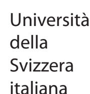
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by RERO DOC Digital Library



USI Technical Report Series in Informatics

Empirical Validation of CodeCity: A Controlled Experiment

Richard Wettel[1], Michele Lanza[1], Romain Robbes[2]

[1] REVEAL @ Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] PLEIAD Lab, DCC, University of Chile

Abstract

We describe an empirical evaluation of a visualization approach based on a 3D city metaphor, implemented in a tool called CodeCity. We designed the controlled experiment based on a set of lessons extracted from the current body of research and perfected it during a preliminary pilot phase. We then conducted the experiment in four locations across three countries over a period of four months, involving participants from both academia and industry. We detail the decisions behind our design as well as the lessons we learned from this experience. Finally, we present the results of the experiment and the complete set of data required to ensure repeatability of the experiment.

Report Info

Published
June 2010

Number
USI-INF-TR-2010/05

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

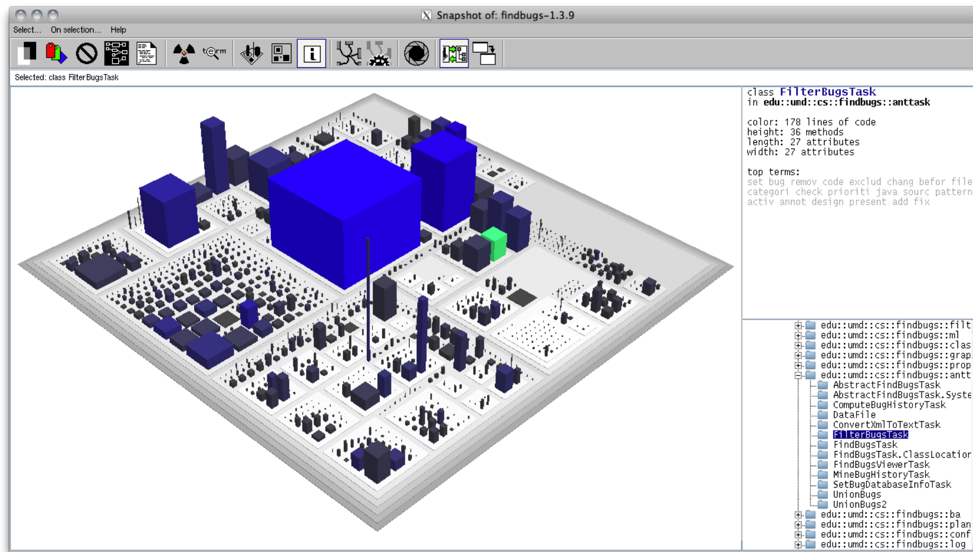
Software visualization is an aid in reverse-engineering software systems and in the resource allocation for maintenance tasks. Therefore, it is important to investigate whether the use of software visualization—which produces oftentimes spectacular interactive tools, capable of generating beautiful images—can make a significant difference in practice, one that can justify the costs of adopting such a tool in an industrial context.

Our approach for the analysis of —primarily— object-oriented software systems, which uses 3D visualizations based on a city metaphor [43], is briefly described next.

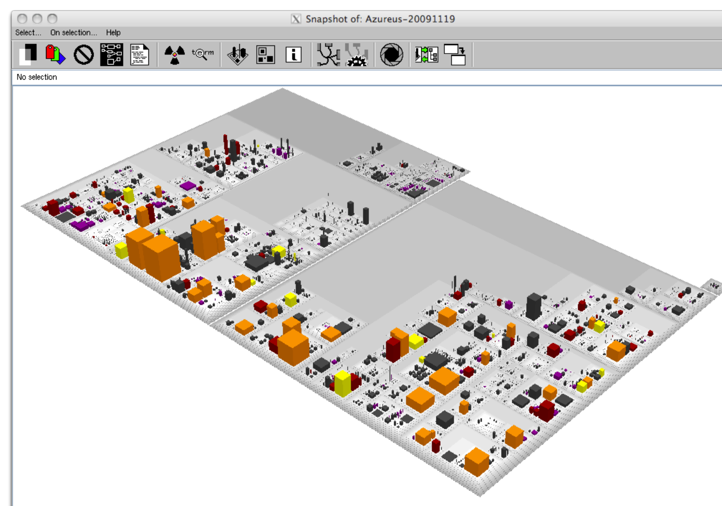
1.1 Software Systems as Cities

Our approach, based on a 3D city metaphor, provides a complex environment for the exploration of software systems, with a clear notion of locality that counteracts disorientation, i.e., an open challenge in 3D visualization. According to this metaphor, software systems are visualized as cities, whose buildings represent the classes of the system and whose districts depict the system's packages. The visual properties of the city artifacts depict software metrics.

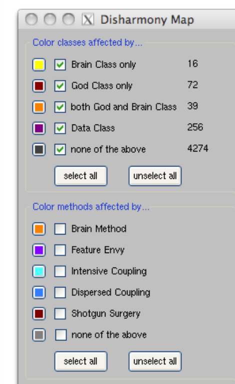
Initially, we designed the metaphor to support users in solving high-level program comprehension tasks based on one system version [42]. Later, we extended it to enable analyses of system evolution [46] and design problem assessments [47]. We validated each of these applications of our approach, by applying it on several case studies consisting of open-source software systems. To be able to apply our approach on real software systems, we implemented all the promising ideas emerging from our research in a tool called CodeCity [45, 44].



(a) Structural overview of Findbugs



(b) Disharmony map of Azureus



(c) Legend

Figure 1: CodeCity visualizations

1.2 CodeCity

From a user interface perspective, CodeCity is a sovereign (i.e., it requires the entire screen [3]) and interactive application. To allow experimenting with visualizations, CodeCity provides a high customizability, by means of view configurations (i.e., property mappings, layouts, etc.) and later through scripting support [41]. The tool, which is freely available[1], has been released in March 2008 and has reached its fifth version (i.e., 1.04) before the beginning of the experiment.

Since we can only perform the evaluation of our approach by means of the tool, the outcome of this evaluation depends not only on the soundness of the approach, but also on the efficiency and usability of its implementation. We believe that, after almost two years, CodeCity has achieved the maturity required by an empirical evaluation. We designed and conducted a controlled experiment aimed at discovering whether, under which circumstances, and to whom is our approach useful.

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

1.3 The Experiment in a Nutshell

In this experiment we evaluate two of the three applications of our approach, i.e., structural understanding and design problem assessment.

The CodeCity view configuration used in the experiment encapsulates the following property mapping: The number of methods (NOM) metric of classes is mapped on the buildings' height, the number of attributes (NOA) on their base size, and the number of lines of code (LOC) on the color of the buildings, i.e., gray buildings are the classes with the lowest LOC value, while intensive blue classes are the classes with the highest LOC value. In the case of packages, the nesting level metric is mapped on the districts' color saturation, i.e., root packages are dark gray, deeply nested packages are light gray. The package hierarchy is thus reflected by the city's district structure, as seen in Figure 1(a).

To support design quality analyses, we devised a technique we call *disharmony maps* [47], which are code cities enriched with design problem data (See Figure 1(b)). Inspired by disease maps, we assigned vivid colors to the various design problems and shades of gray to the unaffected entities, which enables the users to focus on the design disharmonies in the presence of a non-distracting context. Disharmony maps enable the users to localize the affected elements and assess the distribution of design problems throughout the system.

The main goal of the experiment is to compare the efficiency and effectiveness of our approach to the ones of the state-of-the-practice. To this end, we compare the performances in terms of correctness and completion time of CodeCity and of a baseline toolset, in providing support in solving the task set.

Structure of this Report. In Section 2 we cover the related work in a study on empirical validations of visualization tools. Based on the lessons learned from the body of research, we built a "wish list" of desirable features for our experimental design, presented in Section 3. Section 4 describes the experimental design we derived from this wish list, including the list of tasks to be carried out by the subjects of the experiment, as well as the actual subjects that took part in our experiment. Section 5 describes how we actually performed the experiment, while Section 6 details our data collection procedure. In Section 7, we present our data analysis aimed at finding and discarding outliers in data. In Section 8 we present the results of the statistical analysis, in terms of time taken to perform the tasks, and correctness of the solutions. Finally, we discuss potential threats to the validity of our experiment in Section 9, before concluding in Section 10.

