. For instance, although the bundled edges make a promising visualization technique, they do not fit well the city metaphor.

Second, due to our choice of layout and building orientation (i.e., buildings are always parallel), the code cities in our visualizations are more similar to the New York's Manhattan borough rather than to any other city. Moreover, one may argue that the artifacts in a code city (e.g., buildings, districts) are merely schematic representations of real city artifacts. We chose simplicity over accuracy to allow for a simple visual language which facilitates the interpretation of the visualized data.

9.1.2 Efficiency

Our second claim in this dissertation is that the city metaphor enables the creation of efficient software visualizations. The analysis of the data obtained from the controlled experiment provided evidence that, for the two application contexts that we evaluated, our approach overall outperformed the baseline, in terms of both correctness of solutions and task completion time.

However, because of the complexity of such a controlled experiment, there are many threats to validity. Therefore, these results should be interpreted with moderation. It is possible that under different circumstances (e.g., including a third system, having female subjects), the results would have not been the same.

Moreover, the application to software evolution of the city metaphor was not evaluated. We regret our inability of finding a baseline for this, because we consider this to be one of the strongest application contexts of our approach.

9.1.3 People & Tools

From our experience with the users of CodeCity we have evidence that learning the language of the metaphor is fairly easy. Adopting the simple conventions allows the users to look at, and reason about, software in a different way than the traditional one (i.e., source code and UML diagrams). There is a genuine curiosity most people manifest when they first see a visualization of CodeCity: "I would really like to see *my* system visualized as a code city".

After discussing our approach with different developers, we realized that the slow pace of tool adoption in industry is not caused by the people, but rather by the organizations: Most of the practitioners we discussed with were quite enthusiastic about our approach and some of them were looking forward to using CodeCity to complement their working activities.

Moreover, the feedback we received from CodeCity users is encouraging. An experienced industry practitioner shared with us his experience with CodeCity: " [. . .] I loaded my music composition application into CodeCity. See the attached picture. Interestingly, the resulting image looks like expected. To a large extent, it matches the image I always had in my head. [. . .] It feels rewarding and satisfying to see a beautiful visualization of the work that took so many years to accomplish." The fact that our approach produced a visualization of a software system which matched the developer's mental model is a valuable insight.

We strived to build a highly configurable and flexible tool, which would allow us to extend it to unforeseen directions. This touches again the topic of research intertwined with tool building. Had we built a badly designed tool or had we not invested the time and effort into the design and implementation of the tool, we would have not been able to extend the metaphor's implementation so easily. Moreover, the language-independent meta-model that we rely on allowed us to apply our approach on systems written in different programming languages and opened a larger user base for CodeCity.

We invested significant time to make CodeCity robust, to optimize it for scalability, and to take it beyond the "research prototype" status. The robustness, scalability, and usability of the tool were all important for both the outcome of the experiment and for allowing us to chose from a large number of case studies.

The availability of the tool allowed industry practitioners to try CodeCity out before enrolling as subjects in the experiment. After experimenting with CodeCity, many of them believed that it could help them in their daily work and this was enough motivation for them to want to learn how to use it.

9.2 Contributions

After demonstrating our thesis, we reflect on the contributions of the work presented in this dissertation:

The definition of a versatile city metaphor for software visualization. We defined an initial metaphor for program comprehension and iteratively enriched it to support two new applications, i.e., software evolution analysis and design quality assessment. With a similar amount of effort as the one we invested for each new application, we believe we could further extend our city metaphor to other facets of software.

The application of the city metaphor to program comprehension. We first applied our approach based on the city metaphor in the context of program comprehension and obtained a rough "big picture" of our case study system. Moreover, our approach allowed us to detect a number of system hotspots. We used these outliers as starting points for more in-depth analyses that led us to interesting insights about the system.

The application of the city metaphor to software evolution. We described three visualization techniques we devised for the software evolution applications. The visualizations enabled us to acquire valuable insight about the system unattainable outside the evolutionary context and complementary to the insights gained in the context of program comprehension. For this application, we consulted the developers of the case study systems, who confirmed our findings.

The application of the city metaphor to design quality assessment. We described a software visualization technique inspired from disease maps, called disharmony map. Disharmony maps enabled us to focus on the design problems, while maintaining the general context, i.e., the overview of the system. Due to the precise information obtained by using detection strategies, our approach is less prone to false positive and false negatives than traditional visualization approaches, which rely on spotting outliers in terms of simple software metrics.

The implementation of a tool which supports our city metaphor. We implemented CodeCity, a tool that supports all three applications of our city metaphor. The effort we invested into the design and implementation of the tool payed off from several perspectives: it demonstrated the value of our approach, it allowed us to confidently perform an experiment in front of experimented industry practitioners and academics, and it enabled us to illustrate our applications by means of case studies. Moreover, the scalability of CodeCity, and implicitly of our approach, allowed us to visualize large software systems of up to three million lines of code.

The empirical validation of our approach through a replicable controlled experiment. Although our intention has been to empirically evaluate our city metaphor in all three application contexts, we only managed to evaluate two, i.e., program comprehension and software design quality assessment. For reasons described in Section 7.4.2, we dropped the evaluation of the software evolution analysis application context. However, we conducted an extensive experiment, which spanned over six months of time. We were able to engage large samples of our target population, which covered both academia and industry, and included both beginner and advanced participants. The design of our experiment, which followed a set of guidelines extracted from the body of literature, enabled us to collect rich data, which allowed us to perform a number of interesting analyses. The most important results of our controlled experiment was that our approach outperformed in terms of both correctness of the solution and task completion time the state-of-the-practice. The result is statistically significant. Moreover, by providing the entire experimental data set, we ensured the replicability of our experiment.

9.3 Future Work

One part of our metaphor which may be considered insufficiently covered is the representation of relations, which is an open challenge in software visualization. We found a visual representation for relations able to reduce complexity, in particular when combined with an opportunistic approach. The problem of this representation in the context of our research is that it does not fit the city metaphor. Therefore, a potential future work research direction is finding city-related representations for the relations, such as the plumbing system, or the street infrastructure. We briefly explored the possibility of mapping relations to streets, but the resulting layout algorithm turned out to be too computationally expensive.

The layout could also be improved by taking dependencies into account when placing the buildings, i.e., the more connected two software entities are, the closer their city representation. The advantage of such a spring-based layout is that it optimizes the length of the edges: Strongly coupled buildings have shorter edges.

Due to the difficulties in building a fair baseline, we gave up the evaluation of the software evolution application context of our city metaphor. Given that we consider it one of the strong points of our approach, we believe that such an evaluation would complement well the results obtained from our controlled experiment. Moreover, since we have already evaluated the other two application contexts, it would probably be easy to find a state-of-the-practice tool for software system evolution analysis.

Another part at which our approach is deficient is the lack of integration with IDEs and other tools that make up the workflow of practitioners.

An future work direction in this context would be integrating CodeCity with a popular IDE such as Eclipse. A first step towards this goal is Biaggi's Citylyzer [Bia08], a port of CodeCity as an Eclipse plugin.

A second direction we envision is including our approach in the versioning workflow. An example scenario is the following: Whenever a system has a new release or version, the source code of the system is parsed and the model is passed to CodeCity, which produces a canonical visualization of the system. An alternative to performing a *diff* between two revisions would be observing the evolution of the system's code city.

9.4 Final Thoughts

Through the work presented in this dissertation, we demonstrated the usefulness of the city metaphor in building software visualizations that support reverse engineering. Although "software systems as cities" is just one way of visualizing software systems, it illustrates well the advantages of software visualization.

Although many of the practitioners we talked to acknowledge the shortage of tools that support a holistic perception of software systems, visualization is still underrepresented in the current software development workflows. In this context, objective results obtained from empirical evaluations may increase the credibility of software visualization, which should be deemed not as a replacement of the state of the practice in industry, but rather as a complement.

Our vision is to see the day in which software developers will use visualization, while coding, to stay aware of the way their systems evolve. Some might argue that this will never happen. We believe it is just a question of time.

Part V

Appendix

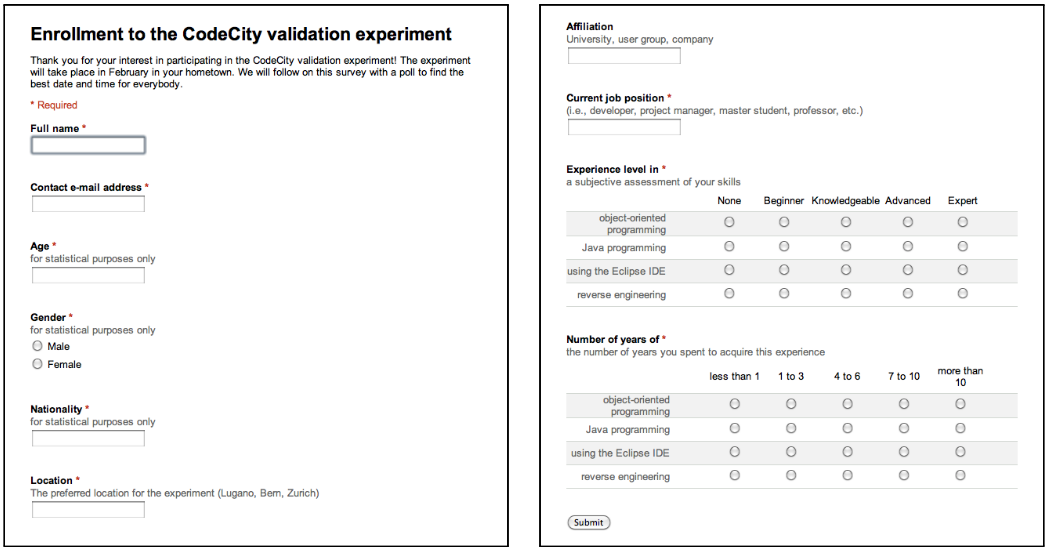
Appendix A

Experimental Data

In this chapter, we present all the details about our experiment, complementary to the ones presented in Part III, which make our experiment repeatable: questionnaires, oracle sets, and the entire experimental data set collected from our subjects.

A.1 Pre-Experiment Questionnaire

Using Google Docs[1], we designed an online questionnaire that served both to provide an easily accessible platform for the volunteers to enroll and to allow capturing the personal information that we used to assign the subjects to blocks and treatments (See Figure A.1).



Enrollment to the CodeCity validation experiment

Thank you for your interest in participating in the CodeCity validation experiment! The experiment will take place in February in your hometown. We will follow on this survey with a poll to find the best date and time for everybody.

* Required

Full name *

Contact e-mail address *

Age *
for statistical purposes only

Gender *
for statistical purposes only
○ Male
○ Female

Nationality *
for statistical purposes only

Location *
The preferred location for the experiment (Lugano, Bern, Zurich)

Affiliation
University, user group, company

Current job position *
(i.e., developer, project manager, master student, professor, etc.)

Experience level in *
a subjective assessment of your skills

	None	Beginner	Knowledgeable	Advanced	Expert
object-oriented programming	○	○	○	○	○
Java programming	○	○	○	○	○
using the Eclipse IDE	○	○	○	○	○
reverse engineering	○	○	○	○	○

Number of years of *
the number of years you spent to acquire this experience

	less than 1	1 to 3	4 to 6	7 to 10	more than 10
object-oriented programming	○	○	○	○	○
Java programming	○	○	○	○	○
using the Eclipse IDE	○	○	○	○	○
reverse engineering	○	○	○	○	○

Figure A.1. The enrollment online questionnaire we used for collecting personal information

[1] http://docs.google.com

A.2 Experiment Questionnaire

The content of the questionnaires, with all the variations due to the different treatment combinations, is presented in the following. Their actual form and presentation is exemplified in Figure A.2 and Figure A.3, which show the questionnaire for the treatment combination *T1*.

A.2.1 Introduction

The aim of this experiment is to compare tool efficiency in supporting software practitioners analyzing medium to large-scale software systems.

You will use <toolset>[2] to analyze <object system name>[3], a <object system description>[4] written in Java.

You are given maximum 100 minutes for solving 10 tasks (10 minutes per task).

You are asked:

- not to consult any other participant during the experiment;
- to perform the tasks in the specified order;
- to write down the current time each time before starting to read a task and once after completing all the tasks;
- to announce the experimenter that you are moving on to another task, in order to reset your 10-minutes-per-task allocated timer;
- not to return to earlier tasks, because it affects the timing;
- for each task, to fill in the required information. In the case of multiple choices check the most appropriate answer and provide additional information, if requested.

The experiment is concluded with a short debriefing questionnaire.

Thank you for participating in this experiment!
Richard Wettel, Michele Lanza, Romain Robbes

A.2.2 Tasks

A1 [Structural Understanding]

Task
Locate all the unit test classes of the system (typically called `*Test` in Java) and identify the convention (or lack of convention) used by the system's developers to organize the unit tests.

