Software Systems as Cities: A Controlled Experiment

Richard Wettel and Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano
{richard.wettel,michele.lanza}@usi.ch

Romain Robbes
PLEIAD @ DCC
University of Chile
rrobbes@dcc.uchile.cl

ABSTRACT

Software visualization is a popular program comprehension technique used in the context of software maintenance, reverse engineering, and software evolution analysis. While there is a broad range of software visualization approaches, only few have been empirically evaluated. This is detrimental to the acceptance of software visualization in both the academic and the industrial world.

We present a controlled experiment for the empirical evaluation of a 3D software visualization approach based on a city metaphor and implemented in a tool called CodeCity. The goal is to provide experimental evidence of the viability of our approach in the context of program comprehension by having subjects perform tasks related to program comprehension. We designed our experiment based on lessons extracted from the current body of research. We conducted the experiment in four locations across three countries, involving 41 participants from both academia and industry. The experiment shows that CodeCity leads to a statistically significant increase in terms of task correctness and decrease in task completion time. We detail the experiment we performed, discuss its results and reflect on the many lessons learned.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Experimentation, Human Factors, Measurement

Keywords

Software visualization, Empirical validation

1. INTRODUCTION

Software visualization is defined by Stasko et al. 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"* [13].

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

It has earned a reputation as an effective program comprehension technique, and is widely used in the context of software maintenance, reverse engineering and reengineering [2].

The last decade has witnessed a proliferation of software visualization approaches, aimed at supporting a broad range of software engineering activities. As a consequence, there is a growing need for objective assessments of visualization approaches to demonstrate their effectiveness. Unfortunately, only few software visualization approaches have been empirically validated so far, which is detrimental to the development of the field [2].

One of the reasons behind the shortage of empirical evaluation lies in the difficulty of performing a controlled experiment for software visualization. On the one hand, the variety of software visualization approaches makes it almost impossible to reuse the design of an experiment from the current body of research. On the other hand, organizing and conducting a proper controlled experiment is, in itself, a difficult endeavor, which can fail in many ways: data which does not support a true hypothesis, conclusions which are not statistically significant due to insufficient data points, etc.

We present a controlled experiment for the empirical evaluation of a software visualization approach based on a city metaphor [15]. The aim is to show that our approach, implemented in a tool called CodeCity, is at least as effective and efficient as the state of the practice at supporting reverse engineering and program comprehension activities. We conceived a set of tasks and measured both the correctness of the task solutions and the task completion time. We conducted the experiment in four locations across three countries, with participants from both industry and academia, with a broad range of experience.

In this paper we make the following two major contributions:

1. we detail the experiment design and its operation, reporting on a number of lessons learned regarding the many pitfalls that this type of experiment entails, and

2. we discuss the results of the experiment, which show that our approach is a viable alternative to existing non-visual techniques.

Structure of the paper. In Section 2 we describe our software visualization approach. In Section 3 we present the related work, followed by a list of *desiderata* in Section 4, extracted from an extensive study of the existing body of research. In Section 5 we describe the design of our experiment and in Section 6 we detail the experiment's operational phase. In Section 7 we present how we collected the data on which we performed the analysis presented in Section 8. In Section 9 we present the results of the experiment, followed by a presentation of the threats to validity in Section 10, and the conclusions in Section 11.

2. CODECITY IN A NUTSHELL

Our approach [15, 18] uses a city metaphor to depict software systems as three-dimensional cities (see Figure 1), where classes are buildings and packages are districts.

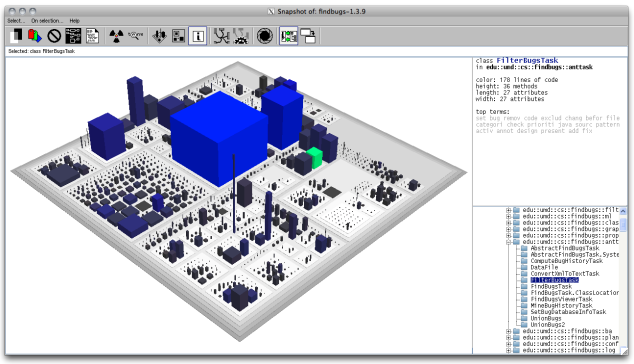

Figure 1: Representation of a software system in CodeCity

Similar to the 2D polymetric view approach [4], we map a set of software metrics on the visual properties of artifacts: The *Number Of Methods* is mapped on the height of the buildings, the *Number Of Attributes* on the base size, and the *number of Lines Of Code* on the color of the buildings, from dark gray (low) to intense blue (high).

We also extended our approach to address the visualization of design problems [18]: Inspired by disease maps we assign vivid colors to design problems and color the affected artifacts according to the design problems that characterize them. The resulting visualization is called disharmony map (see Figure 2) and relies on design problem data computed using Marinescu's detection strategies [7].

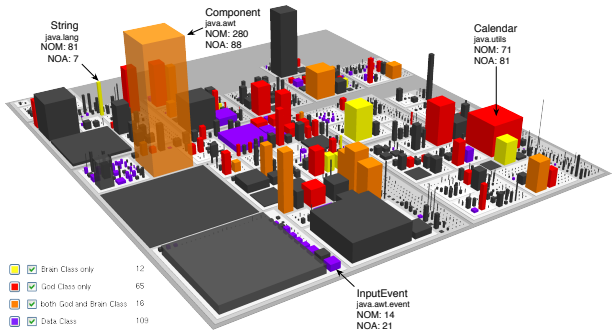

Figure 2: Example of disharmony map (a part of JDK 1.5)

CodeCity also supports software evolution analysis activities [17], which for reasons we explain later we did not evaluate.