2 Related Work

There is a rich body of research on empirical evaluation through controlled experiments. We first conducted an extensive study of the literature to identify both good practices and commonly occurring mistakes in controlled experiments. Given the wide span of related work, we limit ourselves to discuss mostly the work that contributed in one way or another to the design of our experiment. The lessons extracted from this study are synthesized in a list of *desiderata* for the design of our experiment, presented in Section 3.

2.1 Guidelines for Information Visualization Evaluation

Since software visualization is rooted in the information visualization field, we first look at the means of evaluation for information visualization research.

Plaisant acknowledges the challenge of information visualization evaluation, but also its major role in increasing the credibility of tools towards an industry adoption [29]. The author divided the current evaluation practices in the following categories:

- controlled experiments that compare design elements
- usability evaluations of tools
- controlled experiments comparing two or more tools
- case studies of tools in realistic settings

Two important matter emphasized in this work is the use of real datasets and the demonstration of realistic tasks. 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.

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 for benchmarks repositories of datasets and tasks.

In their analysis of user evaluation studies in information visualization [7], 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": When evaluating solely aspects that are questionable, one is more likely to find problems in the visualization, while 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 could also show how much they enjoyed themselves solving it.

Zhu proposed a framework for the definition and measurement of effective data visualization [50], according to three principles:

1. *Accuracy*, i.e., the attributes and structure of a visual element should match the ones of the represented data item
2. *Utility*, i.e., an effective visualization should help users achieve the goal of specific tasks
3. *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—finding concrete means to measure these effectiveness metrics—has unfortunately not been addressed by the author of this work with concrete solutions.

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 [28]. In this work, the author focused mostly on the difference between novice users and experts and discussed these difference. 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—an 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 therefore take it into account in the design of our experience, 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. [48]. 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 created 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, to allow the users to perform real tasks.

Stasko et al. [36] 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 [14]. 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) and Windows Explorer as a baseline [15]. There were 15 tasks and the object system was a test hierarchy representing a subset of a taxonomy of items on eBay. The participants in this experiment 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 a series of interesting insights in relation to each of the tools. An interesting outcome of the experiment was that, in the end, the most preferred tool was the non-visual, yet popular, Windows Explorer.

Kosara et al. [16] 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 displays. 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 the fact that in visualization, one cannot publish null results (i.e., results showing that the original hypothesis was not supported by the data), in spite of their intrinsic value.

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. Finally, we had the chance to watch professionals using our tool in their own after-work environment, i.e., during a user group meeting, which was a valuable experience.

O'Donnell et al. [25] presented an evaluation experiment for an area-based tree visualization called PieTree. 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 just using the PieTree or the TreeView alone. The comparison chosen by the authors is rather weak, 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 showed that the combination of the two tools was outperformed by the use of one of the tools in every task—strengthen the belief that more is *not* always better.

Although there are several other controlled experiments in the information visualization field [2, 8, 40], we restrict ourselves to the ones that influenced our experiment's design.

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 [37], 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, obtained by way of electronic mail [17]. 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. The author 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 the existing representation in 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.

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 [35]. However, this valuable framework focuses on the source code level and supports mainly developers. Therefore, it is not very appropriate for the evaluation of our approach, which supports developers, but also architects, designers, project managers, in solving high-level reverse engineering and comprehension tasks.

Pacione et al. [26] 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.

2.5 Guidelines for Software Visualization Evaluation

Kitchenam et al. [12] proposed a set of guidelines for designing, conducting, and evaluating empirical research in the more general context of software engineering. Some of these are just as 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 in software visualization discussed later.

One of the problems in designing and performing evaluations of software visualization approaches is the lack of software visualization benchmarks, a reality acknowledged by Maletic and Marcus, who launched a call for such contribution, to raise the awareness of the scientific community on this problem [20].

Di Penta et al. synthesized a set of guidelines for designing empirical studies in the field of program comprehension [27]. 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. [33] shared a number of lessons learned during the process. One lesson is the risk of involving participants covering a wide range of levels of experience, 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 a few minutes just before the experiment, we planned to perform a session to present our approach before each experimental run.

With respect to the 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, and therefore its usefulness in finding unexpected facts has only been shown during 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) should be used complementarily to deal with today's software systems. Some of the more experienced industry practitioners that participated in our experiment, or only 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 a tool. To this end, the presentation session preceding every experimental session in our design includes a presentation of the approach and a tool demonstration, which provides benefits for both professional interested in new tools, and academics active in software visualization, reverse engineering, or program comprehension.

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 [17]. Consequently, there is a growing request for empirical evaluations in software visualization.

Storey and Müller were among the first ones to tackle the evaluation of 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 [38]. In a second step, Storey et al. performed the actual experiment [39], 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 their 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. [21] 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 use of their tool to the exploration of a text file containing all the metrics data and of source code in an IDE, which is 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 ubiquitous 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. The authors felt 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. [1] 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. [12], 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 cannot 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 academy), 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. Complementary to the overall results, we also consider each task separately for the same reasons. 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 [18]. 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 decided 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 analyze the time on a task-by-task basis, complementary to the overall time.

Quante performed a controlled experiment for the evaluation of his dynamic object process graphs in supporting program understanding [30]. 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 all the 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 [13]. 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 makes 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.

Cornelissen et al. [4, 5] performed a controlled experiment for the evaluation of EXTRAVIS, an execution trace visualization tool. The experiment consisted of solving four "typical tasks", divided in eight sub-tasks, which—as the authors claimed—cover all the activities in Pacione's model [26]. The choice of the model fits the nature of this 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. Obviously, as the authors discussed in the threats to validity, one single subject from industry cannot generate any statistically relevant insights that holds for industry practitioners in general.

However, 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 [12]. 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 repeatable.

3 Wish List Extracted from the Literature

The literature survey we conducted allowed us to establish the following list of guidelines for our experiment, extracted from both the success and the shortfalls 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 [33]. 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. [21] 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 not possible 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. [16].

7. **Choose real object systems that are relevant for the tasks.** Many of the experiments in the literature use very small systems as objects and therefore, the results 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. [29].

8. **Include more than one subject system in the experimental design**. The experiment of Quante [30] 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 alloted 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. [27] and made available the materials and results to facilitate its evaluation and replication:

    - subject selection criteria and justification
    - subject screening materials and results (with private information replaced by unique identifiers)
    - pre-test questions, and results keyed to the unique subject identifiers, as well as explanation 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 [1].

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 [28, 13], we use blocking, which implies dividing the subjects of each treatment into blocks based on their experience and skills.

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.

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 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 1.

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 1: Null and alternative hypotheses

The remaining three questions, although secondary, allow us to search for more precise insights about our approach. 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 perform an analysis of the data within blocks.

4.2 Dependent & 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 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 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, chosen based on the criteria described in Section 4.2.1. 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 object systems chosen to represent these two treatments are presented in Section 4.2.2.

Similarly to other empirical evaluations of software visualization approaches [18, 30, 4], the dependent variables of our experiment are *correctness* of the task solution and *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 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.

4.2.1 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 context in which we applied our approach: software understanding, evolution analysis, and design quality assessment. Instead, we had to settle to 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. The first candidate was a highly configurable text-based reverse engineering tool called MooseBrowser [24], built on top of the Moose reengineering platform[2]. MooseBrowser is data-compatible with CodeCity, for it uses the same underlying meta-model for object-oriented software (i.e., FAMIX [6]) and is able to cover the features of CodeCity we wanted to test. However, in spite of the enthusiastic Moose community, MooseBrowser cannot yet be considered state-of-the-practice in reverse engineering.

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 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[3], a popular IDE 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[4]) that would either include the metrics we needed for the tasks, or provide support for entering user-defined metrics, or allow us to hard-code the data we had in CodeCity. Again, none of the tools we found allowed 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 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. We felt that 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.