[2] *CodeCity* for treatments 1 and 2, *Eclipse + Excel with CSV data concerning metrics and design problems* for treatments 3 and 4

[3] *Azureus* for treatments 1 and 3, *FindBugs* for treatments 2 and 4

[4] *a BitTorrent client* for treatments 1 and 3, *a bug searching tool based on static analysis* for treatments 2 and 4

Solution (*multiple choice*)

- ○ Centralized. There is a single package hierarchy, whose root package is (write down the full name of the package): . . . [5].
- ○ Dispersed. The test classes are located in the same package as the tested classes.
- ○ Hybrid. Some test classes are defined in the central test package hierarchy, with the root in package (provide the full name of the package) . . . , while some test classes are defined elsewhere. An example of such a test class is: . . . , defined in package (write down the full name): . . . .
- ○ Other. Detail your answer: . . . .

A2.1 [Concept Location]

Task
Using the <feature name>[6] (and any other) feature in <toolset>, look for the term <term 1>[7] in the names of the classes and their attributes and methods, and describe the spread of these classes in the system.

Solution *(multiple choice)*

- ○ Localized. All the classes related to this term are located in one or two packages. Provide the full name of these packages: . . . .
- ○ Dispersed. Many packages in the system contain classes related to the given term. Indicated 5 packages (or all of them if there are less than 5) writing their full names: . . . .

A2.2 [Concept Location]

Task
Using the <feature name> (and any other) feature in <toolset>, look for the term <term 2>[8] in the names of the classes and their attributes and methods, and describe the spread of these classes in the system[9].

Solution *(multiple choice)*

- ○ Localized. All the classes related to this term are located in one or two packages. Provide the full name of these packages: . . . .
- ○ Dispersed. Many packages in the system contain classes related to the given term. Indicated 5 packages (or all of them if there are less than 5) writing their full names:

[5] The placeholders presented here are not proportional in length to the variable-size blanks used in the actual questionnaires.

[6] *search by term* for treatments 1 and 2, *Java search* for treatments 3 and 4

[7] *skin* for treatments 1 and 3, *annotate* for treatments 2 and 4

[8] *tracker* for treatments 1 and 3, *infinite* for treatments 2 and 4

[9] The task is similar to the previous one, but the terms are chosen such that they cover the opposite solution.

. . . .

A3 [Impact Analysis]

Task
Evaluate the change impact of class <class A3>[10], by considering its caller classes (classes invoking any of its methods). The assessment is done in terms of both intensity (number of potentially affected classes) and dispersion (how these classes are distributed in the package structure).

Solution *(multiple choice)*

○ Unique location. There are . . . classes potentially affected by a change in the given class, all defined in a single package, whose full name is . . . .

○ Global. Most of the system's packages (more than half) contain at least one of the . . . classes that would be potentially affected by a change in the given class.

○ Multiple locations. There are . . . classes potentially affected by a change in the given class, defined in several packages, but less than half of the system's packages. Indicate up to 5 packages containing the most of these classes: . . . .

A4.1 [Metric Analysis]

Task
Find the 3 classes with the highest number of methods in the system.

Solution *(ranking)*
The classes with the highest number of methods are (in descending order):

1. class . . . defined in package (full name) . . . , containing . . . methods.
2. class . . . defined in package (full name) . . . , containing . . . methods.
3. class . . . defined in package (full name) . . . , containing . . . methods.

A4.2 [Metric Analysis]

Task
Find the 3 classes with the highest average number of lines of code per method in the system. The value of this metric is computed as:

lines of code per method $= \frac{\text{number of lines of code}}{\text{number of methods}}$

Solution *(ranking)*
The classes with the highest average number of lines of code per methods are (in descending order):

[10] `org.gudy.azureus2.ui.swt.Utils` for treatments 1 and 3, `edu.umd.cs.findbugs.OpcodeStack` for treatments 2 and 4

1. class . . . defined in package (full name) . . . , has an average of . . . lines of code per method.
2. class . . . defined in package (full name) . . . , has an average of . . . lines of code per method.
3. class . . . defined in package (full name) . . . , has an average of . . . lines of code per method.

B1.1 [God Class Analysis]

Task
Identify the package with the highest percentage of god classes in the system. Write down the full name of the package, the number of god classes in this package, and the total number of classes in the package.

Solution
The highest percentage of god classes in the system is found in package . . . , which contains . . . god classes out of . . . classes.

B1.2 [God Class Analysis]

Task
Identify the god class containing the largest number of methods in the system.

Solution
The god class with the largest number of methods in the system is class . . . , defined in package (write down the full name) . . . , which contains . . . methods.

B2.1 [Design Problem Assessment]

Task
Based on the design problem information available in <toolset>[11], identify the dominant class-level design problem (i.e., the design problem that affects the largest number of classes) in the system.

Solution *(multiple choice)*
The dominant class-level design problem is

- ○ Brain Class, which affects a number of . . . classes.
- ○ Data Class, which affects a number of . . . classes.
- ○ God Class, which affects a number of . . . classes.

[11] *CodeCity* for treatments 1 and 2, *the spreadsheet* for treatments 3 and 4

B2.2 [Design Problem Assessment]

Task
Write an overview of the class-level design problems in the system. Are the design problems affecting many of the classes? Are the different design problems affecting the system in an equal measure? Are there packages of the system affected exclusively by only one design problem? Are there packages entirely unaffected by any design problem? Or packages with all classes affected? Describe your most interesting or unexpected observations about the design problems.

Solution *(free form)*
. . .

A.3 Debriefing Questionnaire

Time pressure. On a scale from 1 to 5, how did you feel about the time pressure? Please write in the box below the answer that matches your opinion the most:
. . .
The time pressure scale corresponds to:

1. Too much time pressure. I could not cope with the tasks, regardless of their difficulty.
2. Fair amount of pressure. I could certainly have done better with more time.
3. Not so much time pressure. I had to hurry a bit, but it was OK.
4. Very little time pressure. I felt quite comfortable with the time given.
5. No time pressure at all.

Difficulty. Regardless of the given time, how difficult would you rate this task? Please mark the appropriate difficulty for each of the tasks[12]:
. . .

Comments. Enter comments and/or suggestions you may have about the experiment, which could help us improve it.
. . .

Miscellaneous. It is possible that you have discovered some interesting insights about the system during the experiment and that the format of the answer did not allow you to write it, or that it was not related to the question. In this case, please share with us what you discovered (optional).
. . .

[12] The scale for difficulty was, in decreasing order: impossible, difficult, intermediate, simple, trivial

A.4 Task Solution Oracles

The four oracles we used to grade the task solutions of our subjects are presented in the following.

A.4.1 T1: Azureus, analyzed with CodeCity

A1

Either
There are no unit tests in the system *[1pt]*,
or
Centralized in a single package hierarchy whose root is in **org.gudy.azureus2.ui.console.multiuser** *[1pt]*. Since there is only one test class (i.e., TestUserManager), if they don't give the full correct answer, the answer is completely wrong.

A2.1

Dispersed *[0pts otherwise]*
in the following **(max. 5) packages** *[0.2pts for each]*:

- com.aelitis.azureus.core
- com.aelitis.azureus.core.content
- com.aelitis.azureus.core.download
- com.aelitis.azureus.core.impl
- com.aelitis.azureus.core.lws
- com.aelitis.azureus.core.peermanager.peerdb
- com.aelitis.azureus.core.stats
- com.aelitis.azureus.core.torrent
- com.aelitis.azureus.plugins.net.buddy
- com.aelitis.azureus.plugins.net.buddy.swt
- com.aelitis.azureus.plugins.net.buddy.tracker
- com.aelitis.azureus.plugins.removerules
- com.aelitis.azureus.plugins.sharing.hoster
- com.aelitis.azureus.plugins.startstoprules.defaultplugin
- com.aelitis.azureus.plugins.tracker.dht
- com.aelitis.azureus.plugins.tracker.local
- com.aelitis.azureus.plugins.tracker.peerauth

- com.aelitis.azureus.ui.swt.content.columns
- com.aelitis.azureus.ui.swt.shells.main
- com.aelitis.azureus.util
- org.gudy.azureus2.core3.download
- org.gudy.azureus2.core3.download.impl
- org.gudy.azureus2.core3.global
- org.gudy.azureus2.core3.global.impl
- org.gudy.azureus2.core3.ipfilter.impl.tests
- org.gudy.azureus2.core3.logging
- org.gudy.azureus2.core3.peer
- org.gudy.azureus2.core3.peer.impl.control
- org.gudy.azureus2.core3.tracker.client
- org.gudy.azureus2.core3.tracker.client.impl
- org.gudy.azureus2.core3.tracker.client.impl.bt
- org.gudy.azureus2.core3.tracker.client.impl.dht
- org.gudy.azureus2.core3.tracker.host
- org.gudy.azureus2.core3.tracker.host.impl
- org.gudy.azureus2.core3.tracker.protocol.udp
- org.gudy.azureus2.core3.tracker.server
- org.gudy.azureus2.core3.tracker.server.impl
- org.gudy.azureus2.core3.tracker.server.impl.dht
- org.gudy.azureus2.core3.tracker.server.impl.tcp
- org.gudy.azureus2.core3.tracker.server.impl.udp
- org.gudy.azureus2.core3.tracker.util
- org.gudy.azureus2.core3.util
- org.gudy.azureus2.plugins
- org.gudy.azureus2.plugins.download
- org.gudy.azureus2.plugins.torrent
- org.gudy.azureus2.plugins.tracker

- org.gudy.azureus2.plugins.tracker.web
- org.gudy.azureus2.plugins.ui.config
- org.gudy.azureus2.plugins.ui.tables
- org.gudy.azureus2.pluginsimpl.local
- org.gudy.azureus2.pluginsimpl.local.download
- org.gudy.azureus2.pluginsimpl.local.torrent
- org.gudy.azureus2.pluginsimpl.local.tracker
- org.gudy.azureus2.pluginsimpl.remote
- org.gudy.azureus2.pluginsimpl.remote.download
- org.gudy.azureus2.pluginsimpl.remote.tracker
- org.gudy.azureus2.ui.console.commands
- org.gudy.azureus2.ui.swt
- org.gudy.azureus2.ui.swt.mainwindow
- org.gudy.azureus2.ui.swt.maketorrent
- org.gudy.azureus2.ui.swt.views
- org.gudy.azureus2.ui.swt.views.configsections
- org.gudy.azureus2.ui.swt.views.stats
- org.gudy.azureus2.ui.swt.views.tableitems.mytorrents
- org.gudy.azureus2.ui.swt.views.tableitems.mytracker
- org.gudy.azureus2.ui.webplugin

A2.2

Either
Localized in:

- **com.aelitis.azureus.ui.skin** *[0.5pts]*
- **com.aelitis.azureus.ui.swt** *[0.5pts]*

or
Localized in **com.aelitis.azureus.ui** *[1pt]*.

A3

Multiple locations.
There are **211/212** classes *[0.5pts]*
defined in the following **(max. 5) packages** *[0.1 for each]*:
either aggregated

- com.aelitis.azureus.core.metasearch.impl
- com.aelitis.azureus.plugins.net.buddy.swt
- com.aelitis.azureus.ui.swt
- org.gudy.azureus2.plugins.ui.tables
- org.gudy.azureus2.ui
  - org.gudy.azureus2.ui.common.util
  - org.gudy.azureus2.ui.swt
  - org.gudy.azureus2.ui.systray

or detailed

- com.aelitis.azureus.core.metasearch.impl
- com.aelitis.azureus.plugins.net.buddy.swt
- com.aelitis.azureus.ui.swt
- com.aelitis.azureus.ui.swt.browser
- com.aelitis.azureus.ui.swt.browser.listener
- com.aelitis.azureus.ui.swt.browser.msg
- com.aelitis.azureus.ui.swt.columns.torrent
- com.aelitis.azureus.ui.swt.columns.vuzeactivity
- com.aelitis.azureus.ui.swt.content
- com.aelitis.azureus.ui.swt.content.columns
- com.aelitis.azureus.ui.swt.devices
- com.aelitis.azureus.ui.swt.devices.add
- com.aelitis.azureus.ui.swt.devices.columns
- com.aelitis.azureus.ui.swt.imageloader
- com.aelitis.azureus.ui.swt.shells
- com.aelitis.azureus.ui.swt.shells.main
- com.aelitis.azureus.ui.swt.shells.uiswitcher

- com.aelitis.azureus.ui.swt.skin
- com.aelitis.azureus.ui.swt.subscriptions
- com.aelitis.azureus.ui.swt.uiupdater
- com.aelitis.azureus.ui.swt.utils
- com.aelitis.azureus.ui.swt.views
- com.aelitis.azureus.ui.swt.views.skin
- com.aelitis.azureus.ui.swt.views.skin.sidebar
- org.gudy.azureus2.plugins.ui.tables
- org.gudy.azureus2.ui.common.util
- org.gudy.azureus2.ui.swt
- org.gudy.azureus2.ui.swt.associations
- org.gudy.azureus2.ui.swt.auth
- org.gudy.azureus2.ui.swt.components
- org.gudy.azureus2.ui.swt.components.graphics
- org.gudy.azureus2.ui.swt.components.shell
- org.gudy.azureus2.ui.swt.config
- org.gudy.azureus2.ui.swt.config.generic
- org.gudy.azureus2.ui.swt.donations
- org.gudy.azureus2.ui.swt.help
- org.gudy.azureus2.ui.swt.ipchecker
- org.gudy.azureus2.ui.swt.mainwindow
- org.gudy.azureus2.ui.swt.maketorrent
- org.gudy.azureus2.ui.swt.nat
- org.gudy.azureus2.ui.swt.networks
- org.gudy.azureus2.ui.swt.osx
- org.gudy.azureus2.ui.swt.pluginsimpl
- org.gudy.azureus2.ui.swt.progress
- org.gudy.azureus2.ui.swt.sharing.progress
- org.gudy.azureus2.ui.swt.shells

- org.gudy.azureus2.ui.swt.speedtest
- org.gudy.azureus2.ui.swt.update
- org.gudy.azureus2.ui.swt.views
- org.gudy.azureus2.ui.swt.views.clientstats
- org.gudy.azureus2.ui.swt.views.columnsetup
- org.gudy.azureus2.ui.swt.views.configsections
- org.gudy.azureus2.ui.swt.views.file
- org.gudy.azureus2.ui.swt.views.peer
- org.gudy.azureus2.ui.swt.views.piece
- org.gudy.azureus2.ui.swt.views.stats
- org.gudy.azureus2.ui.swt.views.table.impl
- org.gudy.azureus2.ui.swt.views.tableitems.mytorrents
- org.gudy.azureus2.ui.swt.views.tableitems.peers
- org.gudy.azureus2.ui.swt.views.utils
- org.gudy.azureus2.ui.swt.welcome
- org.gudy.azureus2.ui.swt.wizard
- org.gudy.azureus2.ui.systray

A4.1

The **3 classes** with the highest number of methods are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **PEPeerTransportProtocol**
   defined in package **org.gudy.azureus2.core3.peer.impl.transport**
   contains **161** methods;

2. class **DownloadManagerImpl**
   defined in package **org.gudy.azureus2.core3.download.impl**
   contains **156** methods;

3. class **PEPeerControlImpl**
   defined in package **org.gudy.azureus2.core3.peer.impl.control**
   contains **154** methods.