Tool support. Our approach is implemented in CodeCity [16], a freely available[1] standalone tool programmed in Smalltalk and depicted in Figure 1. Apart from the main visualization panel, there is an information panel providing contextual data on the selected city artifact (e.g., name, metric values, etc.). CodeCity supports a broad range of facilities to interact, such as inspection or querying, and navigate the visualization. Although not integrated in an IDE, CodeCity features unidirectional linking between the visualization and the source code, i.e., the source code of the software element represented by a city artifact can be accessed in an editor from the visualization. CodeCity is customizable: the user can compose a new view by choosing from a set of metrics and artifact representations.

[1] `http://codecity.inf.usi.ch`

3. RELATED WORK

There is a rich body of research on empirical evaluation of information visualization by means of controlled experiments. To identify both good practices and commonly occurring mistakes, we conducted an extensive study of the literature. The study [19] extends far beyond the scope of this paper; we limit the discussion to controlled experiments for the evaluation of software visualization.

Storey and Müller performed a controlled experiment [14] to compare the support of SHriMP and Rigi in solving a number of program comprehension tasks, both to each other and to a baseline (i.e., SNIFF+).

Marcus et al. performed a study to test the support provided by the sv3D visualization tool in answering a set of program comprehension questions [6]. The authors compared the performances obtained by using sv3D to the ones obtained by exploring the source code in an IDE and a text file containing metrics values.

Lange et al. performed a controlled experiment in which they evaluated the usefulness of their enriched UML views, by comparing them with traditional UML diagrams [3]. The baseline of the experiment was composed of a UML tool and a metric analysis tool.

Quante performed a controlled experiment for the evaluation of dynamic object process graphs in supporting program understanding [10], using not one, but two subject systems. A statistically significant improvement of the experimental group could only be detected in the case of one of the systems, which shows that relying on solely one system is unsound.

Cornelissen et al. [1] performed a controlled experiment for the evaluation of EXTRAVIS, an execution trace visualization tool. 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 a number of program comprehension tasks.

From the perspective of the experiment's design, there are two major differences between our experiment and the related work, summarized in Table 1.

Experiment	Subjects		Object system(s)	
	Academia	Industry	Classes	kLOC
Storey et al. [14]	30	0	17	2
Marcus et al. [6]	24	0	27	42
Lange et al. [3]	100	0	38 39	? ?
Quante [10]	25	0	475 1,725	43 160
Cornelissen et al. [1]	20	1	310	57
Wettel et al. [19]	21	20	1,320 4,656	93 454

Table 1: Comparing to the related work

The first is that in our experiment, we have an equal split between subjects from academia (i.e., 21) and industry (i.e., 20), while the rest of the experiments have only subjects from academia, with the exception of Cornelissen et al., who had one single industry practitioner. While students are convenient subjects, a sample made up entirely of students might not adequately represent the intended user population [9].

The second issue regards the fact that the systems used by most of the experiments are rather small and, thus, not representative for a realistic work setting. In our experiment the largest of the two systems has almost half a million lines of code. Moreover, learning from Quante's experiment we do not rely on one system only, but on two.

4. EXPERIMENTAL DESIGN WISH LIST

We conducted a very exhaustive survey of research works dealing with experimental validation of software engineering, information visualization, and software visualization approaches. The survey, which included more than 50 articles and various books, is detailed in a technical report [19]. From the survey we distilled the following experimental design wish list, which we kept in mind as we designed and conducted the experiment:

1. **Choose a fair baseline for comparison.** If one wants to provide evidence for the value of an approach the question of what one compares it to has to be answered unambiguously.
2. **Involve participants from industry**. Any approach devised to support practitioners in their work is best evaluated using a subject sample with a fair share of software practitioners.
3. **Take into account the range of experience level of the participants.** Experience can greatly influence the outcome of the experiment.
4. **Provide a tutorial of the experimental tool to the participants**. The experimental group would require intensive training to even come close to the skills of the control group acquired in years of operation. Therefore, a minimal exercising of the experimental group in using the features required to solve the tasks is paramount.
5. **Find a set of relevant tasks**. The tasks should be close in scope and complexity to real tasks performed by practitioners.
6. **Include tasks which may not advantage the tool being evaluated.** This allows the experimenters to actually learn something during the experiment, including shortcomings of their approach.
7. **Limit the time allowed for solving each task**. Allowing unbounded time for a task to avoid time pressure may lead participants to spend the entire time allotted for the experiment on solving a single task, as Quante's experiment [10] showed.
8. **Choose real-world systems.** Many experiments in the literature use small systems, making it hard to generalize the results for real-world systems.
9. **Include more than one subject system in the experimental design**. Quante [10] showed that performing the same experiment on two different systems can lead to significantly different results.
10. **Provide the same data to all participants**. The observed effect of the experiment is more likely attributable to the independent variables if this guideline is followed.
11. **Report results on individual tasks**. This allows for a more precise and in-depth analysis of the strengths and weaknesses of an approach.
12. **Provide all the details needed to make the experiment replicable**. Di Penta et al. discuss the benefits of replicability and present a set of guidelines that enable the replication of the experiment [9].