4.2.2 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[5], a tool using static analysis to find bugs in Java code, developed as an academic project at the University of Maryland [11], while the larger system is Azureus[6], a popular P2P file sharing clients and one of the most active open-source projects hosted at SourceForge. In Table 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 2: The object systems corresponding to the two levels of system size

[2] http://www.moosetechnology.org
[3] http://www.eclipse.org
[4] http://www.sonarsource.org
[5] http://findbugs.sourceforge.net
[6] http://azureus.sourceforge.net

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 academy. The background information is directly extracted from the personal information provided by the participants at the time of their enrollment. If a participant is active in both an academic and an industrial context, we chose the role that is the most convenient for the experiment.

The second factor is *experience level*, which represents the domain expertise gained by each of the participants. To keep things simple, the experience level also has 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. First, for participants from the 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 that participants with up to three years of experience are 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. The outcome of this procedure is described in Section 4.6, after presenting our subjects.

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. [26], which is biased towards dynamic information visualization. On the other hand, the set of questions asked by developers, synthesized by Sillito et al. [35], 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 for each combination of toolset and object system, but conceptually equal. In the following, we present the conceptual set of tasks, while in Section A 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 [10]. 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 [22]. And finally, at a higher level, software *architects* are interested in maintaining a consistent mapping between the static structure and the domain knowledge. For each of these tasks, an initial step is to locate a particular term or set of terms in the system and assess 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 the one of term *T1*.

A3 **Task.** Evaluate the change impact of class *C* defined in package *P*, by considering its callee 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 either *maintenance* activities 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 [9]. 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 [32] 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 [23]. In spite of the fact that the presence alone of this design problem does not qualify the affected class as harmful [31], 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 and 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 the god class design problem.

B2.1 **Task.** Based on the available design problem information, 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.

4.5 Treatments

By combining the two treatments of each of the two independent variables we obtain four treatment combinations, illustrated in Table 3.

Treatment combination		**Object system size**	
		large	medium
Toolset	CodeCity	T1	T2
	Ecl+Excl	T3	T4

Table 3: Independent variables and resulting treatment combinations

We provided the treatments as virtual images for VirtualBox[7], 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 SP2 operating system.

Each of 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 the class whose representing building is currently selected).

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

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.

4.6 Subjects

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 7.2, 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).

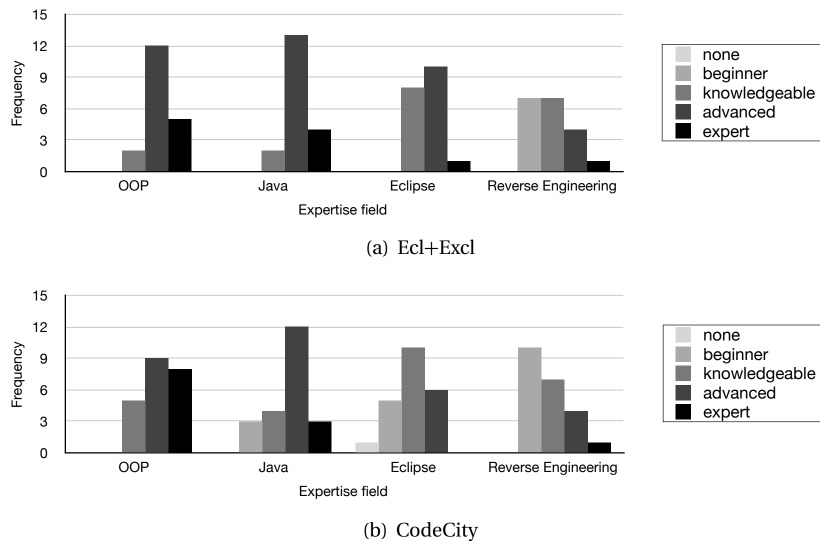
With respect to professional background, our aim was to involve both industry practitioners and people in academia. We managed to obtain valid data for 41 subjects, of which 20 industry practitioners (all advanced), and 21 from academia (of which 9 beginners and 12 advanced). For each of the 4 treatment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

As for the blocks (See Table 4), we obtained a well-balanced distribution of subjects in the four blocks within each treatment.

Number of subjects		**Treatment**				
Blocks		T1	T2	T3	T4	Total
Academy	Beginner	2	3	2	2	9
	Advanced	2	2	3	5	12
Industry	Advanced	6	7	3	4	20
	Total	10	12	8	11	41

Table 4: The number of subjects distributed in treatments and blocks

The random assignments of treatment within blocks led to a fair distribution of the subjects' expertise among treatment combinations, as seen in Figure 2.



(a) Ecl+Excl

(b) CodeCity

Figure 2: Expertise histograms

Only in 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 7.2). 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 2 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 8.7 (i.e., academy versus industry) and Section 8.6 (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 3 show the age of our participants in each of the three blocks: academy-beginner, academy-advanced, and industry-advanced.

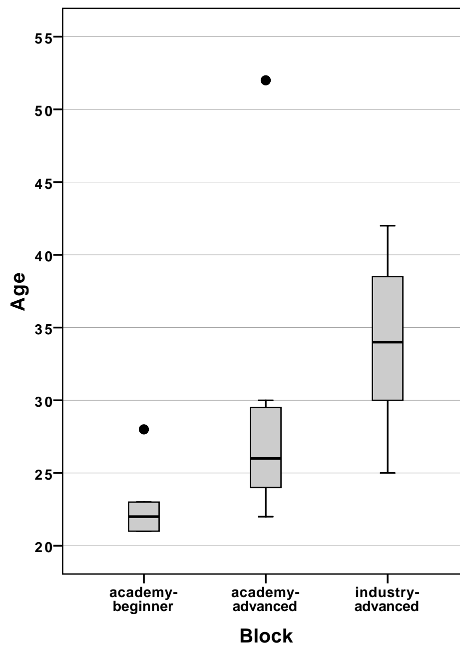


Figure 3: The age of the participant in absolute, and individually, for each of the three blocks

The age of the academy-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 academy-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.

5 Operation

The experiment took place between November 2009 and April 2010 and it consisted in a series of runs. A run consists in a presentation session of about one hour followed by one or more experimental sessions, each taking up to two hours. A presentation session consists in a talk presenting our approach and concluded with a CodeCity tool demonstration, while an experimental session consists in the controlled experiment and a short debriefing. Figure 4 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.