A4.2

The **3 classes** with the highest average number of lines of code per method are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **BouncyCastleProvider**
   defined in package **org.bouncycastle.jce.provider**
   has an average of **547** lines of code per method;

2. class **9 (anonymous)**
   defined in package **com.aelitis.azureus.core.dht.nat.impl**
   has an average of **222** lines of code per method;

3. class **MetaSearchListener**
   defined in package **com.aelitis.azureus.ui.swt.browser.listener**
   has an average of **219** lines of code per method.

Just in case the participant thought class **9** must be an error, the 4th classified is
class **MultiPartDecoder**
defined in package **com.aelitis.azureus.core.util**
has an average of **211** lines of code per method.

B1.1

The package with the highest percentage of god classes in the system is
com.aelitis.azureus.core.metasearch.impl.web.rss *[0.8pts]*
which contains **1** *[0.1pts]* god classes
out of a total of **1** *[0.1pts]* classes.

B1.2

The god class containing the largest number of methods in the system is
class **PEPeerTransportProtocol** *[0.8pts]*
defined in package **org.gudy.azureus2.core3.peer.impl.transport** *[0.1pts]*
which contains **161** *[0.1pts]* methods.

B2.1

The dominant class-level design problem is
DataClass *[0.5pts]*
which affects a number of **256** *[0.5pts]* classes.

A.4.2 T2: Findbugs, analyzed with CodeCity

A1

Dispersed. *[1pt]*

A2.1

Localized *[0.5pts]*
in package **edu.umd.cs.findbugs.detect** *[0.5pts]*.

A2.2

Dispersed *[0pts otherwise]*
in the following **(max. 5) packages** *[0.2pts for each]*:

- edu.umd.cs.findbugs
- edu.umd.cs.findbugs.anttask
- edu.umd.cs.findbugs.ba
- edu.umd.cs.findbugs.ba.deref
- edu.umd.cs.findbugs.ba.jsr305
- edu.umd.cs.findbugs.ba.npe
- edu.umd.cs.findbugs.ba.vna
- edu.umd.cs.findbugs.bcel
- edu.umd.cs.findbugs.classfile
- edu.umd.cs.findbugs.classfile.analysis
- edu.umd.cs.findbugs.classfile.engine
- edu.umd.cs.findbugs.classfile.impl
- edu.umd.cs.findbugs.cloud
- edu.umd.cs.findbugs.cloud.db
- edu.umd.cs.findbugs.detect
- edu.umd.cs.findbugs.gui
- edu.umd.cs.findbugs.gui2
- edu.umd.cs.findbugs.jaif
- edu.umd.cs.findbugs.model
- edu.umd.cs.findbugs.visitclass
- edu.umd.cs.findbugs.workflow

A3

Multiple locations.
There are **40/41** *[0.5pts]* classes
defined in the following **3 packages** *[1/6pts for each]*:

- **edu.umd.cs.findbugs**
- **edu.umd.cs.findbugs.bcel**
- **edu.umd.cs.findbugs.detect**

A4.1

The **3 classes** with the highest number of methods are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **AbstractFrameModelingVisitor**
   defined in package **edu.umd.cs.findbugs.ba**
   contains **195** methods;
2. class **MainFrame**
   defined in package **edu.umd.cs.findbugs.gui2**
   contains **119** methods;
3. class **BugInstance**
   defined in package **edu.umd.cs.findbugs**
   contains **118** methods
   or
   class **TypeFrameModelingVisitor**
   defined in package **edu.umd.cs.findbugs.ba.type**
   contains **118** methods.

A4.2

The **3 classes** with the highest average number of lines of code per method are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **DefaultNullnessAnnotations**
   defined in package **edu.umd.cs.findbugs.ba**
   has an average of **124** lines of code per method;
2. class **DBCloud.PopulateBugs**
   defined in package **edu.umd.cs.findbugs.cloud.db**
   has an average of **114.5** lines of code per method;
3. class **BytecodeScanner**
   defined in package **edu.umd.cs.findbugs.ba**
   has an average of **80.75** lines of code per method.

B1.1

The package with the highest percentage of god classes in the system is
edu.umd.cs.findbugs.ba.deref *[0.8pts]*
which contains **1** *[0.1pts]* god classes
out of a total of **3** *[0.1pts]* classes.

B1.2

The god class containing the largest number of methods in the system is
class **MainFrame** *[0.8pts]*
defined in package **edu.umd.cs.findbugs.gui2** *[0.1pts]*
which contains **119** *[0.1pts]* methods.

B2.1

The dominant class-level design problem is
DataClass *[0.5pts]*
which affects a number of **67** *[0.5pts]* classes.

A.4.3 T3: Azureus, analyzed with Eclipse + Spreadsheet with metrics

A1

Either
There are no unit tests in the system *[1pt]*,
or
Centralized in a single package hierarchy whose root is in **org.gudy.azureus2.ui.console.multiuser** *[1pt]*. Since there is only one test class (i.e., TestUserManager), if they don't give the full correct answer, the answer is completely wrong.

A2.1

Dispersed
in the following **(max. 5) packages** *[0.2pts each]*:

- com.aelitis.azureus.core
- com.aelitis.azureus.core.content
- com.aelitis.azureus.core.download
- com.aelitis.azureus.core.impl
- com.aelitis.azureus.core.lws
- com.aelitis.azureus.core.peermanager.peerdb
- com.aelitis.azureus.core.stats
- com.aelitis.azureus.core.torrent

- com.aelitis.azureus.plugins.net.buddy
- com.aelitis.azureus.plugins.net.buddy.swt
- com.aelitis.azureus.plugins.net.buddy.tracker
- com.aelitis.azureus.plugins.removerules
- com.aelitis.azureus.plugins.sharing.hoster
- com.aelitis.azureus.plugins.startstoprules.defaultplugin
- com.aelitis.azureus.plugins.tracker.dht
- com.aelitis.azureus.plugins.tracker.peerauth
- com.aelitis.azureus.ui
- com.aelitis.azureus.ui.swt.content.columns
- com.aelitis.azureus.ui.swt.shells.main
- com.aelitis.azureus.util
- org.gudy.azureus2.core3.download
- org.gudy.azureus2.core3.download.impl
- org.gudy.azureus2.core3.global
- org.gudy.azureus2.core3.global.impl
- org.gudy.azureus2.core3.logging
- org.gudy.azureus2.core3.peer
- org.gudy.azureus2.core3.peer.impl.control
- org.gudy.azureus2.core3.tracker.client
- org.gudy.azureus2.core3.tracker.client.impl
- org.gudy.azureus2.core3.tracker.client.impl.bt
- org.gudy.azureus2.core3.tracker.client.impl.dht
- org.gudy.azureus2.core3.tracker.host
- org.gudy.azureus2.core3.tracker.host.impl
- org.gudy.azureus2.core3.tracker.protocol.udp
- org.gudy.azureus2.core3.tracker.server
- org.gudy.azureus2.core3.tracker.server.impl
- org.gudy.azureus2.core3.tracker.util

- org.gudy.azureus2.core3.util
- org.gudy.azureus2.plugins
- org.gudy.azureus2.plugins.download
- org.gudy.azureus2.plugins.torrent
- org.gudy.azureus2.plugins.tracker
- org.gudy.azureus2.plugins.tracker.web
- org.gudy.azureus2.plugins.ui.config
- org.gudy.azureus2.plugins.ui.tables
- org.gudy.azureus2.pluginsimpl.local
- org.gudy.azureus2.pluginsimpl.local.download
- org.gudy.azureus2.pluginsimpl.local.tracker
- org.gudy.azureus2.pluginsimpl.remote
- org.gudy.azureus2.pluginsimpl.remote.download
- org.gudy.azureus2.pluginsimpl.remote.tracker
- org.gudy.azureus2.ui.console.commands
- org.gudy.azureus2.ui.swt
- org.gudy.azureus2.ui.swt.mainwindow
- org.gudy.azureus2.ui.swt.maketorrent
- org.gudy.azureus2.ui.swt.views
- org.gudy.azureus2.ui.swt.views.configsections
- org.gudy.azureus2.ui.swt.views.tableitems.mytorrents)
- org.gudy.azureus2.ui.webplugin

A2.2

Localized in
com.aelitis.azureus.ui *[1pt]*.

To ease the grading for the case in which the answer is incomplete, here is the complete hierarchy:

- com.aelitis.azureus.ui.common.viewtitleinfo
- com.aelitis.azureus.ui.skin

- com.aelitis.azureus.ui.swt
  - com.aelitis.azureus.ui.swt.content
  - com.aelitis.azureus.ui.swt.devices
    - com.aelitis.azureus.ui.swt.devices.add
  - com.aelitis.azureus.ui.swt.imageloader
  - com.aelitis.azureus.ui.swt.shells.main
  - com.aelitis.azureus.ui.swt.skin
  - com.aelitis.azureus.ui.swt.subscription
  - com.aelitis.azureus.ui.swt.toolbar
  - com.aelitis.azureus.ui.swt.views
    - com.aelitis.azureus.ui.swt.views.skin
      - com.aelitis.azureus.ui.swt.views.skin.sidebar

A3

Multiple locations *[0pts otherwise]*
There are **220/221** classes *[0.5pts]*
defined in the following **(max. 5) packages** *[0.1 each]*:

- com.aelitis.azureus.core.metasearch.impl
- com.aelitis.azureus.plugins.net.buddy.swt
- com.aelitis.azureus.plugins.startstoprules.defaultplugin.ui.swt
- com.aelitis.azureus.ui.swt
- com.aelitis.azureus.ui.swt.browser
- com.aelitis.azureus.ui.swt.browser.listener
- com.aelitis.azureus.ui.swt.browser.msg
- com.aelitis.azureus.ui.swt.columns.torrent
- com.aelitis.azureus.ui.swt.columns.vuzeactivity
- com.aelitis.azureus.ui.swt.content
- com.aelitis.azureus.ui.swt.content.columns
- com.aelitis.azureus.ui.swt.devices
- com.aelitis.azureus.ui.swt.devices.add
- com.aelitis.azureus.ui.swt.devices.columns
- com.aelitis.azureus.ui.swt.imageloader
- com.aelitis.azureus.ui.swt.shells

- com.aelitis.azureus.ui.swt.shells.main
- com.aelitis.azureus.ui.swt.shells.uiswitcher
- com.aelitis.azureus.ui.swt.skin
- com.aelitis.azureus.ui.swt.subscriptions
- com.aelitis.azureus.ui.swt.uiupdater
- com.aelitis.azureus.ui.swt.utils
- com.aelitis.azureus.ui.swt.views
- com.aelitis.azureus.ui.swt.views.skin
- com.aelitis.azureus.ui.swt.views.skin.sidebar
- org.gudy.azureus2.plugins.ui.tables
- org.gudy.azureus2.ui.common.util
- org.gudy.azureus2.ui.swt
- org.gudy.azureus2.ui.swt.associations
- org.gudy.azureus2.ui.swt.auth
- org.gudy.azureus2.ui.swt.components
- org.gudy.azureus2.ui.swt.components.graphics
- org.gudy.azureus2.ui.swt.components.shell
- org.gudy.azureus2.ui.swt.config
- org.gudy.azureus2.ui.swt.config.generic
- org.gudy.azureus2.ui.swt.config.wizard
- org.gudy.azureus2.ui.swt.donations
- org.gudy.azureus2.ui.swt.help
- org.gudy.azureus2.ui.swt.ipchecker
- org.gudy.azureus2.ui.swt.mainwindow
- org.gudy.azureus2.ui.swt.maketorrent
- org.gudy.azureus2.ui.swt.minibar
- org.gudy.azureus2.ui.swt.nat
- org.gudy.azureus2.ui.swt.networks
- org.gudy.azureus2.ui.swt.osx