5. EXPERIMENTAL DESIGN

The purpose of our experiment is to quantitatively evaluate the effectiveness and efficiency of our approach when compared to state-of-the-practice exploration approaches.

5.1 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's 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's size?

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

$Q4$: 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 hypotheses and alternative hypotheses corresponding to the first two research questions are synthesized in Table 2.

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 2: Null and alternative hypotheses

For the third question, we perform a separate analysis of correctness and completion time for each of the tasks. For the last two questions we analyze the data within blocks, which we however do not discuss in this paper due to space reasons. We refer the interested reader to our detailed technical report [19].

5.2 Dependent and Independent Variables

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 exploration tools. Additionally, we are interested in how well CodeCity performs compared to the baseline when analyzing systems of different magnitudes. Consequently, our experiment has two independent variables: the *tool* used to solve the tasks and the *object system size*. The tool variable has two levels, i.e., CodeCity and a baseline, chosen based on the criteria described in Section 5.2.1. The object system size has two levels, i.e., medium and large, because visualization starts to become useful only when the analyzed system has a reasonable size. The object systems chosen to represent these two treatments are presented in Section 5.2.2.

Similarly to other empirical evaluations of software visualization approaches [3, 10, 1], the dependent variables of our experiment are *correctness* of the task solution (i.e., a measure of effectiveness) and *completion time* (i.e., measure of efficiency). 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.

5.2.1 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 settle for a reasonable compromise. We looked for two characteristics in an appropriate baseline: 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. To allow a fair comparison, without having to limit the task range, we opted to build a baseline from several tools. The baseline needed to provide exploration and querying functionality, support for 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 and visualization 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, a popular IDE in both academia and industry, which represents the current state-of-the-practice. 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 metrics tool, but could not find one that fit our requirements (including support for user defined metrics). 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 tables containing the metrics and 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. Providing the users with several projects in Eclipse representing different versions of the same system, with no relation among them (or even worse, with just a versioning repository), would have been unfair.

5.2.2 Object Systems

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[2], a tool using static analysis to find bugs in Java code, developed as an academic project at the University of Maryland, while the larger system is Azureus[3], a popular P2P file sharing client and one of the most active open-source projects hosted at SourceForge. In Table 3, we present the main characteristics of the two systems related to the tasks of the experiment.

	System size	
	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 3: The two object systems

[2] http://findbugs.sourceforge.net
[3] http://azureus.sourceforge.net

5.3 Controlled Variables

We identified two factors that could have an influence on the subjects' performance, i.e., their background and experience level. The *background* represents the working context of a subject, which we divided into *industry* and *academy*. The second factor is *experience level*, which represents the domain expertise gained by each of the participants, divided into *beginner* and *advanced*. For academia, subjects below a PhD were considered beginners, while researchers (i.e., PhD students, post-docs and professors) were considered advanced. For industry, we considered that participants with up to three years of experience were beginners, and the rest advanced.

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., academy-beginner, academy-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.

5.4 Treatments

By combining the two levels of each of the two independent variables we obtain four treatments, illustrated in Table 4.

Tool	**Azureus**	**Findbugs**	**Treatment Description**
CodeCity	**T1**	**T2**	CodeCity installation with a loaded model of the system to be analyzed, and the source code of the object system, accessible from the visualizations.
Ecl+Exl	**T3**	**T4**	Eclipse installation with default development tools, an Eclipse workspace containing a project with the entire source code of the object system, 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.

Table 4: Treatments

We provided the treatments as virtual images for VirtualBox[4], which was the only piece of software required to be installed by the participants. Each virtual image contained only the necessary pieces of software (i.e., either CodeCity or Eclipse+Excel), installed on a Windows XP SP2 operating system.

5.5 Tasks

Our approach provides aid in comprehension tasks supporting adaptive and perfective maintenance. We tried to use the previously-defined maintenance task definition frameworks by Pacione et al. [8] and by Sillito et al. [12] to design the tasks of our evaluation. However, both frameworks proved ill-suited. Since CodeCity relies exclusively on static information extracted from the source code, it was not realistic to map our tasks over Pacione's model, which is biased towards dynamic information visualization. Sillito's well-known set of questions asked by developers, 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 systems at higher levels of abstraction not covered by Sillito's framework.

We decided to use these works as inspirations and defined our own set of tasks, detailed in Table 5, dealing with various maintenance concerns and split into program comprehension (A) and design quality assessment (B).