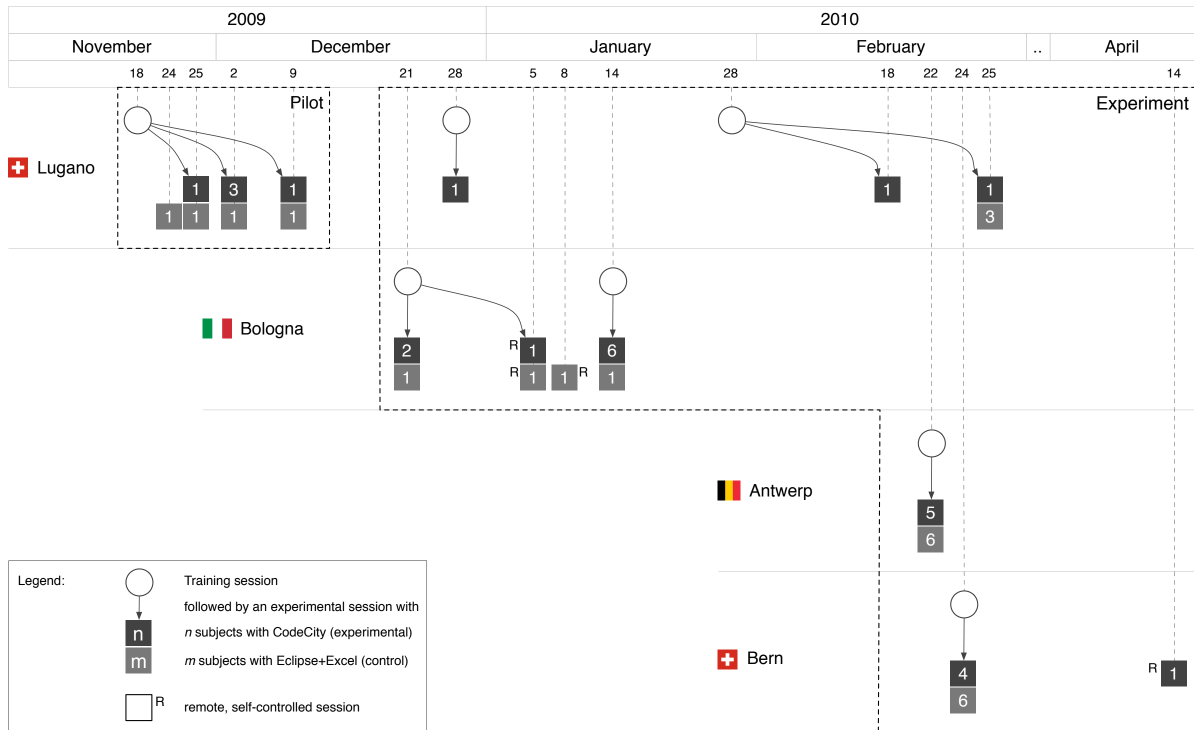


Figure 4: The experiment's timeline

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 arranging four consecutive hours), these two phases of the experimental session were separated by time windows whose length ranged from 20 minutes to one week of time. The numbers 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 7.2.

5.1 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 iteration. 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, which made the first drafts incompatible with the final version.

5.2 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 (and not only) 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.

The first group was composed of practitioners working for several companies from and around Bologna, including Cineca[8], 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, we had to exclude two from the analysis because of the reasons detailed in Section 7.2. In spite of all these, the practitioners reacted quite positively, found CodeCity useful, and were looking forward to use it on their own systems.

The second practitioner group we conducted our experiment with was part of the eXtreme Programming User Group (XPUG) in Bologna[9]. 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 the Jan 14 2010, an interesting thing happened. After the presentation, almost the entire audience remained for the experiment, including the enrolled volunteers we were counting on and 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 condition 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 ad-hoc dynamics of the group. Instead, we chose to observe these enthusiastic groups performing pair-analysis and enjoying every moment, which was one of the most gratifying experiences throughout the experiment.

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.

The third group of industry practitioners we approached was the Java User Group (JUG) in Lugano[10]. 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). First, we went to Antwerp, where we were hosted by Prof. Serge Demeyer and his research group LORE[11]. 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

[8] http://www.cineca.it
[9] http://bo-xpug.homeip.net
[10] http://www.juglugano.ch
[11] http://lore.ua.ac.be

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 would interfere 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.

Only two days after Antwerp, we went to Bern, where we were hosted by Prof. Oscar Nierstrasz and his research group SCG[12]. We performed the experiment session with mostly Ph.D. students and a couple of Master students from the research group. Some of the participants had already seen CodeCity earlier, given that the underlying platform—Moose—was developed in this research group. 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 send us the result. Eventually, we got this final data point in April 2010.

6 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.

6.1 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 the blocking technique, by means of an online questionnaire presented in Section A. 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.

6.2 Timing Data

To measure the completion time, we asked the participants to log the start time, split times, and end time. We learned that this measure alone was not a reliable solution, when several participants who, excited by the upcoming task, simply forgot to write down the split times. In addition, we needed to make sure that none of them would use more than 10 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[13]. 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, the current task, and the maximum remaining time for the current task (See Figure 5).

The subjects were asked to announce 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. If a subject was unable to finish a task in the allotted time, the message "overtime" would appear beside his name and the experimenter would ask the subject to immediately pass to the next task and would reset 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 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 to quickly identify the subjects.

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

[12]http://scg.unibe.ch
[13]http://www.seaside.st

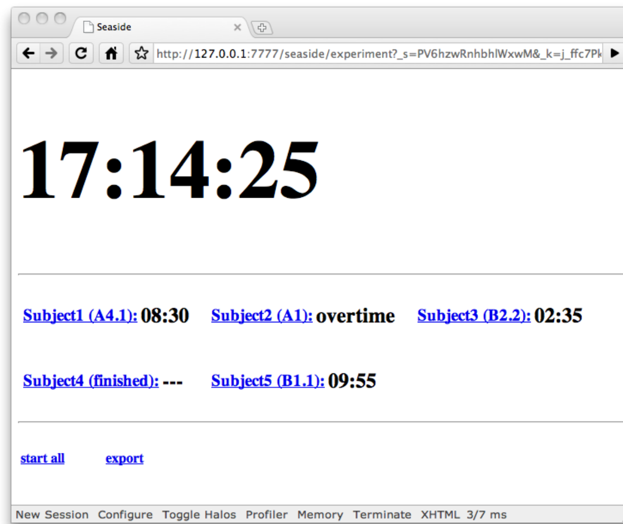


Figure 5: Our timing web application

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 10 in Section D.

6.3 Correctness Data

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

Then, to convert task solutions into quantitative information, we needed an oracle set, which provides 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 control groups use source code, while the experimental groups use a FAMIX model of the object system extracted from the source code, which in practice is never 100% accurate. To obtain the four oracle sets, two of the authors and a third experimenter 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 each task. In addition, for the two experimental groups, we computed the results using queries on the model of the object system, to make sure that we do not miss any information because they are not visible due to the visual presentation (e.g., occlusion, too small buildings, etc.). Eventually, by merging the solution and after discussing the divergences, we obtained the four oracle sets, presented in Section E.

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 does not know whether the subject that provided the solution has used an experimental treatment 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 by him. Then he provided the other two experimenters the encoded data, which allowed them to perform blind grading. In addition, the first author performed his grading unblinded. Eventually, the authors discussed the differences and converged to the final grading, presented in Table 9 of Section D.

6.4 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, interesting insights they encountered during their analysis they wanted to share with us.

7 Data Analysis

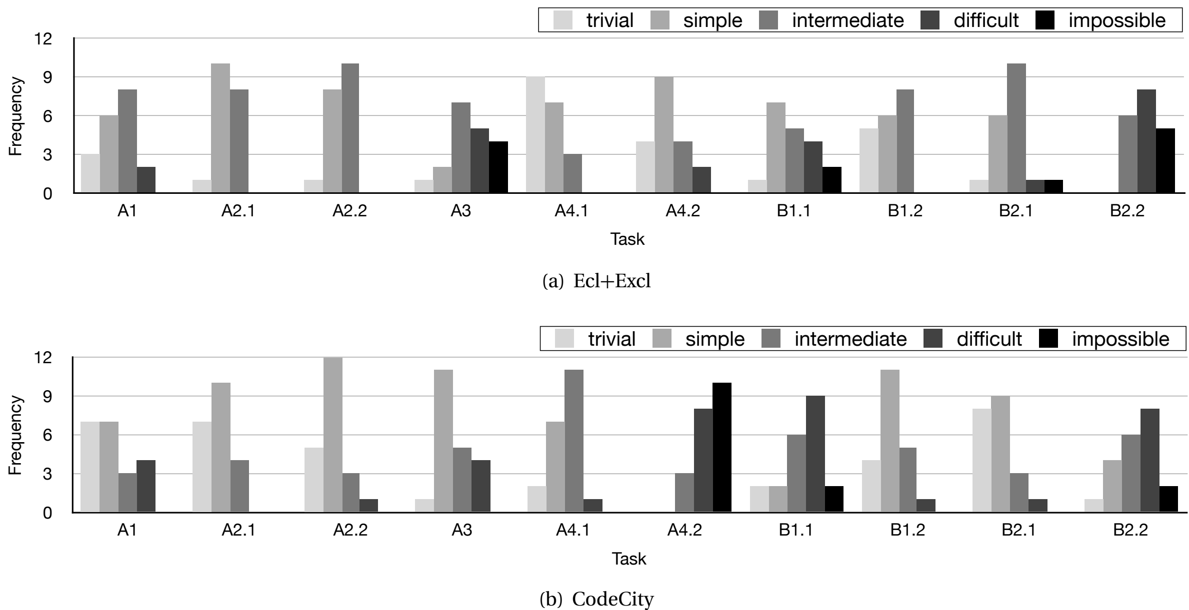
7.1 Preliminary Data Analysis

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 were not able to solve this task, while the control group was quite successful at solving it.