- org.gudy.azureus2.ui.swt.pluginsimpl
- org.gudy.azureus2.ui.swt.progress
- org.gudy.azureus2.ui.swt.sharing.progress
- org.gudy.azureus2.ui.swt.shells
- org.gudy.azureus2.ui.swt.speedtest
- org.gudy.azureus2.ui.swt.update
- org.gudy.azureus2.ui.swt.views
- org.gudy.azureus2.ui.swt.views.clientstats
- org.gudy.azureus2.ui.swt.views.columnsetup
- org.gudy.azureus2.ui.swt.views.configsections
- org.gudy.azureus2.ui.swt.views.file
- org.gudy.azureus2.ui.swt.views.peer
- org.gudy.azureus2.ui.swt.views.piece
- org.gudy.azureus2.ui.swt.views.stats
- org.gudy.azureus2.ui.swt.views.table.impl
- org.gudy.azureus2.ui.swt.views.tableitems.mytorrents
- org.gudy.azureus2.ui.swt.views.tableitems.peers
- org.gudy.azureus2.ui.swt.views.utils
- org.gudy.azureus2.ui.swt.welcome
- org.gudy.azureus2.ui.swt.wizard
- org.gudy.azureus2.ui.systray

A4.1

The **3 classes** with the highest number of methods are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **PEPeerTransportProtocol**
   defined in package **org.gudy.azureus2.core3.peer.impl.transport**
   contains **161** methods;
2. class **DownloadManagerImpl**
   defined in package **org.gudy.azureus2.core3.download.impl**
   contains **156** methods;
3. class **PEPeerControlImpl**
   defined in package **org.gudy.azureus2.core3.peer.impl.control**
   contains **154** methods.

A4.2

The **3 classes** with the highest average number of lines of code per method are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **BouncyCastleProvider**
   defined in package **org.bouncycastle.jce.provider**
   has an average of **547** lines of code per method;

2. class **9 (anonymous)**
   defined in package **com.aelitis.azureus.core.dht.nat.impl**
   has an average of **222** lines of code per method;

3. class **MetaSearchListener**
   defined in package **com.aelitis.azureus.ui.swt.browser.listener**
   has an average of **219** lines of code per method.

Just in case the participant thought class **9** must be an error, the 4th classified is
class **MultiPartDecoder**
defined in package **com.aelitis.azureus.core.util**
has an average of **211** lines of code per method.

B1.1

The package with the highest percentage of god classes in the system is
com.aelitis.azureus.core.metasearch.impl.web.rss *[0.8pts]*
which contains **1** *[0.1pts]* god classes
out of a total of **1** *[0.1pts]* classes.

B1.2

The god class containing the largest number of methods in the system is
class **PEPeerTransportProtocol** *[0.8pts]*
defined in package **org.gudy.azureus2.core3.peer.impl.transport** *[0.1pts]*
which contains **161** *[0.1pts]* methods.

B2.1

The dominant class-level design problem is
DataClass *[0.5pts]*
which affects a number of **255** *[0.5pts]* classes.

A.4.4 T4: Findbugs, analyzed with Eclipse + Spreadsheet with metrics

A1

Dispersed. *[1pt]*

A2.1

Localized *[0.5pts]*
in package **edu.umd.cs.findbugs.detect** *[0.5pts]*.

A2.2

Dispersed
in the following **5 packages** *[0.2pts each]*:

- edu.umd.cs.findbugs.ba
- edu.umd.cs.findbugs.ba.jsr305
- edu.umd.cs.findbugs.classfile.analysis
- edu.umd.cs.findbugs.detect
- edu.umd.cs.findbugs.gui

A3

Multiple locations. *[0pts otherwise]*
There are **41/42** *[0.5pts]* classes
defined in the following **4 packages** *[0.125pts each]*:

- **edu.umd.cs.findbugs**
- **edu.umd.cs.findbugs.ba**
- **edu.umd.cs.findbugs.bcel**
- **edu.umd.cs.findbugs.detect**

A4.1

The **3 classes** with the highest number of methods are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **MainFrame**
   defined in package **edu.umd.cs.findbugs.gui2**
   contains **119** methods;

2. class **BugInstance**
   defined in package **edu.umd.cs.findbugs**
   contains **118** methods;

3. class **TypeFrameModelingVisitor**
   defined in package **edu.umd.cs.findbugs.ba.type**
   contains **118** methods;

A4.2

The **3 classes** with the highest average number of lines of code per method are *[$\frac{1}{3}$pts each correctly placed and $\frac{1}{6}$pts each misplaced]*:

1. class **DefaultNullnessAnnotations**
   defined in package **edu.umd.cs.findbugs.ba**
   has an average of **124** lines of code per method;
2. class **DBCloud.PopulateBugs**
   defined in package **edu.umd.cs.findbugs.cloud.db**
   has an average of **114.5** lines of code per method;
3. class **BytecodeScanner**
   defined in package **edu.umd.cs.findbugs.ba**
   has an average of **80.75** lines of code per method.

B1.1

The package with the highest percentage of god classes in the system is
edu.umd.cs.findbugs.ba.deref *[0.8pts]*
which contains **1** *[0.1pts]* god classes
out of a total of **3** *[0.1pts]* classes.

B1.2

The god class containing the largest number of methods in the system is
class **MainFrame** *[0.8pts]*
defined in package **edu.umd.cs.findbugs.gui2** *[0.1pts]*
which contains **119** *[0.1pts]* methods.

B2.1

The dominant class-level design problem is
DataClass *[0.5pts]*
which affects a number of **65** *[0.5pts]* classes.

A.5 Data

To provide a fully transparent experimental setup, we make available the entire data set of our experiment.

In Table A.1 we present the subjects and the personal information that we relied on when we assigned them to the different blocks (i.e., based on experience and background).

Once the subjects were assigned to the three blocks (i.e., we did not have any subjects in the industry-beginner block), within each block we assigned the subjects to treatment combinations using randomization. The assignment of subjects to treatments and blocks is presented in Table A.2, clustered by the treatment combination, to ease comparison between the different levels of the independent variables.

Using the criteria described in detail in Section 8.3.3, we obtained the correctness levels presented in Table A.3. Based on the reasoning presented in the Section 8.4.2, we decided to eliminate the correctness and timing results for task *A4.2*. Therefore, the last column of the table, which represents the correctness after discarding the aforementioned task, presents the data that we used for our analysis on correctness.

The completion times for each tasks and overall are presented in Table A.4. Since we discarded the correctness results for task *A4.2*, we also discard the completion time data for the same task. The last column of the table, which represents the overall completion time after discarding the aforementioned task, presents the data that we used for our analysis on completion time.

Finally, Table A.5 presents the data we collected from the subjects regarding the perceived time pressure and the difficulty level per task, as experienced by our subjects. This data allowed us to determine whether there was a task which was highly unfair for one of the groups. Moreover, it provided us important hints on the type of tasks where CodeCity is most beneficial and for which type of users.

Subject ID	Age	Job Position	Experience Level				Number of Years			
			OOP	Java	Eclipse	Rev.Eng.	OOP	Java	Eclipse	Rev.Eng.
IA01	30	Developer	knowledgeable	advanced	knowledgeable	beginner	7–10	7–10	4–6	1–3
IA02	34	Developer	advanced	advanced	knowledgeable	knowledgeable	7–10	4–6	1–3	4–6
IA03	42	CTO, Developer	expert	knowledgeable	beginner	knowledgeable	>10	1–3	1–3	>10
IA04	37	Developer	advanced	advanced	knowledgeable	beginner	7–10	7–10	4–6	1–3
AB01	21	Master Student	advanced	advanced	advanced	beginner	4–6	4–6	4–6	<1
AB02	21	Master Student	advanced	advanced	advanced	beginner	1–3	1–3	1–3	<1
IA05	29	Consultant, Ph.D. Student	expert	beginner	beginner	knowledgeable	7–10	7–10	4–6	4–6
AA01	26	Ph.D. Student	expert	expert	knowledgeable	beginner	>10	>10	1–3	<1
AA02	26	Ph.D. Student	expert	advanced	knowledgeable	knowledgeable	4–6	1–3	1–3	1–3
IA06	35	Head of IT	expert	expert	advanced	advanced	>10	>10	4–6	7–10
IA07	27	Software Engineer	knowledgeable	knowledgeable	beginner	knowledgeable	7–10	7–10	1–3	4–6
IA08	25	Software Engineer	knowledgeable	advanced	knowledgeable	beginner	4–6	4–6	1–3	<1
IA09	32	Development Leader, Researcher	advanced	beginner	none	advanced	7–10	7–10	<1	4–6
AB03	28	Student	knowledgeable	knowledgeable	beginner	beginner	4–6	1–3	1–3	1–3
IA10	39	Project Manager	expert	advanced	knowledgeable	knowledgeable	>10	7–10	7–10	4–6
IA11	38	Consultant, System Manager/Analyst	knowledgeable	beginner	beginner	advanced	7–10	7–10	1–3	7–10
IA12	34	Senior Java Architect	expert	expert	advanced	advanced	>10	>10	>10	>10
AB04	22	Master Student	advanced	advanced	advanced	knowledgeable	4–6	1–3	1–3	<1
AA03	22	Master Student	advanced	advanced	advanced	beginner	7–10	4–6	4–6	<1
AB05	22	Master Student	advanced	advanced	knowledgeable	beginner	4–6	1–3	1–3	1–3
AA04	29	Ph.D. Student	advanced	advanced	knowledgeable	beginner	4–6	4–6	1–3	<1
IA13	32	Consultant	expert	knowledgeable	knowledgeable	expert	7–10	4–6	1–3	7–10
IA14	31	Software Architect	advanced	knowledgeable	knowledgeable	beginner	7–10	7–10	1–3	1–3
AB06	23	Master Student	advanced	advanced	advanced	beginner	4–6	1–3	1–3	<1
AB07	23	Master Student	advanced	advanced	advanced	beginner	4–6	1–3	1–3	<1
AA05	30	Ph.D. Student	advanced	advanced	advanced	knowledgeable	7–10	7–10	7–10	4–6
AA06	26	Ph.D. Student	expert	knowledgeable	advanced	expert	7–10	7–10	4–6	4–6
AA07	30	Ph.D. Student	advanced	advanced	knowledgeable	knowledgeable	7–10	7–10	1–3	1–3
IA15	40	Project Manager	expert	expert	advanced	advanced	>10	>10	7–10	4–6
IA16	39	Software Architect	advanced	advanced	knowledgeable	knowledgeable	4–6	4–6	4–6	1–3
IA01	30	Developer	knowledgeable	advanced	knowledgeable	beginner	7–10	7–10	4–6	1–3
IA17	27	Software Engineer	knowledgeable	advanced	knowledgeable	beginner	4–6	4–6	4–6	<1
IA19	39	Consultant, Project Manager, Architect	expert	expert	knowledgeable	advanced	>10	7–10	7–10	4–6
AB08	21	Master Student	advanced	advanced	advanced	beginner	1–3	1–3	1–3	<1
AB09	23	Ph.D. Student	advanced	advanced	advanced	knowledgeable	4–6	1–3	1–3	1–3
AA10	24	Ph.D. Student	advanced	advanced	advanced	advanced	4–6	4–6	4–6	1–3
AA11	23	Ph.D. Student	advanced	advanced	advanced	knowledgeable	4–6	4–6	4–6	1–3
AA12	52	Professor	expert	advanced	knowledgeable	advanced	>10	>10	4–6	>10
AA13	28	Ph.D. Student	advanced	advanced	knowledgeable	knowledgeable	4–6	4–6	4–6	1–3
AA14	24	Master Student	expert	expert	expert	knowledgeable	4–6	4–6	4–6	1–3
IA20	36	Developer	advanced	expert	advanced	beginner	>10	7–10	7–10	1–3

Table A.1. The subjects' personal information, clustered by treatment combinations