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

Id	Task	Concern
A1	**Description.** Locate all the unit test classes of the system and identify the convention (or lack therof) used by the developers to organize the tests. **Rationale.** Test classes are typically defined in packages according to a project-specific convention. Before integrating their work in 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 in the system.	Structural understanding
A2.1	**Description.** Look for the term *T1* in the names of classes and their attributes and methods, and describe the spread of these classes in the system. **Rationale.** Assessing how domain knowledge is encapsulated in source code is important in several scenarios. To understand a system they are not familiar with, *developers* often start by locating familiar domain concepts in the source code. *Maintainers* use concept location on terms extracted from change requests to identify where changes need to be performed in the system. Software *architects* want to maintain a consistent mapping between the static structure and the domain knowledge. Each of these tasks starts with locating a term or set of terms in the system and assess its dispersion.	Concept location
A2.2	**Description.** Look for the term *T2* in the names of classes and their attributes and methods, and describe the spread of these classes in the system. **Rationale.** Same as for task A2.1. However, the term *T2* was chosen such that it had a different type of spread than *T1*.	Concept location
A3	**Description.** Evaluate the change impact of class *C* 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). **Rationale.** Impact analysis allows one to estimate how a change to a part of the system impacts 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	**Description.** Find the three classes with the highest number of methods (NOM) in the system. **Rationale.** Classes in object-oriented systems ideally encapsulate one single responsibility. Since methods are 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 either *maintenance* activities or *quality assurance*.	Metric-based analysis
A4.2	**Description.** Find the three classes with the highest average number of lines of code per method in the system. **Rationale.** It is difficult to prioritize candidates for refactoring from a list of large classes. In the absence of other criteria, the number and complexity of methods can be used as a measure of the amount of functionality for solving this problem related to *maintenance* and *quality assurance*.	Metric-based analysis
B1.1	**Description.** Identify the package with the highest percentage of god classes in the system. **Rationale.** God classes are classes that tend to incorporate an overly large amount of intelligence. Their size and complexity often make them a *maintainer*'s nightmare. Keeping these potentially problematic classes under control is important. By maintaining the ratio of god classes in packages to a minimum, the *quality assurance engineer* keeps this problem manageable. 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.	Focused design assessment
B1.2	**Description.** Identify the god class containing the largest number of methods in the system. **Rationale.** It is difficult to prioritize candidates for refactoring from a list of god classes. In the absence of other criteria (e.g., the stability of a god class over its 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*.	Focused design assessment
B2.1	**Description.** Identify the dominant class-level design problem (the design problem that affects the largest number of classes) in the system. **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*.	Holistic design assessment
B2.2	**Description.** Write an overview of the class-level design problems in the system. Describe your most interesting or unexpected observations. **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.	Holistic design assessment

Table 5: Tasks

6. OPERATION

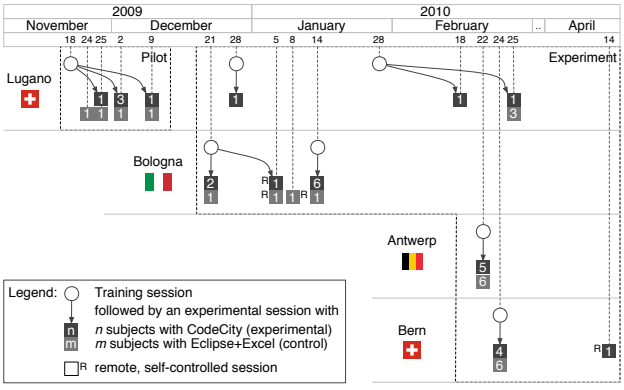

Figure 3: The timeline of the experiment

The operation covered the period Nov 2009–Apr 2010 and was divided in two phases: the pilot and the experiment. Figure 3 shows the timeline of the operation. The operation is composed of a series of experimental runs; each run consists of a one hour training session, followed by one or more experimental sessions of up to two hours. A training session consists of a talk in which the experimenter presents the approach, concluded with a CodeCity demonstration. During the experimental session(s), the subjects solve the tasks with the assigned tool on the assigned object system, under the experimenter's observation. The numbers in Figure 3 reflect only the participants whose data points were taken into account during the analysis, and not those excluded from it, which we discuss later.

6.1 The Pilot Study

We conducted a pilot study with Master students of the University of Lugano enrolled in a course on Software Design; improving the questionnaire and solving problems as they emerged required several iterations. We organized a joint training session, followed by three experimental sessions, each one week apart. Before the first session we conducted the experiment with a researcher from our group, with extensive experience with Eclipse, to make sure the tasks for the control group were doable in the allotted time. We did not include any of these data points in the analysis.

6.2 Experimental Runs

At this point, we were confident enough to start our experiment. We performed the following experimental runs:
Bologna I. 8 professionals with 4–10 years of experience.
Bologna II. 9 professionals with 7–20 years of experience.
Lugano I. 1 researcher/development leader of a small company.
Lugano II & III. 5 practioners with 10+ years of experience.
Antwerp. 3 Ph.D. and 8 MSc students.
Bern. 2 consultants, 1 professor, 7 Ph.D. and 1 MSc student.

7. DATA COLLECTION

We collected data continuously—before, during, and after the experiment.

7.1 Personal Information

Before the experiment, we collected both personal information (e.g., gender, age, nationality) and professional data (e.g., job position, experience with object-oriented programming, Java, Eclipse, and reverse engineering) by means of an online questionnaire, to be found in [19].

7.2 Timing Data

To time participants accurately, we developed our own timing web application in Smalltalk. During the experimental sessions, the timing application would run on the experimenter's computer and project the current time.

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, for each participant, the name, current task, and the remaining time allotted for the task.

The subjects were asked to announce to the experimenter every time they logged 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 (10 minutes for each task), the application would display the message "overtime" beside the name, and the experimenter would ask the subject to immediately pass to the next task, before resetting the timer.

7.3 Correctness Data

To convert the collected task solutions into quantitative information, we needed an oracle, which would provide both the set of correct answers and the grading scheme for each task. Having two object systems and two data sources (source code for the control group and a model of the system for the experimental group), led to four oracle sets. To build them, two of the authors and a third experimenter solved the tasks with each treatment and designed the grading scheme for each task. In addition, for the two experimental treatments, we computed the results using queries on the model of the object system, to make sure that we were not missing any detail because of particularities of the visual presentation (e.g., occlusion, too small buildings, etc.). Eventually, we discussed the divergences and merged our solutions.