The experimental group had an average correctness score of 0.06. Out of 22 solutions of the experimental group only one achieved a perfect score for this task, while 19 achieved a null score (see the individual scores in Table 9), in spite of the fact that most of the participants used up the entire ten minutes window allotted for the task (see the completion times in Figure 10). It turned out that the only perfect score was provided by a participant who had used CodeCity on several previous occasions and he, as a central figure in the Moose community, 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 ones. They were able to complete the task in roughly half the allotted time 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 6).



(a) Ecl+Excl

(b) CodeCity

Figure 6: Task perceived difficulty histograms

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

7.2 Outlier Analysis

Before performing our statistical test, we also followed the suggestion of Wohlin et al. [49] regarding the removal of outliers caused by exceptional conditions, in order to enable us to draw valid conclusion 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 therefore, 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 second Bologna run, two of the participants had incompatibility problems with the virtualization software under the operating system 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.

7.3 Analysis Techniques

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[14] 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 [19] 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 [34], 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 typical significance level of .05 (i.e., $\alpha = .05$), which corresponds to a 95% confidence interval.

8 Results

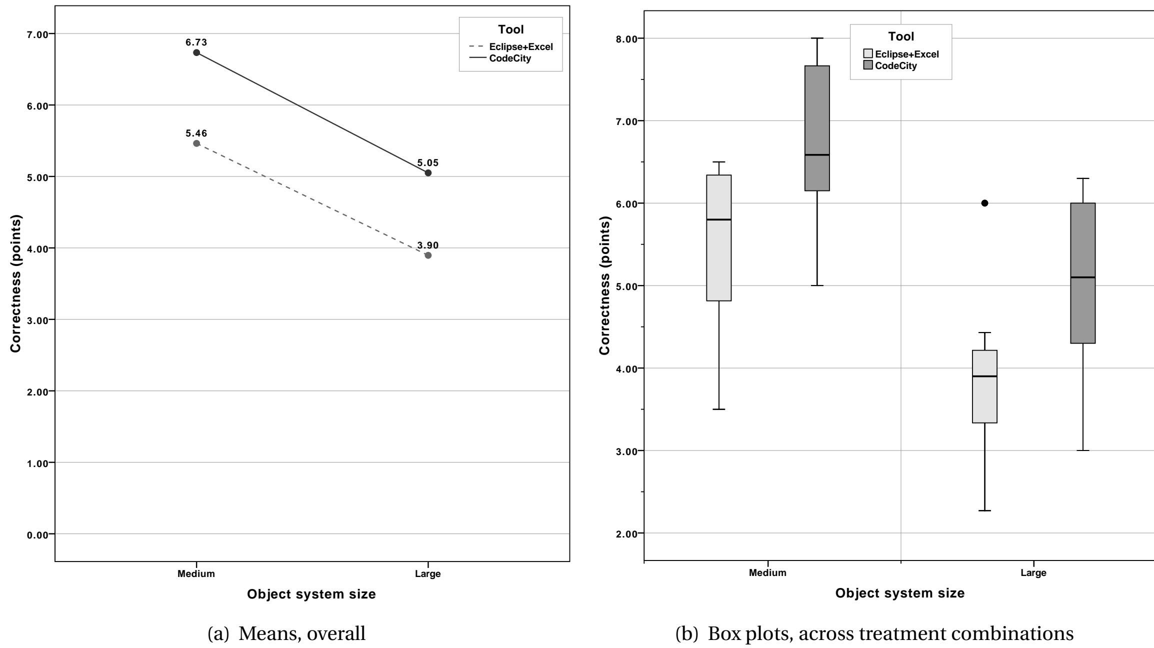
We present the results of the analysis separately for each of the two dependent variables, i.e., correctness and completion time. Apart from the effect of the main factors, i.e., tool and system size, the ANOVA test allows one to test the interaction between the two factors.

8.1 Analysis Results 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.

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.

[14] http://www.spss.com



(a) Means, overall

(b) Box plots, across treatment combinations

Figure 7: Graphs for correctness

System size	*medium*		*large*		*any*	
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	3.500	5.000	2.270	3.00	2.270	3.000
max	6.500	8.000	6.000	6.30	6.500	8.000
median	5.800	6.585	3.900	5.100	4.430	6.065
stdev	1.147	0.959	1.085	1.031	1.349	1.294

Table 5: Descriptive statistics related to 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.

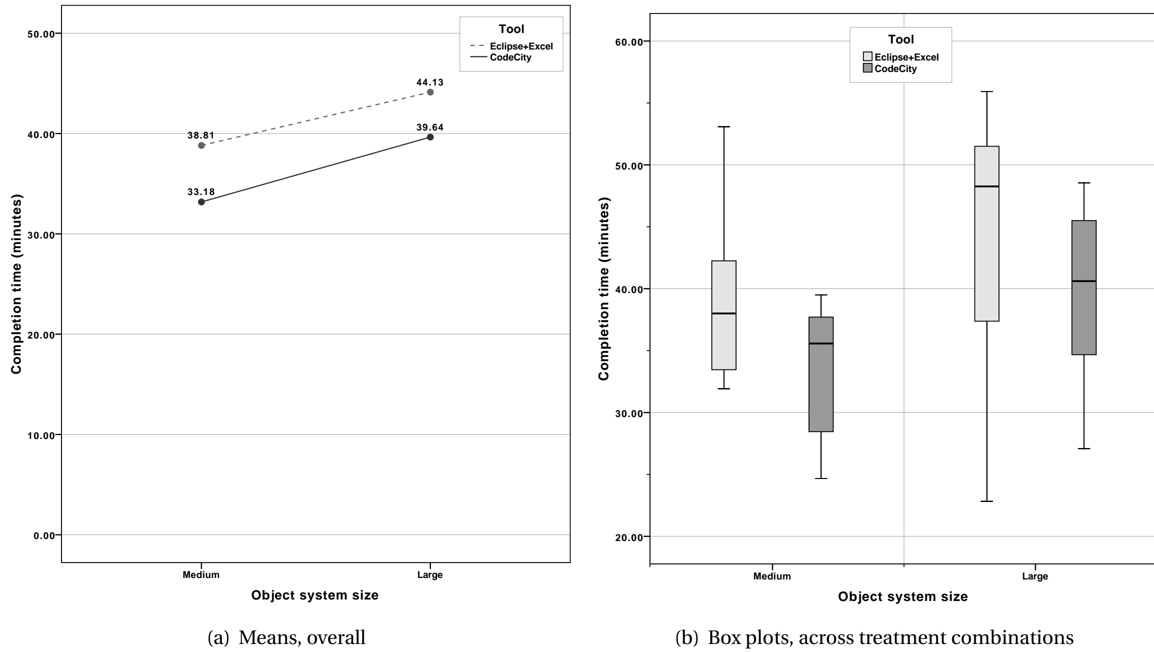
Although not the object of the experiment, an expected significant main effect of system size on the correctness of the solutions was observed, $F(1,37) = 26.453$, $p < .001$, indicating that the correctness score was significantly higher for users performing the analysis on the medium size system than 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, as well as the lack of the effect of interaction between tool and object system size on correctness, are illustrated in Figure 7(a). The correctness box plots for each combination of treatments are presented in Figure 7(b) and a detailed description of the statistics related to correctness is given in Table 5.