Subject ID		Treatment		Blocking Criteria	
	No.	Tool	System size	Background	Experience
IA01	1	CodeCity	large	industry	advanced
IA02	1	CodeCity	large	industry	advanced
IA03	1	CodeCity	large	industry	advanced
IA04	1	CodeCity	large	industry	advanced
AB01	1	CodeCity	large	academia	beginner
AB02	1	CodeCity	large	academia	beginner
IA05	1	CodeCity	large	industry	advanced
AA01	1	CodeCity	large	academia	advanced
AA02	1	CodeCity	large	academia	advanced
IA06	1	CodeCity	large	industry	advanced
IA07	2	CodeCity	medium	industry	advanced
IA08	2	CodeCity	medium	industry	advanced
IA09	2	CodeCity	medium	industry	advanced
AB03	2	CodeCity	medium	academia	beginner
IA10	2	CodeCity	medium	industry	advanced
IA11	2	CodeCity	medium	industry	advanced
IA12	2	CodeCity	medium	industry	advanced
AB04	2	CodeCity	medium	academia	beginner
AA03	2	CodeCity	medium	academia	advanced
AB05	2	CodeCity	medium	academia	beginner
AA04	2	CodeCity	medium	academia	advanced
IA13	2	CodeCity	medium	industry	advanced
IA14	3	Ecl+Excl	large	industry	advanced
AB06	3	Ecl+Excl	large	academia	beginner
AB07	3	Ecl+Excl	large	academia	beginner
AA05	3	Ecl+Excl	large	academia	advanced
AA06	3	Ecl+Excl	large	academia	advanced
AA07	3	Ecl+Excl	large	academia	advanced
IA15	3	Ecl+Excl	large	industry	advanced
IA16	3	Ecl+Excl	large	industry	advanced
IA01	4	Ecl+Excl	medium	industry	advanced
IA18	4	Ecl+Excl	medium	industry	advanced
IA19	4	Ecl+Excl	medium	industry	advanced
AB08	4	Ecl+Excl	medium	academia	beginner
AB09	4	Ecl+Excl	medium	academia	beginner
AA10	4	Ecl+Excl	medium	academia	advanced
AA11	4	Ecl+Excl	medium	academia	advanced
AA12	4	Ecl+Excl	medium	academia	advanced
AA13	4	Ecl+Excl	medium	academia	advanced
AA14	4	Ecl+Excl	medium	academia	advanced
IA20	4	Ecl+Excl	medium	industry	advanced

Table A.2. The assignment of the subjects to treatments and blocks

Subject ID				**Correctness Per Task**						**Total**	**Correctness**
	A1	**A2.1**	**A2.2**	**A3**	**A4.1**	***A4.2***	**B1.1**	**B1.2**	**B2.1**		***(excl. A4.2)***
IA01	0.00	1.00	1.00	1.00	1.00	*0.00*	0.00	1.00	1.00	6.00	6.00
IA02	1.00	0.80	0.50	1.00	1.00	*0.00*	0.00	1.00	1.00	6.30	6.30
IA03	0.00	0.00	1.00	0.00	1.00	*0.00*	0.00	1.00	0.00	3.00	3.00
IA04	0.00	1.00	0.00	1.00	1.00	*0.00*	0.00	1.00	1.00	5.00	5.00
AB01	0.00	1.00	1.00	0.80	1.00	*0.00*	0.00	1.00	1.00	5.80	5.80
AB02	0.00	1.00	1.00	0.70	1.00	*0.00*	0.00	0.00	1.00	4.70	4.70
IA05	0.00	1.00	1.00	0.20	1.00	*0.00*	0.00	1.00	1.00	5.20	5.20
AA01	0.00	0.80	1.00	0.40	1.00	*0.00*	0.00	1.00	0.00	4.20	4.20
AA02	0.00	1.00	1.00	0.00	1.00	*0.00*	1.00	1.00	1.00	6.00	6.00
IA06	0.00	1.00	0.00	0.30	1.00	*0.00*	0.00	1.00	1.00	4.30	4.30
IA07	1.00	1.00	1.00	0.17	1.00	*0.00*	0.00	1.00	1.00	6.17	6.17
IA08	1.00	0.50	1.00	0.83	1.00	*0.00*	1.00	1.00	1.00	7.33	7.33
IA09	1.00	1.00	1.00	1.00	1.00	*0.00*	1.00	1.00	1.00	8.00	8.00
AB03	1.00	1.00	1.00	1.00	1.00	*0.00*	1.00	1.00	1.00	8.00	8.00
IA10	1.00	1.00	1.00	0.17	1.00	*0.00*	0.00	1.00	1.00	6.17	6.17
IA11	1.00	1.00	1.00	0.00	1.00	*0.16*	0.00	0.00	1.00	5.16	5.00
IA12	1.00	1.00	1.00	1.00	1.00	*0.00*	1.00	1.00	1.00	8.00	8.00
AB04	1.00	1.00	1.00	0.67	1.00	*0.00*	0.00	1.00	1.00	6.67	6.67
AA03	1.00	1.00	0.80	0.33	1.00	*0.16*	0.00	1.00	1.00	6.29	6.13
AB05	1.00	1.00	1.00	1.00	1.00	*0.00*	0.00	1.00	1.00	7.00	7.00
AA04	1.00	1.00	1.00	0.50	1.00	*0.00*	0.00	1.00	1.00	6.50	6.50
IA13	1.00	1.00	1.00	0.83	1.00	*1.00*	0.00	0.00	1.00	6.83	5.83
IA14	0.00	0.80	0.33	0.30	1.00	*0.67*	0.00	1.00	1.00	5.10	4.43
AB06	1.00	1.00	1.00	0.00	0.00	*0.00*	0.00	1.00	0.00	4.00	4.00
AB07	0.00	0.60	0.67	0.00	0.00	*0.00*	0.00	0.00	1.00	2.27	2.27
AA05	1.00	1.00	1.00	0.00	1.00	*1.00*	0.00	1.00	1.00	7.00	6.00
AA06	0.00	1.00	0.00	0.00	1.00	*1.00*	0.00	0.00	1.00	4.00	3.00
AA07	1.00	1.00	0.00	0.00	1.00	*1.00*	0.00	1.00	0.00	5.00	4.00
IA15	1.00	0.80	0.00	0.00	1.00	*1.00*	0.00	1.00	0.00	4.80	3.80
IA16	0.00	0.00	0.67	0.00	1.00	*0.67*	0.00	1.00	1.00	4.34	3.67
IA01	1.00	1.00	1.00	0.50	1.00	*1.00*	0.00	1.00	1.00	7.50	6.50
IA18	1.00	1.00	1.00	0.50	1.00	*1.00*	0.00	1.00	1.00	7.50	6.50
IA19	0.00	1.00	0.60	0.00	1.00	*1.00*	0.00	1.00	0.00	4.60	3.60
AB08	0.00	1.00	0.40	0.38	1.00	*1.00*	1.00	1.00	1.00	6.78	5.78
AB09	1.00	1.00	0.60	0.50	1.00	*1.00*	0.00	1.00	1.00	7.10	6.10
AA10	1.00	1.00	0.80	0.38	1.00	*1.00*	0.00	1.00	1.00	7.18	6.18
AA11	1.00	1.00	0.80	0.00	1.00	*1.00*	0.00	1.00	1.00	6.80	5.80
AA12	0.00	0.50	0.00	0.00	1.00	*1.00*	0.00	1.00	1.00	4.50	3.50
AA13	0.00	1.00	0.00	0.25	1.00	*1.00*	0.00	1.00	1.00	5.25	4.25
AA14	1.00	1.00	0.60	0.00	1.00	*1.00*	0.90	1.00	1.00	7.50	6.50
IA20	0.00	1.00	1.00	0.38	1.00	*1.00*	0.00	1.00	1.00	6.38	5.38

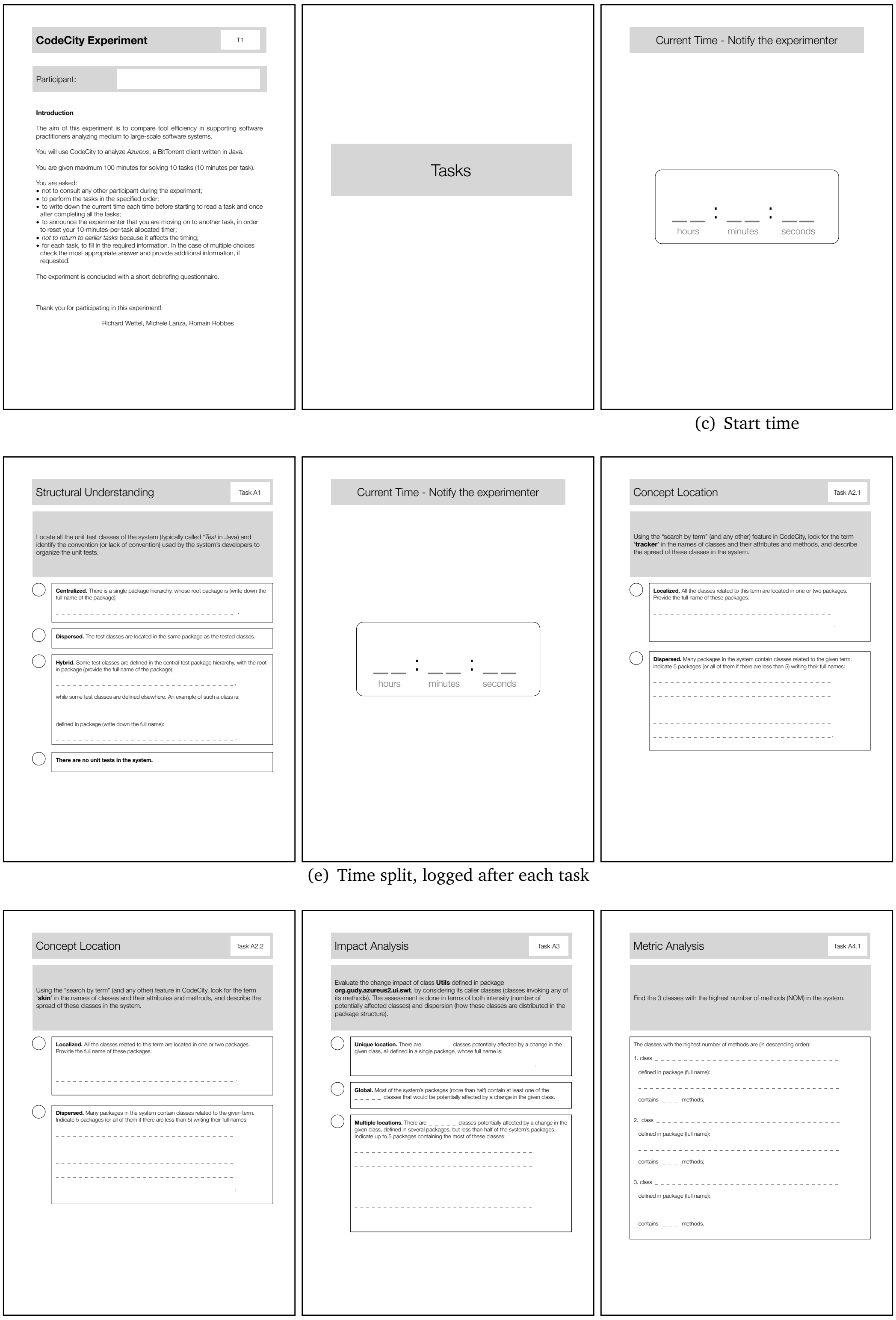
Table A.3. The *correctness* of the subjects' solutions to the tasks