Finally, we needed to grade the solution of the subjects. We employed blind marking to rule out bias; when grading a solution the experimenter does not know whether the subject that provided the solution has used an experimental or a control treatment. For this, one of the authors created four code names for the groups and created a mapping between groups and code names, known only to 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 author who encoded the data performed his grading unblinded. Eventually, the authors discussed the differences and converged towards the final grading. The minimum grade was 0, while the maximum was 8.

7.4 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 which could potentially help us improve the experiment, and optionally, to share with us any interesting insights they encountered during the analysis.

8. DATA ANALYSIS

8.1 Preliminary Data Analysis

We observed an exceptional condition related to task *A4.2*, which had the highest discrepancy in time and correctness between control and experimental groups. The data points showed that the experimental group was not able to solve this task, while the control group was quite successful at solving it. The experimental group had an average of 0.06, with 19 null scores (out of 22), and most subjects used up the entire allotted time (i.e., ceiling effect). The control group had an average of 0.86 with 15 perfect scores (out of 19) and most subjects finished in roughly half the allotted time. The subjects perceived its difficulty accordingly: In the debriefing questionnaire, experimental subjects graded task *A4.2* as "impossible", while the control group described it as "simple", as shown in Figure 4.

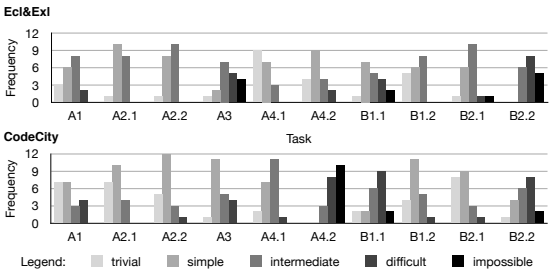

Figure 4: Histograms of perceived difficulty per task

The reason is that we underestimated the knowledge of CodeCity required to perform this task. Solving this task with our tool implies a deep knowledge of CodeCity's customization features, and of the underlying Smalltalk programming language: the only subject who managed to solve the task in the experimental group is a Smalltalk expert. These were unreasonable requirements to expect from the experimental subjects, who were not trained to use such features. To eliminate this unusually large discrepancy between the two groups, we excluded the task from the analysis.

8.2 Outlier Analysis

Before performing our statistical test, we followed the suggestion of Wohlin et al. [20] regarding the removal of outliers caused by exceptional conditions, to allow us to draw valid conclusions from the data. During the Bologna I experimental run, one of the participants assigned with an experimental treatment experienced serious performance slowdowns due to the low performance of his computer. 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. Another participant got assigned to an Ecl+Exl treatment, although 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 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 the control groups), 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. Eventually they were borrowed our two replacement machines, but due to the meeting room's tight schedule, we were not able to wait for them to finish the experiment. We decided to exclude these two data points from our analysis.

8.3 Subject Analysis

After the pilot study involving nine participants, we conducted the experiment with a total of 45 participants in several runs. After removing four data points during the outlier analysis, we were left with 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 treatments, we have 8–12 data points and a fair subject distribution within the remaining three blocks, as shown in Table 6.

			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 6: Subject distribution

Moreover, the random assignments of treatment within blocks led to a balanced distribution of the subjects' expertise among treatments, as we see in Figure 5.

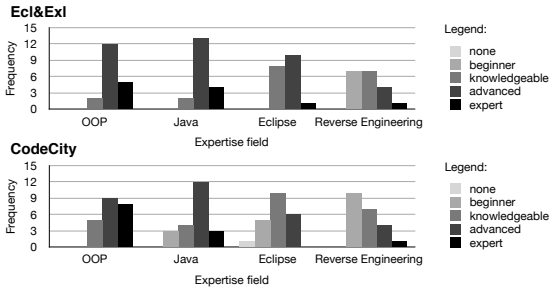

Figure 5: Subjects' expertise

While some of the subjects assigned to CodeCity have little or no experience with Eclipse, every subject assigned to Ecl+Exl is at least *knowledgeable* in using this IDE.

9. RESULTS

Based on the design of our experiment, i.e., a between-subjects, unbalanced (i.e., implying unequal sample sizes) design with two independent variables (tool and system size), the suitable parametric test for hypothesis testing is a two-way Analysis Of Variance (ANOVA). We performed the test for *correctness* and *completion time* using the SPSS statistical package. Before the analysis, we ensured that our data met the test's assumptions:

1. *Independence of observations*. This assumption is implicitly met through the choice of a between-subjects design.
2. *Homogeneity of variances of the dependent variables*. We tested our data for homogeneity of both *correctness* and *completion time*, using Levene's test [5] and in both cases the assumption was met.
3. *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 [11], and also this assumption was met in all the cases.

We chose a significance level of .05 ($\alpha = .05$), which corresponds to a 95% confidence interval. The statistics related to correctness and completion time are presented in Table 7.

9.1 Analysis Results on Correctness

Interaction effect between tool and system size on correctness. It is important that there is no interaction between the two factors, which could have affected the correctness. The interaction effect of tool and system size on correctness was not significant, $F(1, 37) = .034$, $p = .862$. The data shows no evidence that the variation in correctness between CodeCity and Ecl+Exl 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 obtained by CodeCity users was significantly higher than the one for Ecl+Exl users, regardless of the size of the object system.