8.2 Analysis Results 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.

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



(a) Means, overall (b) Box plots, across treatment combinations

Figure 8: Graphs for completion time

System size	*medium*		*large*		*any*	
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

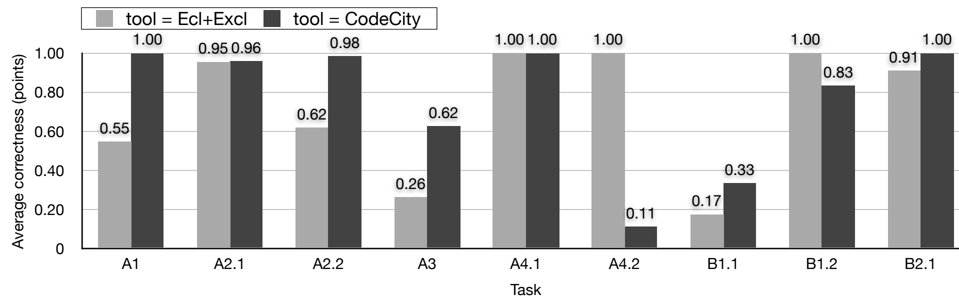
Table 6: Descriptive statistics related to completion time, in minutes

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 indicates 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.

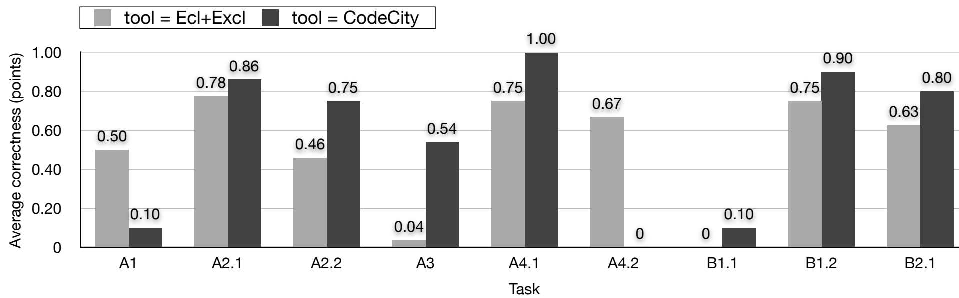
The analyzed data allows us to reject the second null hypothesis $H2_0$ in favor of the alternative hypothesis $H2$, which states that the tool impacts the time required to complete program comprehension tasks. Overall, CodeCity enabled a reduction of the completion time of 12.01% over Ecl+Excl.

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.

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(a). The completion time box plots for each combination of treatments are presented in Figure 8(b), while a detailed description of the statistics related to completion time is given in Table 8(b).



(a) System size = medium



(b) System size = large

Figure 9: Average correctness per task

8.3 Task Analysis

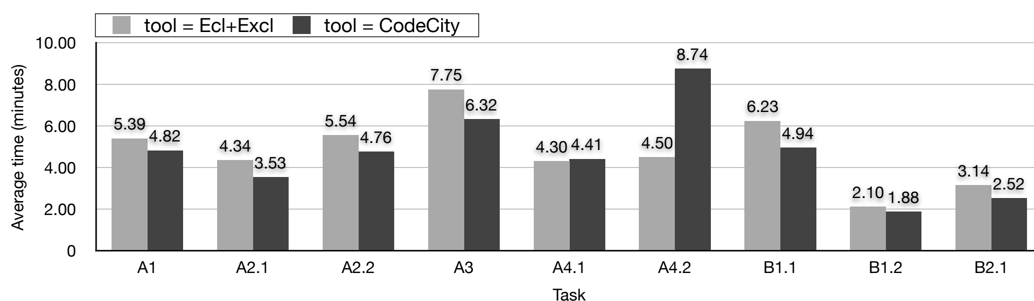
One of the research goals of our experiment was to identify the types of tasks for which CodeCity provides an advantage over Ecl+Excl. To this end, we compared for each task described in Section 4.4 the performances (i.e., in terms of correctness and time) of the two levels of the *tool* and reasoned about the potential causes behind the differences. See Figure 9 and Figure 10 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 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.

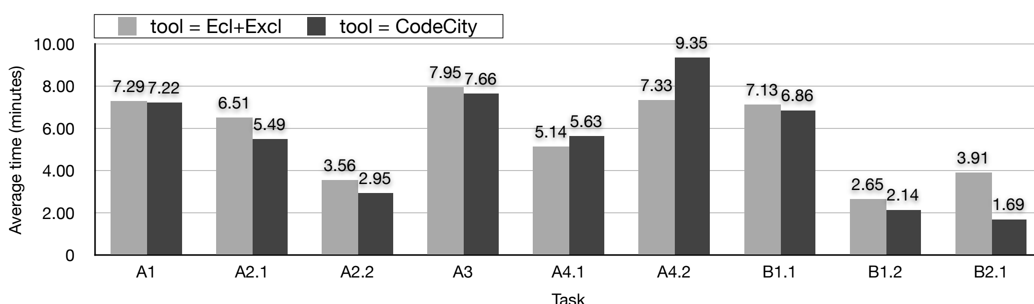
A2.1 - Determining the spread of a term among the classes. CodeCity performed only marginally better than Eclipse in both correctness and completion time. In CodeCity once the search for the term is completed, finding any kind of spread is straightforward with the overview. In Eclipse, the search for 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.

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 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 callee classes of a given class in Eclipse, as opposed to CodeCity, is not straightforward and the result list provides no overview.



(a) System size = medium



(b) System size = large

Figure 10: Average time per task

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 7.1. We 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, i.e., when handling 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.

8.3.1 Conclusions of the Quantitative Task-Based Analysis

As expected, at focused tasks such as *A4.1*, *A4.2*, or *B1.1* CodeCity does not perform better than Ecl+Excl, because Excel is extremely efficient in finding precise answers (e.g., the largest, the top 3, etc.). However, it is surprising that, in most of these tasks, CodeCity managed to be on a par with Ecl+Excl. 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.

8.4 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 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 with the ones obtained with Eclipse 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 for the assessment (see Section 4.4), we encouraged the subjects to share with us the most interesting or unexpected observations regarding the design problems, in this open-ended task.

8.4.1 Ecl + Excl

As expected, many of the 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)

8.4.2 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)

8.4.3 Conclusions of the Qualitative Task-Based Analysis

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".

8.5 Debriefing Questionnaire

In the following we briefly summarize 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 9. Two other subjects in the control group said they hated the search functionality in Eclipse.

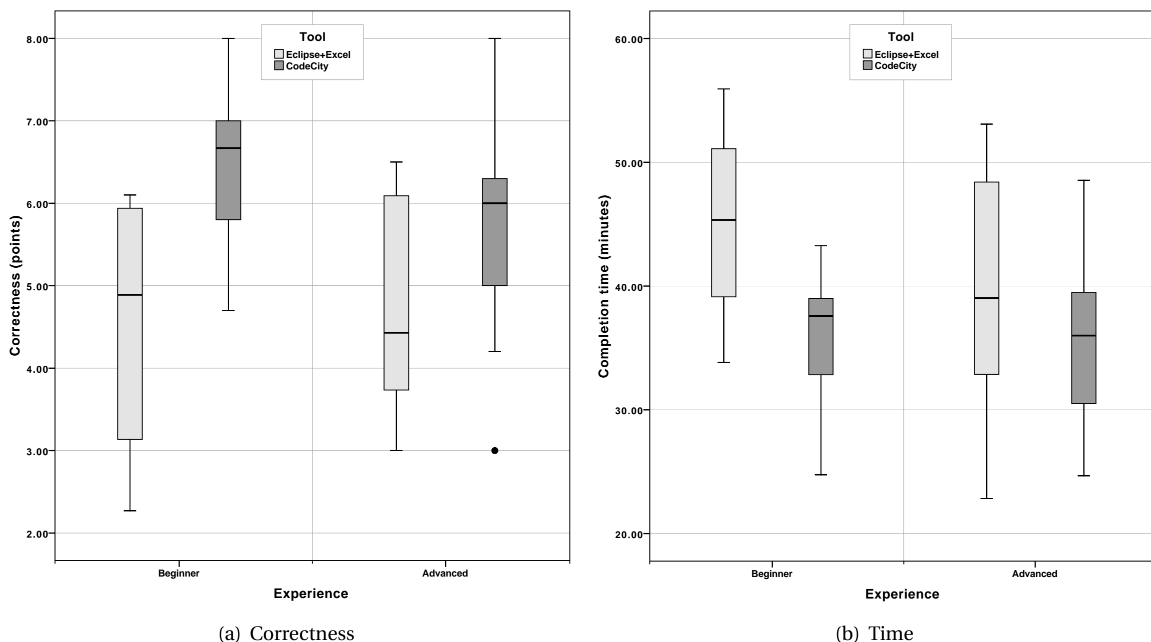
Two subjects praised the setup and the organization of the experimental runs. One subject found the experiment very stimulating. Several industry developers expressed their interest in using CodeCity in their line of work after the experiment.