Subject ID	Completion Time Per Task									Total	Compl. Time
	A1	A2.1	A2.2	A3	A4.1	*A4.2*	B1.1	B1.2	B2.1		*(excl. A4.2)*
IA01	8.80	4.32	5.38	10.00	7.28	*4.42*	9.85	1.90	1.02	52.97	48.55
IA02	2.42	6.08	1.75	8.58	5.33	*10.00*	5.25	2.92	2.33	44.67	34.67
IA03	6.25	6.92	2.42	9.43	5.62	*9.37*	2.55	2.95	1.83	47.33	37.97
IA04	9.08	6.00	4.00	6.08	7.92	*10.00*	10.00	1.33	1.08	55.50	45.50
AB01	1.33	4.83	3.92	4.75	6.25	*10.00*	6.92	2.25	2.58	42.83	32.83
AB02	10.00	5.58	1.67	4.25	8.08	*10.00*	9.42	2.58	1.67	53.25	43.25
IA05	6.33	3.67	1.33	4.75	3.50	*10.00*	3.75	1.58	2.17	37.08	27.08
AA01	9.67	4.42	2.08	8.75	3.17	*10.00*	5.17	1.33	1.42	46.00	36.00
AA02	8.33	7.17	2.50	10.00	5.00	*9.75*	8.33	3.25	1.33	55.67	45.92
IA06	10.00	5.92	4.42	10.00	4.17	*10.00*	7.33	1.33	1.50	54.67	44.67
IA07	7.25	5.67	6.25	6.33	4.58	*10.00*	2.42	1.42	2.33	46.25	36.25
IA08	2.67	2.25	2.95	4.55	3.75	*10.00*	3.33	4.00	1.17	34.67	24.67
IA09	10.00	3.67	3.00	10.00	5.25	*7.83*	3.67	2.50	1.42	47.33	39.50
AB03	6.33	2.00	9.33	5.00	5.00	*8.50*	7.17	1.50	1.25	46.08	37.58
IA10	3.25	3.67	7.33	6.83	3.83	*7.75*	3.08	1.75	8.08	45.58	37.83
IA11	3.83	2.40	5.92	7.17	4.00	*6.92*	7.42	2.83	1.75	42.23	35.32
IA12	3.75	2.67	4.17	5.58	5.17	*10.00*	5.75	2.00	1.25	40.33	30.33
AB04	2.67	3.92	2.58	3.67	5.58	*9.75*	2.00	0.50	3.83	34.50	24.75
AA03	2.50	3.17	3.75	10.00	3.17	*6.33*	4.08	1.25	2.58	36.83	30.50
AB05	5.50	4.67	4.33	5.58	5.83	*10.00*	9.58	1.83	1.67	49.00	39.00
AA04	7.08	3.67	5.17	6.33	3.50	*7.83*	6.00	1.25	2.83	43.67	35.83
IA13	3.00	4.67	2.33	4.75	3.25	*10.00*	4.83	1.67	2.08	36.58	26.58
IA14	6.67	9.00	2.42	10.00	4.50	*5.83*	5.33	1.83	4.17	49.75	43.92
AB06	5.67	5.75	2.05	6.95	10.00	*10.00*	10.00	3.85	2.00	56.27	46.27
AB07	8.75	9.33	4.67	10.00	6.83	*10.00*	10.00	3.67	2.67	65.92	55.92
AA05	7.00	6.08	3.75	8.42	5.25	*4.42*	10.00	3.17	6.67	54.75	50.33
AA06	6.83	2.00	3.67	6.58	3.25	*6.75*	5.33	1.58	1.58	37.58	30.83
AA07	3.67	3.67	2.50	2.17	2.92	*5.17*	2.75	3.33	1.83	28.00	22.83
IA15	9.75	8.83	5.08	10.00	4.33	*10.00*	5.08	1.75	7.83	62.67	52.67
IA16	10.00	7.42	4.33	9.50	4.00	*6.50*	8.50	2.00	4.50	56.75	50.25
IA01	2.55	3.90	4.38	9.20	4.03	*4.30*	9.92	2.10	2.93	43.32	39.02
IA18	5.28	5.13	4.58	9.82	4.72	*3.98*	9.77	4.13	3.12	50.53	46.55
IA19	3.33	3.83	4.50	3.50	3.83	*5.08*	6.58	1.92	5.58	38.17	33.08
AB08	6.08	1.08	10.00	8.83	3.50	*5.92*	9.42	3.58	1.92	50.33	44.42
AB09	5.83	4.83	4.33	10.00	3.42	*6.00*	2.00	1.92	1.50	39.83	33.83
AA10	3.17	6.08	5.08	7.33	8.00	*3.33*	4.83	1.17	2.33	41.33	38.00
AA11	6.17	4.08	6.08	3.83	3.50	*5.17*	3.67	1.67	2.92	37.08	31.92
AA12	6.75	4.75	3.92	4.75	5.25	*3.33*	4.42	1.42	3.00	37.58	34.25
AA13	7.00	7.00	10.00	10.00	5.92	*5.67*	6.33	1.42	5.42	58.75	53.08
AA14	6.33	1.50	3.33	10.00	2.83	*3.25*	9.17	2.42	4.50	43.33	40.08
IA20	6.83	5.58	4.75	8.00	2.33	*3.42*	2.42	1.42	1.33	36.08	32.67

Table A.4. The subjects' task *completion time*, in minutes

Subject ID	Difficulty Level Per Task										Time Pressure
	A1	A2.1	A2.2	A3	A4.1	*A4.2*	B1.1	B1.2	B2.1	B2.2	
IA01	difficult	intermediate	intermediate	simple	simple	impossible	difficult	trivial	trivial	intermediate	fair amount
IA02											fair amount
IA03	difficult	intermediate	intermediate	difficult	intermediate	difficult	simple	simple	simple	impossible	fair amount
IA04	simple	simple	simple	simple	intermediate	impossible	difficult	simple	trivial	difficult	too much
AB01	trivial	trivial	trivial	simple	simple	intermediate	difficult	simple	simple	intermediate	very little
AB02	trivial	trivial	trivial	trivial	simple	difficult	impossible	simple	trivial	simple	fair amount
IA05	simple	simple	simple	simple	intermediate	impossible	difficult	difficult	trivial	simple	not so much
AA01	intermediate	trivial	simple	intermediate	trivial	impossible	impossible	trivial	simple	trivial	very little
AA02	trivial	trivial	trivial	simple	simple	impossible	difficult	simple	trivial	intermediate	not so much
IA06	intermediate	simple	simple	simple	simple	intermediate	intermediate	simple	simple	difficult	fair amount
IA07	difficult	intermediate	intermediate	simple	simple	difficult	simple	simple	intermediate	intermediate	not so much
IA08	trivial	trivial	simple	intermediate	intermediate	impossible	difficult	intermediate	simple	intermediate	very little
IA09	difficult	simple	simple	intermediate	intermediate	difficult	intermediate	simple	trivial	difficult	not so much
AB03	simple	simple	simple	simple	simple	intermediate	intermediate	simple	simple	intermediate	very little
IA10	simple	simple	simple	simple	difficult	impossible	intermediate	intermediate	intermediate	difficult	not so much
IA11	simple	simple	simple	difficult	intermediate	difficult	difficult	simple	simple	difficult	fair amount
IA12	simple	simple	simple	intermediate	intermediate	difficult	trivial	simple	simple	difficult	none
AB04	trivial	trivial	trivial	simple	intermediate	difficult	trivial	trivial	simple	difficult	very little
AA03	trivial	trivial	trivial	difficult	trivial	impossible	difficult	trivial	trivial	simple	not so much
AB05	simple	simple	simple	intermediate	intermediate	impossible	difficult	intermediate	trivial	simple	not so much
AA04	intermediate	intermediate	difficult	difficult	intermediate	impossible	intermediate	intermediate	difficult	impossible	fair amount
IA13	trivial	simple	simple	simple	intermediate	difficult	intermediate	intermediate	intermediate	difficult	none
IA14	intermediate	intermediate	intermediate	intermediate	intermediate	intermediate	intermediate	intermediate	intermediate	intermediate	fair amount
AB06	simple	simple	simple	intermediate	intermediate	difficult	impossible	intermediate	intermediate	intermediate	fair amount
AB07	intermediate	simple	simple	impossible	trivial	difficult	simple	intermediate	intermediate	difficult	too much
AA05	simple	simple	simple	intermediate	intermediate	intermediate	difficult	intermediate	intermediate	intermediate	fair amount
AA06	intermediate	simple	simple	difficult	trivial	intermediate	simple	simple	intermediate	difficult	very little
AA07	trivial	intermediate	intermediate	trivial	trivial	simple	intermediate	trivial	simple	intermediate	none
IA15	difficult	intermediate	intermediate	difficult	simple	simple	intermediate	intermediate	impossible	impossible	fair amount
IA16	intermediate	intermediate	intermediate	impossible	simple	simple	intermediate	simple	intermediate	difficult	fair amount
IA01	intermediate	intermediate	intermediate	difficult	trivial	simple	impossible	trivial	simple	difficult	not so much
IA18	intermediate	simple	intermediate	difficult	simple	simple	difficult	intermediate	simple	difficult	fair amount
IA19	simple	simple	intermediate	simple	trivial	simple	simple	simple	intermediate	impossible	fair amount
AB08	intermediate	intermediate	simple	intermediate	trivial	trivial	difficult	trivial	simple	intermediate	not so much
AB09	trivial	simple	simple	impossible	simple	simple	intermediate	intermediate	simple	difficult	fair amount
AA10	simple	intermediate	intermediate	intermediate	trivial	trivial	simple	simple	intermediate	impossible	very little
AA11	difficult	simple	intermediate	intermediate	simple	simple	simple	intermediate	intermediate	impossible	not so much
AA12	simple	simple	simple	simple	simple	intermediate	simple	simple	intermediate	difficult	not so much
AA13	simple	intermediate	intermediate	difficult	simple	simple	simple	simple	simple	intermediate	not so much
AA14	trivial	trivial	trivial	impossible	trivial	trivial	difficult	trivial	difficult	impossible	not so much
IA20	intermediate	simple	simple	intermediate	trivial	trivial	trivial	trivial	trivial	difficult	not so much

Table A.5. The subjects' perceived *time pressure* and task *difficulty*



CodeCity Experiment T1

Participant:

Introduction

The aim of this experiment is to compare tool efficiency in supporting software practitioners analyzing medium to large-scale software systems.

You will use CodeCity to analyze *Azureus*, a BitTorrent client written in Java.

You are given maximum 100 minutes for solving 10 tasks (10 minutes per task).

You are asked:
- not to consult any other participant during the experiment;
- to perform the tasks in the specified order;
- to write down the current time each time before starting to read a task and once after completing all the tasks;
- to announce the experimenter that you are moving on to another task, in order to reset your 10-minutes-per-task allocated timer;
- *not to return to earlier tasks* because it affects the timing;
- for each task, to fill in the required information. In the case of multiple choices check the most appropriate answer and provide additional information, if requested.

The experiment is concluded with a short debriefing questionnaire.

Thank you for participating in this experiment!

Richard Wettel, Michele Lanza, Romain Robbes

Tasks

Current Time - Notify the experimenter

_ _ : _ _ : _ _
hours minutes seconds

(c) Start time

Structural Understanding Task A1

Locate all the unit test classes of the system (typically called **Test* in Java) and identify the convention (or lack of convention) used by the system's developers to organize the unit tests.

○ **Centralized.** There is a single package hierarchy, whose root package is (write down the full name of the package):

_ .

○ **Dispersed.** The test classes are located in the same package as the tested classes.

○ **Hybrid.** Some test classes are defined in the central test package hierarchy, with the root in package (provide the full name of the package):

_ ,

while some test classes are defined elsewhere. An example of such a class is:

_ _

defined in package (write down the full name):

_ .

○ **There are no unit tests in the system.**

Current Time - Notify the experimenter

_ _ : _ _ : _ _
hours minutes seconds

Concept Location Task A2.1

Using the "search by term" (and any other) feature in CodeCity, look for the term '**tracker**' in the names of classes and their attributes and methods, and describe the spread of these classes in the system.

○ **Localized.** All the classes related to this term are located in one or two packages. Provide the full name of these packages:

_ _
_ .

○ **Dispersed.** Many packages in the system contain classes related to the given term. Indicate 5 packages (or all of them if there are less than 5) writing their full names:

_ _
_ _
_ _
_ _
_ _.

(e) Time split, logged after each task

Concept Location Task A2.2

Using the "search by term" (and any other) feature in CodeCity, look for the term '**skin**' in the names of classes and their attributes and methods, and describe the spread of these classes in the system.

○ **Localized.** All the classes related to this term are located in one or two packages. Provide the full name of these packages:

_ _
_ .

○ **Dispersed.** Many packages in the system contain classes related to the given term. Indicate 5 packages (or all of them if there are less than 5) writing their full names:

_ _
_ _
_ _
_ _
_ _.

Impact Analysis Task A3

Evaluate the change impact of class **Utils** defined in package **org.gudy.azureus2.ui.swt**, by considering its caller classes (classes invoking any of its methods). The assessment is done in terms of both intensity (number of potentially affected classes) and dispersion (how these classes are distributed in the package structure).

○ **Unique location.** There are _ _ _ _ _ classes potentially affected by a change in the given class, all defined in a single package, whose full name is:

_ .

○ **Global.** Most of the system's packages (more than half) contain at least one of the _ _ _ _ _ classes that would be potentially affected by a change in the given class.

○ **Multiple locations.** There are _ _ _ _ _ classes potentially affected by a change in the given class, defined in several packages, but less than half of the system's packages. Indicate up to 5 packages containing the most of these classes:

_ _
_ _
_ _
_ _
_ _

Metric Analysis Task A4.1

Find the 3 classes with the highest number of methods (NOM) in the system.