Overall, there was an increase in correctness of 24.26% for CodeCity users ($M = 5.968$, $SD = 1.294$) over Ecl+Exl 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+Exl 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+Exl users ($M = 3.896$, $SD = 1.085$). The data shows that the increase in correctness for CodeCity over Ecl+Exl was higher for the larger system.

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.

The main effect of both tool and object system size on correctness and the lack of the effect of interaction between tool and object system size on correctness are illustrated in Figure 6, as well as the correctness box plots for the four treatments.

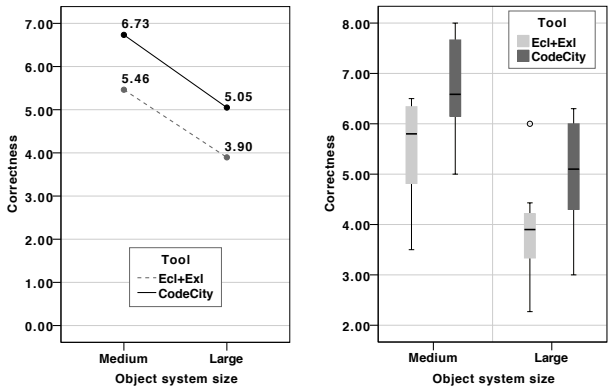

Figure 6: Means and box plots for correctness

9.2 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, which could have affected the completion time. The interaction effect of tool and system size on completion time was not significant, $F(1, 37) = .057$, $p = .813$. The data shows no evidence that the variation in completion time between CodeCity and Ecl+Exl depends on the size of the system, which strengthens any observed effect of the tool factor on the completion time.

Dep. var.	Correctness (points)						Completion Time (minutes)					
System size	Medium		Large		Overall		Medium		Large		Overall	
Tool	Ecl+Exl	CodeCity	Ecl+Exl	CodeCity	Ecl+Exl	CodeCity	Ecl+Exl	CodeCity	Ecl+Exl	CodeCity	Ecl+Exl	CodeCity
mean	5.462	6.733	3.896	5.050	4.803	5.968	38.809	33.178	44.128	39.644	41.048	36.117
difference		**+23.27%**		**+29.62%**		**+24.26%**		**-14.51%**		**-10.16%**		**-12.01%**
min	3.50	5.00	2.27	3.00	2.27	3.00	31.92	24.67	22.83	27.08	22.83	24.67
max	6.50	8.00	6.00	6.30	6.50	8.00	53.08	39.50	55.92	48.55	55.92	48.55
median	5.800	6.585	3.900	5.100	4.430	6.065	38.000	35.575	48.260	40.610	40.080	36.125
stdev	1.147	.959	1.085	1.031	1.349	1.294	6.789	5.545	11.483	6.963	9.174	6.910

Table 7: Descriptive statistics related to correctness and 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 was significantly lower for CodeCity users than for Ecl+Exl users.

Overall, there was a decrease in completion time of 12.01% for CodeCity users ($M = 36.117$, $SD = 6.910$) over Ecl+Exl 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+Exl 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+Exl users ($M = 44.128$, $SD = 11.483$). The data shows that the time decrease for CodeCity users over Ecl+Exl users is slightly lower in the case of the large system compared to that obtained for the medium system.

The effect of system size on completion time. While not the goal of the experiment, a significant effect of system size on the completion time was observed, $F(1, 37) = 5.962$, $p = .020$, indicating that completion time was significantly lower for users analyzing the medium system than for users analyzing the large system.

The main effect of both tool and object system size on completion time and the lack of the effect of interaction between tool and object system size on completion time, are illustrated in Figure 7, as well as the completion time box plots for the four treatments.

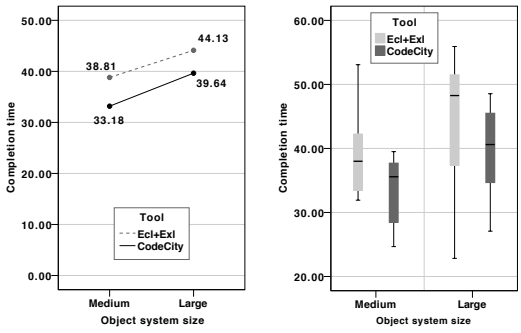

Figure 7: Means and box plots for completion time

9.3 Result Summary

Correctness. The 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+Exl. This result is statistically significant.

Completion time. We can also reject the second null hypothesis $H2_0$ in favor of the alternative hypothesis $H2$, which states that the tool impacts the time required to complete program comprehension tasks. Overall, CodeCity enabled a completion time reduction of 12.01% over Ecl+Exl. This result is statistically significant.

9.4 Task Analysis

A secondary goal of our experiment was to identify the types of tasks for which CodeCity provides an advantage over the baseline. To this end, for each task described in Section 5.5 we compared the performances—in terms of correctness and time—of the two *tools* and reasoned about the potential causes behind the differences. See Figure 8 for a graphical overview supporting our task analysis[5].

A1 - Identifying the convention used to organize unit tests relatively to the tested classes. While Eclipse performed consistently, CodeCity outperformed it on the medium system and underperformed it on the large system. The difference in performance is partially owed to fact that, in spite of the existence of a number of classes named `*Test`, there are no unit tests in the large system. Only a small number of CodeCity users examined the inheritance relations, while the majority relied only on name matching. The time is slightly better for the CodeCity subjects, who benefited from the visual overview of the system, while Eclipse required scrolling up and down through the package structure.