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 Experience Level



(a) Correctness

(b) Time

Figure 11: Beginner versus Advanced

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 11.

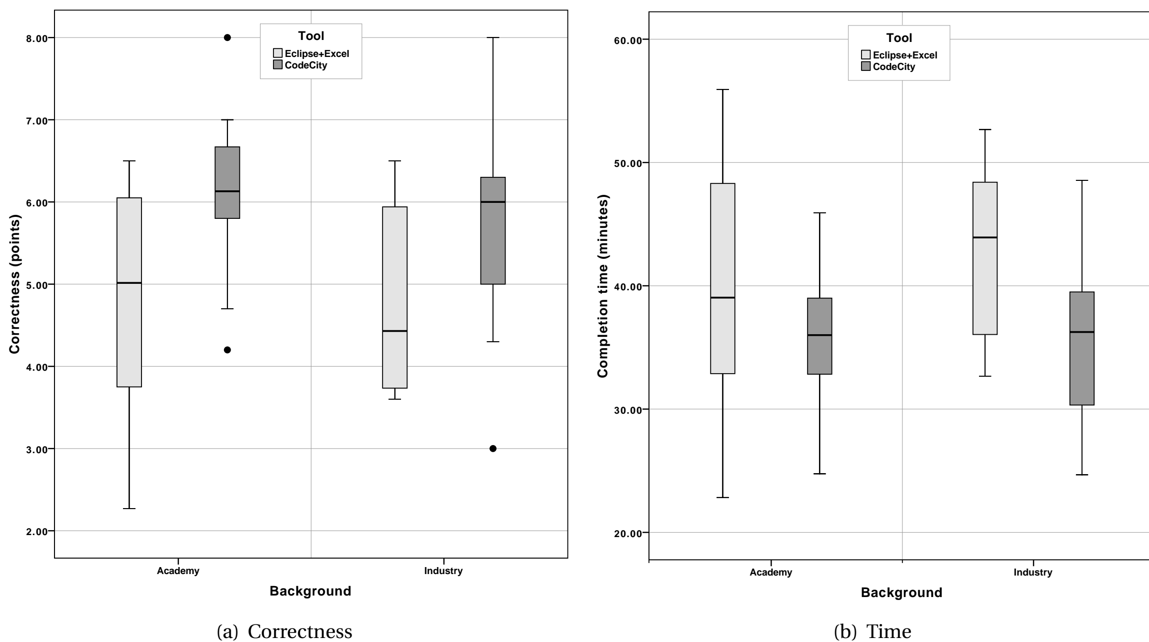
An interesting observation is that CodeCity users have much less variability in performance than the users of Ecl+Excl, which shows a more consistent performance of CodeCity compared to Ecl+Excl. This can be assessed visually, as the boxes of the box plots for CodeCity are much smaller than the one for 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 supports obtaining better results than with conventional non-visual approaches, even without extensive training.

8.7 Background



(a) Correctness (b) Time

Figure 12: Academy versus Industry

We also compared the correctness and time scores across the two levels of background, i.e., academy and industry. The performance for CodeCity is better in both correctness and completion time, regardless of the background, as shown in Figure 12.

Again, the box plots of CodeCity users (in particular the ones from academy) have shorter boxes than the ones of Ecl+Excl, which shows 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 academy than for industry.

In terms of completion time, the difference between CodeCity and Ecl+Excl is minimal in the case of academy 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 the industry practitioners.

9 Threats to Validity

In this section, we discuss the threats to our experiment's validity. For experiments validating applied research the categories are—in decreasing order of importance: internal, external, construct and conclusion validity [49].

9.1 Internal Validity

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

Subjects. Several of the threats to internal validity refer to the experiment's subjects. A first threat was 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 the subjects 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 and, in addition, showed a ceiling effect (i.e., most subjects used up the entire time) for one group and rather trivial for the other.

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 works with a FAMIX model of the system, while Eclipse works with the source code (although in reality Eclipse has its own proprietary model). 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 that 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.

9.2 External Validity

The external validity refers to the generalizability of the experiment's results.

Subjects. A 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 these categories. Eventually, we obtained a balanced mix of academics (both beginners and advanced) and industry practitioners (only advanced). The lack of industry beginners may have an influence on the results. However, our analysis of the performances across experience levels indicates that CodeCity supported well beginner users, who obtained even better results than the advanced users in outperforming Ecl+Excl. Therefore, we believe that the presence of industry beginners could have 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 none of the existing frameworks, because they do not support design problem assessment and, in addition, they are either a) too low-level, such as the set of questions asked by practitioners during a programming change task set synthesized by Sillito et al. [35], or b) biased towards dynamic analysis tools as the framework of comprehension activities compiled by Pacione et al. [26]. To alleviate this threat, we described the tasks in the context of reverse engineering and high-level program comprehension scenarios.

Object systems. The representativeness of the object systems is another threat. In spite of the increased complexity in the organization and analysis of the experiment 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 [30] 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 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. However, we mitigate this threat in several ways. When building the oracle set, three experimenters created their own oracle set independently and then converged through discussions to a common set. When grading the correctness scores, three experimenters independently graded the solutions and again converged to a common set through discussions. Moreover, two of the three performed the grading blinded, i.e., without knowing whether they are grading a solution obtained with CodeCity or with Ecl+Excl.

9.3 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 of the fact that we were the authors of CodeCity and that the purpose of the experiment was to compare the performance of CodeCity with a baseline. To alleviate this threat, we clearly explained them before each experiment session that it is the tool's support that is 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 was obtained by a control subject (i.e., AA07) and one of the best correctness scores in the case of the large object system was obtained by another control subject (i.e., AA05).

9.4 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, 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.

10 Conclusions

The contributions of this paper are threefold:

1. We performed a literature study and extracted a wish list for our experiment, which allowed us to put in practice the lessons we learned from the body of related work.
2. We designed and performed a controlled experiment for the evaluation of our approach, which showed that, at least for the tasks assigned to our subjects, our approach outperforms in both correctness and completion time the combination of two state-of-the-practice exploration tools, Besides an aggregated analysis, we provide a detailed analysis of each task, as well as comments on the last task of the experiment, which was more qualitative in nature, as it was focused on gathering unexpected insights about the systems under study.
3. Since we believe that other researchers interested in evaluating their tools should benefit from our experience, we provided the complete raw data and other details (i.e., the pre-experiment questionnaire, the experiment questionnaires, solution oracles and grading systems), which allow reviewers to better evaluate the experiment and fellow researchers to repeat the experiment or start from its design as a base for their own experiment.

Acknowledgements

We thank Prof. Radu Marinescu, Mircea Lungu, Alberto Bacchelli, and Lile Hattori for helping us with the design of the experiment. We also thank Prof. Oscar Nierstrasz, Prof. Serge Demeyer, Fabrizio Perin, Quinten Soetens, Alberto Bacchelli, and Sebastiano Cobianco for helping us with the local organization of the experimental sessions. Last, but not least, we thank all the subjects of our experiment: the developers in Bologna and Lugano, the Software Composition Group in Bern, and the Master students in both Lugano and Antwerp.

References

[1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.

[2] T. Barlow and P. Neville. A comparison of 2-d visualizations of hierarchies. In *InfoVis '01: Proceedings of the 2001 IEEE Symposium on Information Visualization*, pages 131–138. IEEE Computer Society Press, 2001.

[3] A. Cooper and R. Reimann. *About Face 2.0 - The Essentials of Interaction Design*. Wiley, 2003.

[4] B. Cornelissen, A. Zaidman, B. V. Rompaey, and A. 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.