The classes with the highest number of methods are (in descending order):

1. class _

defined in package (full name):

_ _

contains _ _ _ methods;

2. class _

defined in package (full name):

_ _

contains _ _ _ methods;

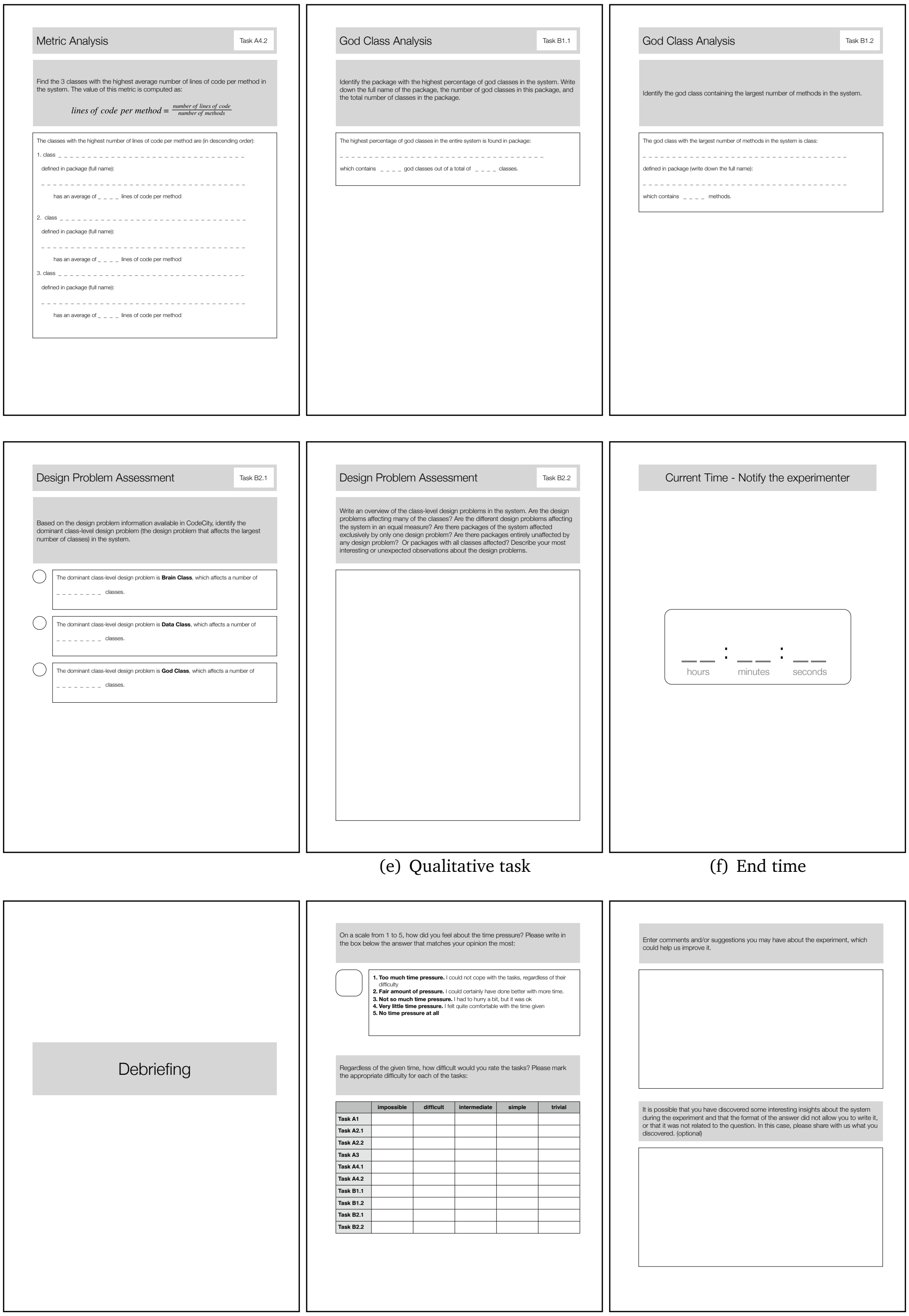
3. class _

defined in package (full name):

_ _

contains _ _ _ methods.

Figure A.2. Handout for Treatment 1 (Part 1 of 2)



Metric Analysis

Task A4.2

Find the 3 classes with the highest average number of lines of code per method in the system. The value of this metric is computed as:

$$lines\ of\ code\ per\ method = \frac{number\ of\ lines\ of\ code}{number\ of\ methods}$$

The classes with the highest number of lines of code per method are (in descending order):

1. class _

defined in package (full name):

_ _

has an average of _ _ _ _ lines of code per method

2. class _

defined in package (full name):

_ _

has an average of _ _ _ _ lines of code per method

3. class _

defined in package (full name):

_ _

has an average of _ _ _ _ lines of code per method

God Class Analysis

Task B1.1

Identify the package with the highest percentage of god classes in the system. Write down the full name of the package, the number of god classes in this package, and the total number of classes in the package.

The highest percentage of god classes in the entire system is found in package:

_ _

which contains _ _ _ _ god classes out of a total of _ _ _ _ classes.

God Class Analysis

Task B1.2

Identify the god class containing the largest number of methods in the system.

The god class with the largest number of methods in the system is class:

_ _

defined in package (write down the full name):

_ _

which contains _ _ _ _ methods.

Design Problem Assessment

Task B2.1

Based on the design problem information available in CodeCity, identify the dominant class-level design problem (the design problem that affects the largest number of classes) in the system.

○ The dominant class-level design problem is **Brain Class**, which affects a number of _ _ _ _ _ _ _ _ classes.

○ The dominant class-level design problem is **Data Class**, which affects a number of _ _ _ _ _ _ _ _ classes.

○ The dominant class-level design problem is **God Class**, which affects a number of _ _ _ _ _ _ _ _ classes.

Design Problem Assessment

Task B2.2

Write an overview of the class-level design problems in the system. Are the design problems affecting many of the classes? Are the different design problems affecting the system in an equal measure? Are there packages of the system affected exclusively by only one design problem? Are there packages entirely unaffected by any design problem? Or packages with all classes affected? Describe your most interesting or unexpected observations about the design problems.

(e) Qualitative task

Current Time - Notify the experimenter

_ _ : _ _ : _ _

hours minutes seconds

(f) End time

Debriefing

On a scale from 1 to 5, how did you feel about the time pressure? Please write in the box below the answer that matches your opinion the most:

1. **Too much time pressure.** I could not cope with the tasks, regardless of their difficulty
2. **Fair amount of pressure.** I could certainly have done better with more time.
3. **Not so much time pressure.** I had to hurry a bit, but it was ok
4. **Very little time pressure.** I felt quite comfortable with the time given
5. **No time pressure at all**

Regardless of the given time, how difficult would you rate the tasks? Please mark the appropriate difficulty for each of the tasks:

	impossible	difficult	intermediate	simple	trivial
Task A1					
Task A2.1					
Task A2.2					
Task A3					
Task A4.1					
Task A4.2					
Task B1.1					
Task B1.2					
Task B2.1					
Task B2.2					

Enter comments and/or suggestions you may have about the experiment, which could help us improve it.

It is possible that you have discovered some interesting insights about the system during the experiment and that the format of the answer did not allow you to write it, or that it was not related to the question. In this case, please share with us what you discovered. (optional)

Figure A.3. Handout for Treatment 1 (Part 2 of 2)

Part VI

Bibliography

Bibliography

[ABHL06] Erik Arisholm, Lionel C. Briand, Siw Elisabeth Hove, and Yvan Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.

[ABW$^+$09] Sazzadul Alam, Sandro Boccuzzo, Richard Wettel, Philippe Dugerdil, Harald Gall, and Michele Lanza. EvoSpaces - multi-dimensional navigation spaces for software evolution. In *Human Machine Interaction*, LNCS, pages 167–192. Springer, 2009.

[AD07] Sazzadul Alam and Philippe Dugerdil. EvoSpaces: 3D visualization of software architecture. In *SEKE '07: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering*, pages 500–505. IEEE Computer Society Press, 2007.

[AHK$^+$01] Atsushi Aoki, Kaoru Hayashi, Kouichi Kishida, Kumiyo Nakakoji, Yoshiyuki Nishinaka, Brent Reeves, Akio Takashima, and Yasuhiro Yamamoto. A case study of the evolution of Jun: an object-oriented open-source 3D multimedia library. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 524–533. IEEE Computer Society Press, 2001.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[AMNB07] Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. X3D software visualization. In *NZCSRSC '07: Proceedings of the New Zealand Computer Science Research Students Conference*, 2007.

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information pyramids: A new approach to visualising large hierarchies. In *VIS '97: Proceedings of the 8th conference on Visualization*, pages 49–52. IEEE Computer Society Press, 1997.

[Bae98] Ronald M. Baecker. Sorting out sorting: A case study of software visualization for teaching computer science. In John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 369–381. MIT Press, 1998.

[BD04] Michael Balzer and Oliver Deussen. Hierarchy based 3D visualization of large software structures. In *VIS '04: Proceedings of the 15th Conference on Visualization, Poster Session*, page 4. IEEE Computer Society Press, 2004.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[Ber67] Jacques Bertin. *Sémiologie graphique*. Mouton, 1967.

[BG07] Sandro Boccuzzo and Harald C. Gall. CocoViz: Towards cognitive software visualizations. In *VISSOFT '07: Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 72–79. IEEE Computer Society Press, 2007.

[BG08] Sandro Boccuzzo and Harald Gall. Software visualization with audio supported cognitive glyphs. In *ICSM '08: Proceedings of the 24th IEEE International Conference on Software Maintenance*, pages 366–375. IEEE Computer Society Press, 2008.

[Bia08] Andrea Biaggi. Citylyzer - a 3D visualization plug-in for eclipse. Bachelor's thesis, University of Lugano, June 2008.

[BK95] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *SSR '95: Proceedings of the 1995 ACM Symposium on Software Reusability*, pages 259–262. ACM Press, 1995.

[BM86] R. Baecker and A. Marcus. Design principles for the enhanced presentation of computer program source text. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 51–58. ACM Press, 1986.

[BM89] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. ACM Press, 1989.

[BNDL04] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *VisSym '04: Symposium on Visualization*, pages 261–266. Eurographics Association, 2004.

[Boo09] Grady Booch. Like a river. *IEEE Software*, 26(3):10–11, 2009.

[Bro88] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, 1988.

[BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *ACM SIGGRAPH Computer Graphics*, 18(3):177–186, 1984.

[CCI90] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CHZ+07] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *ICPC '07: Proceedings of the 15th International Conference on Program Comprehension*, pages 49–58. IEEE Computer Society Press, 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, 1994.

[Cor89] T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[CR03] Alan Cooper and Robert Reimann. *About Face 2.0 - The Essentials of Interaction Design*. Wiley, 2003.

[CS70] Kenneth Conrow and Ronald G. Smith. NEATER2: a PL/I source statement reformatter. *Communications of the ACM*, 13(11):669–675, 1970.

[CZRvD09] Bas Cornelissen, Andy Zaidman, Bart Van Rompaey, and Arie van Deursen. Trace visualization for program comprehension: A controlled experiment. In *ICPC '09: Proceedings of the 17th IEEE International Conference on Program Comprehension*, pages 100–109. IEEE Computer Society Press, 2009.

[CZvDVR09] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart Van Rompaey. Trace visualization for program comprehension: A controlled experiment. Technical Report TUD-SERG-2009-001, Delft University of Technology, 2009.

[DA08] Philippe Dugerdil and Sazzadul Alam. Execution trace visualization in a 3D space. In *ITGN '08: Proceedings of the 5th International Conference on Information Technology: New Generations*, pages 38–43. IEEE Computer Society Press, 2008.

[Dak09] Ermira Daka. Parsing and modeling C# systems. Master's thesis, University of Lugano, June 2009.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering approach combining metrics and program visualisation. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DeL05] Robert DeLine. Staying oriented with software terrain maps. In *DMS '05: Proceedings of the 11th International Conference on Distributed Multimedia Systems*, pages 309–314. Knowledge Systems Institute, 2005.

[DGKR09] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, February 2009.

[Die02] Stephan Diehl, editor. *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, Lecture Notes in Computer Science. Springer, 2002.

[Die07] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[DL06a] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198. IEEE Computer Society Press, 2006.

[DL06b] Marco D'Ambros and Michele Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *CSMR '06: Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, pages 227–236. IEEE Computer Society Press, 2006.

[DL10] Marco D'Ambros and Michele Lanza. Distributed and collaborative software evolution analysis with Churrasco. *Journal of Science of Computer Programming (SCP)*, 75(4):276–287, April 2010.

[DLG05] Marco D'Ambros, Michele Lanza, and Harald Gall. Fractal figures: Visualizing development effort for CVS entities. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 46–51. IEEE Computer Society Press, 2005.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[ED06] Geoffrey Ellis and Alan Dix. An explorative analysis of user evaluation studies in information visualisation. In *BELIV '06: Proceedings of the 2006 AVI workshop on BEyond time and errors*, pages 1–7. ACM Press, 2006.

[EGK$^+$01] Stephen Eick, Todd Graves, Alan Karr, J. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[ESEE92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[Fav01] Jean-Marie Favre. GSEE: A generic software exploration environment. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE Computer Society Press, 2001.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Few04] Stephen Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 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.

[Fur86] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23. ACM Press, 1986.

[Gab96] Richard P. Gabriel. *Patterns of Software*. Oxford University Press, 1996.

[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, October 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gîr05] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, November 2005.

[GJR99] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *ICSM '99: Proceedings of the 15th IEEE International Conference on Software Maintenance*, pages 99–108. IEEE Computer Society Press, 1999.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 113–122. IEEE Computer Society Press, 2005.

[GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *CSMR '05: Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pages 2–11. IEEE Computer Society Press, 2005.

[GLW06] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3D. In *SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 47–56. ACM Press, 2006.

[GME05] Denis Gracanin, Kresimir Matkovic, and Mohamed Eltoweissy. Software visualization. *Innovations Syst. Softw. Eng.*, 1(2):221–230, 2005.

[GYB04] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A solar system metaphor for 3D visualisation of object oriented software metrics. In *APVis '04: Proceedings of the 2004 Australasian symposium on Information Visualisation*, pages 53–59. Australian Computer Society, Inc., 2004.

[Hai59] Lois M. Haibt. A program to draw multilevel flow charts. In *Proceedings of the Western Joint Computer Conference*, pages 131–137. ACM Press, 1959.

[HL77] Jon Hueras and Henry Ledgard. An automatic formatting program for PASCAL. *ACM SIGPLAN Notices*, 12(7):82–84, 1977.

[HM08] Sonia Haiduc and Andrian Marcus. On the use of domain terms in source code. In *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 113–122. IEEE Computer Society Press, 2008.

[Hol06] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):741–748, 2006.

[HP96] Richard Holt and Jason Pak. GASE: Visualizing software evolution-in-the-large. In *WCRE '96: Proceedings of the Third Working Conference on Reverse Engineering*, pages 163–167. IEEE Computer Society Press, 1996.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[IEE90] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: Proceedings of the 7th Annual Conference on Object Oriented Programming Systems Languages and Applications*, pages 63–76. ACM Press, 1992.