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 term search is completed, the spread can be visually assessed. In Eclipse, the term search produces a list of class names. For this task the list showed classes in many packages belonging to different hierarchies and, therefore, a dispersed spread may represent a safe guess.

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 significantly 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 control subjects into believing that the spread of the term was dispersed, while the CodeCity users took advantage of the "big picture" and better identified the localized spread of this term.

A3 - Estimating impact. CodeCity significantly outperformed Eclipse in correctness by 40–50%, and was slightly faster than Eclipse. Finding the caller classes of a given class in Eclipse is not straightforward. Moreover, the resulting 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 system and slightly better for the large system. In terms of completion time, Excel was slightly faster than CodeCity. While CodeCity is faster at building an approximate overview of systems, a spreadsheet is faster at finding precise answers in large data sets.

B1.1 - Identifying the package with the highest percentage of god classes. In both correctness and completion time, CodeCity slightly outperformed Excel. The low scores in correctness of both tools shows that none of them is good enough to solve this problem alone. Instead, CodeCity's visual presentation and Excel 's precision could complement each other well.

[5]Due to space concerns, we do not discuss task B2.2

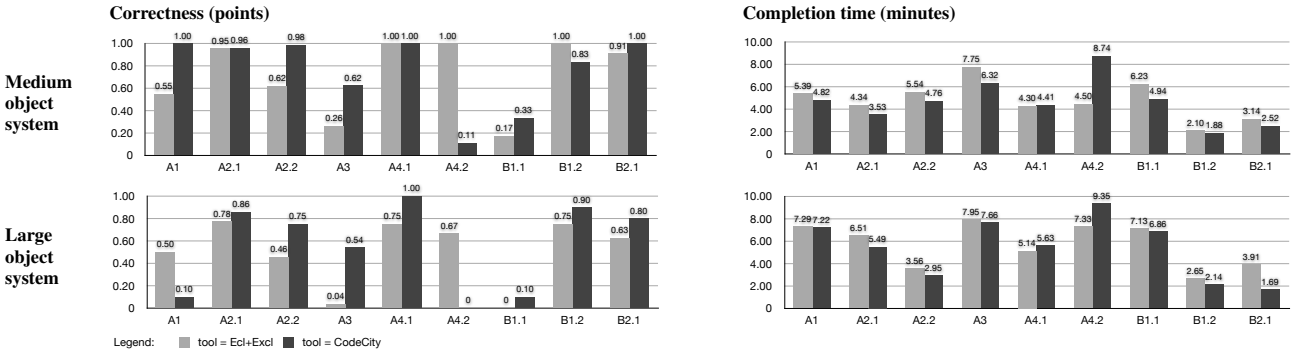

Figure 8: Average correctness and completion time per task

B1.2 - Identifying the god class with the highest number of methods. Both tools obtained good correctness scores, i.e., over 75%. Excel was slightly better in the case of the medium size system, while CodeCity outperformed Excel in the case of the large system. While CodeCity's performance was consistent across systems with different sizes, Excel's support was 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 outperformed Excel regardless of system size. The aggregated data found in CodeCity's disharmony map was less error-prone than counting rows in Excel. In terms of completion time, CodeCity significantly outperformed Excel and the difference was probably caused by the lack of overview in Excel.

Summary. As expected, at focused tasks (e.g., *A4.1*, *B1.1*) CodeCity did not perform better than the baseline, because Excel is very efficient in finding precise answers (e.g., the largest, the top N). However, it is surprising that, in most of these tasks, CodeCity managed to be on par with Excel. At tasks that benefit from an overview, (e.g., *A2.1*, *A3*, or *B1.2*), CodeCity constantly outperformed the baseline, in particular in terms of correctness, mainly because the overview enabled the experimental group to produce a faster and more confident solution compared to the control group.

10. THREATS TO VALIDITY

10.1 Internal Validity

The internal validity refers to uncontrolled factors that may influence the effect of the treatments on the dependent variables.

Subjects. To reduce the threat that the subjects may not have been competent enough, we ensured they had expertise in relevant fields, using an online questionnaire. Second, to mitigate the threat that the subjects' expertise may not have been fairly distributed across the control and experimental groups, we used randomization and blocking to assign treatments to subjects.

Tasks. The choice of tasks may have been biased to the advantage of CodeCity. We alleviate this threat by presenting the tasks in context with a rationale (described in [19] and left out here due to space constraints) and the targeted user roles. Moreover, we included tasks which disadvantage CodeCity (e.g., tasks focused on precision, rather than on locality). Second, the tasks may have been too difficult. Third, the allotted time per task may have been insufficient. To alleviate these two threats we performed a pilot study and collected feedback about the perceived task difficulty and time pressure. Moreover, we excluded the only task whose difficulty was discordantly perceived by the two groups. 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. We compared CodeCity with a baseline composed of two different tools and this might have affected the performance of the control group. We attenuate this threat by designing the task set 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 part of the experiment, while all the tasks that were to be solved with Excel were grouped in the second part 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+Exl subjects on task *A4.1*, in both correctness and time, provide no indication of such a negative effect.

Data differences. CodeCity relies on models of the systems, while Eclipse works with the source code. The data differences might have had an effect on the results of the two groups. To alleviate this threat, we accurately produced the answer model based on the available data source, i.e., source code or model, and made sure that the slight differences did not lead to incompatible answers.

Session differences. There were several runs 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 the pilot phase and obtained a stable and reliable experimental setup (e.g., instrumentation, questionnaires, experimental kit). 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. We only trained the subjects with the experimental treatment and this may have influenced the result of the experiment. We afforded to do so because we chose a 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 control subjects remarked the fact that we should have included Excel among the assessed competencies, they scored well on the tasks solved with Excel, due to the rather simple operations (i.e., sorting, arithmetic operations between columns) required to solve them. 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.

10.2 External Validity

This refers to the generalizability of the experiment's results.

Subjects. To mitigate the threat of the representativeness of the subjects, we categorized our subjects in four categories along two axes (i.e., background and experience level) and strived to cover all categories. We obtained a balanced mix of academics (beginners and advanced) and industry practitioners (only advanced).

Tasks. Our choice of 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, are either too low-level (e.g., the questions asked by practitioners during a programming change task by Sillito et al. [12]), or biased towards dynamic analysis tools (e.g., Pacione's framework [8]). To alleviate this threat, we complemented our tasks with usage scenarios and targeted users.

Object systems. The representativeness of our object systems is another threat. We chose to perform the experiment with two different object systems, in spite of the added complexity in organizing the experiment and analyzing the data introduced by a second independent variable. The two object systems we chose are well-known open-source systems of different, realistic sizes and of orthogonal application domains. It is not known how appropriate they 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. One instance of this threat is that the task solutions may not have been graded correctly. To mitigate this threat, the three authors 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. Even if we tried to mitigate this threat extensively, we cannot exclude all the possible influences of this factor on the results of the experiment.

11. CONCLUSION

We presented a controlled experiment aimed at evaluating our software visualization approach based on a city metaphor. We designed our experiment from a list of desiderata built during an extensive study of the current body of research. We recruited large samples of our target population, with subjects from both academia and industry with a broad range of experience.

The main result of the experiment is that our approach leads to an improvement, in both correctness (+24%) and completion time (-12%), over the state-of-the-practice exploration tools. This result is statistically significant. We believe this is due to both the visualization as such, as well as the metaphor used by CodeCity, but we can not measure the exact contribution of each factor.

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 Eclipse+Excel. Surprisingly, for most of these tasks, our approach managed to be on a par with Excel. As for the tasks that benefit from an overview of the system, CodeCity constantly outperformed the baseline.

Finally, we designed our experiment with repeatability in mind. In [19], we provided the complete raw and processed data of this experiment (i.e., the pre-experiment, in-experiment and debriefing questionnaires, solution oracles and grading systems, correction scores and measured completion time) to allow reviewers to better evaluate the experiment's design and results, and fellow researchers to repeat the experiment, or reuse its design as a base for their own.

Acknowledgments. We acknowledge R Marinescu, M Lungu, A Bacchelli, L Hattori, O Nierstrasz, S Demeyer, F Perin, and Q Soetens for helping us with the experiment. We thank our participants: the developers in Bologna and Lugano, the SCG in Bern, and the students in Lugano and Antwerp. We thank the European Smalltalk User Group (`http://esug.org`) for financial support.

12. REFERENCES

[1] B. Cornelissen, A. Zaidman, B. V. Rompaey, and A. van Deursen. Trace visualization for program comprehension: A controlled experiment. In *Proceedings of ICPC 2009*, pages 100–109. IEEE CS Press, 2009.

[2] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[3] C. F. J. Lange and M. R. V. Chaudron. Interactive views to improve the comprehension of UML models - an experimental validation. In *ICPC 2007*, pages 221–230. IEEE, 2007.

[4] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[5] H. Levene. Robust tests for equality of variances. In I. Olkin, editor, *Contributions to Probability and Statistics*, pages 278–292. Stanford U Press, 1960.

[6] A. Marcus, D. Comorski, and A. Sergeyev. Supporting the evolution of a software visualization tool through usability studies. In *ICPC 2005*, pages 307–316. IEEE, 2005.

[7] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM 2004*, pages 350–359. IEEE CS Press, 2004.

[8] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *WCRE 2004*, pages 70–79. IEEE CS Press, 2004.

[9] M. D. Penta, R. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *ICPC 2007*, pages 281–285. IEEE CS Press, 2007.

[10] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *ICPC 2008*, pages 73–82. IEEE CS Press, 2008.

[11] S. Shapiro and M. Wilk. An analysis of variance test for normality. *Biometrika*, 52(3–4):591–611, 1965.

[12] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *FSE-14*, pages 23–34. ACM Press, 2006.

[13] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization*. The MIT Press, 1998.

[14] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In *Proceedings of WCRE 1997*, pages 12–21. IEEE CS Press, 1997.

[15] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007*, pages 231–240. IEEE CS Press, 2007.

[16] R. Wettel and M. Lanza. CodeCity: 3D visualization of large-scale software. In *Proceedings of ICSE 2008, Tool Demo Track*, pages 921–922. ACM Press, 2008.

[17] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of WCRE 2008*, pages 219–228. IEEE CS Press, 2008.

[18] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of Softvis 2008*, pages 155–164. ACM Press, 2008.

[19] R. Wettel, M. Lanza, and R. Robbes. Empirical validation of CodeCity: A controlled experiment. Tech Report 2010/05, University of Lugano, June 2010.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.