[JS03] Juanjuan Jiang and Tarja Systä. Exploring differences in exchange formats - tool support and case studies. In *CSMR '03: Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 389–398. IEEE Computer Society Press, 2003.

[KDJ04] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 273–281. IEEE Computer Society Press, 2004.

[KG88] Michael F. Kleyn and Paul C. Gingrich. Graphtrace–understanding object-oriented systems using concurrently animated views. *SIGPLAN Not.*, 23(11):191–205, 1988.

[KHI$^+$03] R. Kosara, C.G. Healey, V. Interrante, D.H. Laidlaw, and C. Ware. User studies: Why, how, and when? *IEEE Computer Graphics and Applications*, 23(4):20–25, July-Aug. 2003.

[KLN08] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*, pages 209–218. IEEE Computer Society Press, 2008.

[KM00] Claire Knight and Malcolm C. Munro. Virtual but visible software. In *IV '00: Proceedings of the International Conference on Information Visualisation*, pages 198–205. IEEE Computer Society Press, 2000.

[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.

[Kno66a] Kenneth C. Knowlton. L6: Bell Telephone Laboratories low-level linked list language. 16-minute black-and-white film, 1966.

[Kno66b] Kenneth C. Knowlton. L6: Part II. an example of L6 programming. 30-minute black-and-white film, 1966.

[Knu63] Donald E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6(9):555–563, 1963.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97—111, 1984.

[Knu02] Donald Knuth. All questions answered. *Notices of the AMS*, 49(3):318–324, 2002.

[Kob01] Alfred Kobsa. An empirical comparison of three commercial information visualization systems. In *InfoVis '01: Proceedings of the 2001 IEEE Symposium on Information Visualization*, pages 123–130. IEEE Computer Society Press, 2001.

[Kob04] Alfred Kobsa. User experiments with tree visualization systems. In *InfoVis '04: Proceedings of the 2004 IEEE Symposium on Information Visualization*, pages 9–16. IEEE Computer Society Press, 2004.

[Kos03] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[KPP$^+$02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[Lan99] Michele Lanza. Combining metrics and graphs for object-oriented reverse engineering. Master's thesis, University of Berne, Switzerland, 1999.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42. ACM Press, 2001.

[Lan03] Michele Lanza. CodeCrawler — lessons learned in building a software visualization tool. In *CSMR '03: Proceedings of the 7th European Conference on Software Maintenance and Reengineering*, pages 409–418. IEEE Computer Society Press, 2003.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.

[LC07] Christian F. J. Lange and Michel R. V. Chaudron. Interactive views to improve the comprehension of UML models - an experimental validation. In *ICPC '07: Proceedings of the 15th International Conference on Program Comprehension*, pages 221–230. IEEE Computer Society Press, 2007.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[Lev60] Howard Levene. Robust tests for equality of variances. In Ingram Olkin, editor, *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, pages 278–292. Stanford University Press, 1960.

[LGD09] Michele Lanza, Harald Gall, and Philippe Dugerdil. EvoSpaces: Multi-dimensional navigation spaces for software evolution. In *CSMR '09: Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 293–296. IEEE Computer Society Press, 2009.

[LJ80] George Lakoff and Mark Johnson. *Metaphors We Live By*. University of Chicago Press, 1980.

[LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 91–100. IEEE Computer Society Press, 2007.

[LLGH07] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Reinout Heeck. Reverse engineering super-repositories. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 120–129. IEEE Computer Society Press, 2007.

[LLGR10] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The Small Project Observatory: Visualizing software ecosystems. *Journal of Science of Computer Programming (SCP)*, 75(4):264–275, April 2010.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.

[LMSW03] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: Experiences integrating a visualization tool with Eclipse. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 47–56. ACM Press, 2003.

[LN00] George Lakoff and Rafael E. Núñez. *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics into Being*. Basic Books, 2000.

[LS81] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[LSP05] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223. ACM Press, 2005.

[LSP08] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Exploring the evolution of software quality with animated visualization. In *VL/HCC '08: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 13–20. IEEE Computer Society Press, 2008.

[Lun09] Mircea Lungu. *Reverse Engineering Software Ecosystems*. Phd thesis, University of Lugano, October 2009.

[Mac99] Lindsay W. MacDonald. Tutorial: Using color effectively in computer graphics. *IEEE Computer Graphics and Applications*, 19:20–35, 1999.

[Mal07] Jacopo Malnati. X-Ray - an Eclipse plug-in for software visualization. Bachelor's thesis, University of Lugano, June 2007.

[Mar04a] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[Mar04b] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2004.

[McC76] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[McK84] James R. McKee. Maintenance as a function of design. In *AFIPS '84: Proceedings of the July 9-12, 1984, National Computer Conference and Exposition*, pages 187–193. ACM Press, 1984.

[MCS05] Andrian Marcus, Denise Comorski, and Andrey Sergeyev. Supporting the evolution of a software visualization tool through usability studies. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 307–316. IEEE Computer Society Press, 2005.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 27–36. ACM Press, 2003.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 135–144. ACM Press, 2006.

[MK88] H.A. Muller and K. Klashinsky. Rigi: a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[ML05] Cédric Mesnage and Michele Lanza. White Coats: Web-visualization of evolving software in 3D. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 40–45. IEEE Computer Society Press, 2005.

[MLMD01] J. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object-oriented software in Virtual Reality. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 26–35. IEEE Computer Society Press, 2001.

[MM03] Jonathan Maletic and Andrian Marcus. CFB: A call for benchmarks - for software visualization. In *VISSOFT '03: Proceedings of the 2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society Press, 2003.

[MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael Collard. A task oriented view of software visualization. In *VISSOFT '02: Proceedings of the 1st IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE Computer Society Press, 2002.

[MMM$^+$05] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, Daniel Raţiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, Industrial and Tool Volume*, pages 77–80. IEEE Computer Society Press, 2005.

[MRB$^+$05] Andrian Marcus, Václav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42. IEEE Computer Society Press, 2005.

[Mül86] Hausi A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[Mye83] Brad A. Myers. Incense: A system for displaying data structures. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 115–125. ACM Press, 1983.

[Mye86] Brad A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *SIGCHI Bulletin*, 17(4):59–66, 1986.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–10. ACM Press, 2005.

[NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, 1973.

[ODB06] Richard O'Donnell, Alan Dix, and Linden J. Ball. Exploring the PieTree for representing numerical hierarchical data. In *HCI '06: Proceedings of International Workshop on Human-Computer Interaction*, pages 239–254. Springer, 2006.

[PAA08] Wim De Pauw, Henrique Andrade, and Lisa Amini. Streamsight: a visualization tool for large-scale streaming applications. In *SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization*, pages 125–134. ACM Press, 2008.

[PBG03] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3D metaphor for software production visualization. In *IV '03: Proceedings of the Seventh International Conference on Information Visualization*, pages 314–319. IEEE Computer Society Press, 2003.

[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.

[PEQ$^+$07] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating software architecture using a unified single-view visualization. In *ICECCS '07: Proceedings of 12th the IEEE International Conference on Engineering Complex Computer Systems*, pages 217–228. IEEE Computer Society Press, 2007.

[Pet95] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 67–75. ACM Press, 2005.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA '93: Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications*, pages 326–337. ACM Press, 1993.

[Pin05] Martin Pinzger. *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.

[Pla86] William Playfair. *The Commercial and Political Atlas*. Corry, London, 1786.

[Pla04] Catherine Plaisant. The challenge of information visualization evaluation. In *AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–116. ACM Press, 2004.

[PLL05] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualizations with Vizz3D. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 173–182. ACM Press, 2005.

[PRW04] Michael J. Pacione, Marc Roper, and Murray Wood. A novel software visualisation model to support software comprehension. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 70–79. IEEE Computer Society Press, 2004.

[PSK07] Massimiliano Di Penta, R.E.K. Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *ICPC '07: Proceedings of the 15th International Conference on Program Comprehension*, pages 281–285. IEEE Computer Society Press, 2007.

[Qua08] Jochen Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 73–82. IEEE Computer Society Press, 2008.

[RC93] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.

[RDGM04] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *CSMR '04: Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 223–232. IEEE Computer Society Press, 2004.

[Rei84] Steven P. Reiss. Pecan: Program development systems that support multiple views. In *ICSE '84: Proceedings of the 7th International Conference on Software Engineering*, pages 324–333. IEEE Computer Society Press, 1984.

[Rei95] Steven P. Reiss. An engine for the 3D visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995.

[RFG05] J. Ratzinger, M. Fischer, and H. Gall. EvoLens: lens-view visualizations of evolution data. In *IWPSE '05: Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 103–112. IEEE Computer Society Press, 2005.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: animated 3D visualizations of hierarchical information. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM Press, 1991.

[RMJ09] Daniel Raţiu, Radu Marinescu, and Jan Jürjens. The logical modularity of programs. In *WCRE '09: Proceedings of the 16th Working Conference on Reverse Engineering*, pages 123–127. IEEE Computer Society Press, 2009.

[SB99] Matthew L. Staples and James M. Bieman. 3-D visualization of software structure. *Advances in Computers*, 49:96–143, 1999.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.

[SFM99] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw.*, 44(3):171–185, 1999.

[Shn92] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343. IEEE Computer Society Press, 1996.

[SM95] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *ICSM '95: Proceedings of the 11th IEEE International Conference on Software Maintenance*, pages 275–284. IEEE Computer Society Press, 1995.

[SMDV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 23–34. ACM Press, 2006.

[SMW96] M.-A. D. Storey, H.A. Müller, and K. Wong. Manipulating and documenting software structures. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 244–263. World Scientific Publishing Co., 1996.

[SOT09] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Evaluation of software visualization tools: Lessons learned. In *VISSOFT '09: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2009.

[Sta00] John Stasko. An evaluation of space-filling information visualizations for depicting hierarchical structures. *Int. J. Hum.-Comput. Stud.*, 53(5):663–694, 2000.

[Sto06] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Control*, 14(3):187–208, 2006.

[SW65] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3–4):591–611, 1965.

[SWM97] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society Press, 1997.

[TA08] Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, May 2008.

[TC09] A.R. Teyseyre and M.R. Campo. An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, Jan.-Feb. 2009.

[Tei85] W. Teitelman. A tour through Cedar. *IEEE Transactions on Software Engineering*, 11:285–302, 1985.

[TM02] Christopher Taylor and Malcolm Munro. Revision towers. In *VISSOFT '02: Proceedings of the 1st IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society Press, 2002.

[Tuf90] Edward Tufte. *Envisioning Information*. Graphics Press, 1990.

[Tuf97] Edward Tufte. *Visual Explanations*. Graphics Press, 1997.

[Tuf01] Edward Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[VLT07] Lucian Voinea, Johan Lukkien, and Alexandru Telea. Visual assessment of software evolution. *Science of Computer Programming*, 65(3):222–248, 2007.

[War04] Colin Ware. *Information visualization: perception for design*. Morgan Kaufmann Publishers Inc., 2 edition, 2004.

[WCJ98] U. Wiss, D. Carr, and H. Jonsson. Evaluating three-dimensional information visualization designs: A case study of three designs. In *IV '98: Proceedings of the International Conference on Information Visualisation*, pages 137–145. IEEE Computer Society Press, 1998.

[Wei98] Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, 1998.

[Wet08] Richard Wettel. Scripting 3D visualizations with CodeCity. In *FAMOOSr '08: Proceedings of the 2nd Workshop on FAMIX and Moose in Reengineering*, 2008.

[WHH04] J. Wu, A. Hassan, and R. Holt. Exploring software evolution using spectrographs. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89. IEEE Computer Society Press, 2004.

[WL07a] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *ICPC '07: Proceedings of the 15th International Conference on Program Comprehension*, pages 231–240. IEEE Computer Society Press, 2007.

[WL07b] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *VISSOFT '07: Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE Computer Society Press, 2007.

[WL08a] Richard Wettel and Michele Lanza. CodeCity. In *WASDeTT '08: In Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[WL08b] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering, Tool Demo*, pages 921–922. ACM Press, 2008.

[WL08c] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*, pages 219–228. IEEE Computer Society Press, 2008.

[WL08d] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization*, pages 155–164. ACM Press, 2008.

[WLR10] Richard Wettel, Michele Lanza, and Romain Robbes. Empirical validation of CodeCity: A controlled experiment. Technical Report 2010/05, University of Lugano, June 2010.

[WRH$^+$00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.

[XPM06] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. Visualization of CVS repository information. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 231–242. IEEE Computer Society Press, 2006.

[YM98] P. Young and M. Munro. Visualizing software in Virtual Reality. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 19–27. IEEE Computer Society Press, 1998.

[Zha03] Kang Zhang, editor. *Software Visualization: From Theory to Practice*. Kluwer Academic Publishers, 2003.

[Zhu07] Ying Zhu. Measuring effective data visualization. In *ISVC (2)*, volume 4842 of *Lecture Notes in Computer Science*, pages 652–661. Springer, 2007.

[ZSG79] Marvin Zelkowitz, Alan Shaw, and John Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.