[5] B. Cornelissen, A. Zaidman, A. van Deursen, and B. Van Rompaey. Trace visualization for program comprehension: A controlled experiment. Technical Report TUD-SERG-2009-001, Delft University of Technology, 2009.

[6] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[7] G. Ellis and A. 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.

[8] M. Granitzer, W. Kienreich, V. Sabol, K. Andrews, and W. Klieber. Evaluating a system for interactive exploration of large, hierarchically structured document repositories. In *InfoVis '04: Proceedings of the 2004 IEEE Symposium on Information Visualization*, pages 127–134. IEEE Computer Society Press, 2004.

[9] T. Gyimóthy, R. Ferenc, and I. 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.

[10] S. Haiduc and A. 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.

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, December 2004.

[12] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[13] J. Knodel, D. Muthig, and M. Naab. An experiment on the role of graphical elements in architecture visualization. *Empirical Software Engineering*, 13(6):693–726, 2008.

[14] A. 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.

[15] A. 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.

[16] R. Kosara, C. Healey, V. Interrante, D. Laidlaw, and C. Ware. User studies: Why, how, and when? *IEEE Computer Graphics and Applications*, 23(4):20–25, July-Aug. 2003.

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

[18] C. F. J. Lange and M. 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.

[19] H. Levene. Robust tests for equality of variances. In I. Olkin, editor, *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, pages 278–292. Stanford University Press, 1960.

[20] J. Maletic and A. 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.

[21] A. Marcus, D. Comorski, and A. 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.

[22] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. 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.

[23] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2004.

[24] O. Nierstrasz, S. Ducasse, and T. 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.

[25] R. O'Donnell, A. Dix, and L. 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.

[26] M. J. Pacione, M. Roper, and M. 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.

[27] M. D. Penta, R. Stirewalt, and E. 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.

[28] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[29] C. 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.

[30] J. 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.

[31] D. Raţiu, S. Ducasse, T. Gîrba, and R. 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.

[32] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[33] M. Sensalire, P. Ogao, and A. 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.

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

[35] J. Sillito, G. C. Murphy, and K. 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.

[36] J. Stasko. An evaluation of space-filling information visualizations for depicting hierarchical structures. *Int. J. Hum.-Comput. Stud.*, 53(5):663–694, 2000.

[37] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Control*, 14(3):187–208, 2006.

[38] M.-A. D. Storey, H. Müller, and K. Wong. Manipulating and documenting software structures. In P. D. Eades and K. Zhang, editors, *Software Visualisation*, volume 7, pages 244–263. World Scientific Publishing Co., 1996.

[39] M.-A. D. Storey, K. Wong, and H. 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.

[40] Y. Wang, S. T. Teoh, and K.-L. Ma. Evaluating the effectiveness of tree visualization systems for knowledge discovery. In *In Proceedings of Eurographics Visualization Symposium*, pages 67–74. Eurographics Association, 2006.

[41] R. Wettel. Scripting 3d visualizations with codecity. In *FAMOOSr '08: Proceedings of the 2nd Workshop on FAMIX and Moose in Reengineering*, 2008.

[42] R. Wettel and M. 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.

[43] R. Wettel and M. 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.

[44] R. Wettel and M. Lanza. CodeCity. In *WASDeTT '08: In Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[45] R. Wettel and M. 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.

[46] R. Wettel and M. 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.

[47] R. Wettel and M. 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.

[48] 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.

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

[50] Y. Zhu. Measuring effective data visualization. In *ISVC (2)*, volume 4842 of *Lecture Notes in Computer Science*, pages 652–661. Springer, 2007.

A Pre-Experiment Questionnaire

Using Google Docs[15], 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 13).

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	○	○	○	○	○

Submit

Figure 13: The enrollment online questionnaire we used for collecting personal information about the participants

B Experiment Questionnaire

The content of the questionnaires, with all the variations due to the different treatment combinations, is presented in the following. The actual form and presentation of the questionnaire is presented in Figure 14 and Figure 15, which shows a series of the actual pages of the questionnaires for the *T1* combination of treatments.

B.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>[16] to analyze <object system name>[17], a <object system description>[18] 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;

[15] http://docs.google.com
[16] *CodeCity* for treatments 1 and 2, *Eclipse + Excel with CSV data concerning metrics and design problems* for treatments 3 and 4
[17] *Azureus* for treatments 1 and 3, *FindBugs* for treatments 2 and 4
[18] *a BitTorrent client* for treatments 1 and 3, *a bug searching tool based on static analysis* for treatments 2 and 4

- 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

B.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.

Solution (*multiple choice*)

- Centralized. There is a single package hierarchy, whose root package is (write down the full name of the package): . . . [19].
- 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>[20] (and any other) feature in <toolset>, look for the term <term 1>[21] 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>[22] in the names of the classes and their attributes and methods, and describe the spread of these classes in the system[23].

[19] The placeholders presented here are not proportional in length to the variable-size blanks used in the actual questionnaires.
[20] *search by term* for treatments 1 and 2, *Java search* for treatments 3 and 4
[21] *skin* for treatments 1 and 3, *annotate* for treatments 2 and 4
[22] *tracker* for treatments 1 and 3, *infinite* for treatments 2 and 4
[23] The task is similar to the previous one, but the terms are chosen such that they cover the opposite solution.

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: . . . .

A3 [Impact Analysis]

Task
Evaluate the change impact of class <class A3>[24], by considering its callee 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:

$$\text{lines of code per method} = \frac{\text{number of lines of code}}{\text{number of methods}}$$

Solution *(ranking)*
The classes with the highest average number of lines of code per methods are (in descending order):

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.

[24] `org.gudy.azureus2.ui.swt.Utils` for treatments 1 and 3, `edu.umd.cs.findbugs.OpcodeStack` for treatments 2 and 4

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>[25], 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.

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)*
. . .

C 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.

[25] *CodeCity* for treatments 1 and 2, *the spreadsheet* for treatments 3 and 4

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[26]:
...

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).
...

D Data

To provide a fully transparent experimental setup, we make available the entire data set of our experiment.

In Table 7 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 8, clustered by the treatment combination, to ease comparison between the different levels of the independent variables.

Using the criteria described in detail in Section 6.3, we obtained the correctness levels presented in Table 9. Based on the reasoning presented in the Section 7.2, 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 10. 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 11 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.

[26]The scale for difficulty was, in decreasing order: impossible, difficult, intermediate, simple, trivial

Code	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 7: The subjects' personal information, clustered by treatment combinations.

Code	Treatment			Blocking Criteria	
	Number	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	academy	beginner
AB02	1	CodeCity	large	academy	beginner
IA05	1	CodeCity	large	industry	advanced
AA01	1	CodeCity	large	academy	advanced
AA02	1	CodeCity	large	academy	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	academy	beginner
IA10	2	CodeCity	medium	industry	advanced
IA11	2	CodeCity	medium	industry	advanced
IA12	2	CodeCity	medium	industry	advanced
AB04	2	CodeCity	medium	academy	beginner
AA03	2	CodeCity	medium	academy	advanced
AB05	2	CodeCity	medium	academy	beginner
AA04	2	CodeCity	medium	academy	advanced
IA13	2	CodeCity	medium	industry	advanced
IA14	3	Ecl+Excl	large	industry	advanced
AB06	3	Ecl+Excl	large	academy	beginner
AB07	3	Ecl+Excl	large	academy	beginner
AA05	3	Ecl+Excl	large	academy	advanced
AA06	3	Ecl+Excl	large	academy	advanced
AA07	3	Ecl+Excl	large	academy	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	academy	beginner
AB09	4	Ecl+Excl	medium	academy	beginner
AA10	4	Ecl+Excl	medium	academy	advanced
AA11	4	Ecl+Excl	medium	academy	advanced
AA12	4	Ecl+Excl	medium	academy	advanced
AA13	4	Ecl+Excl	medium	academy	advanced
AA14	4	Ecl+Excl	medium	academy	advanced
IA20	4	Ecl+Excl	medium	industry	advanced

Table 8: Subjects assigned to treatments and blocks

Code	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 9: Correctness of subjects' solutions to the tasks

Code	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 10: Completion time in minutes

Code	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 11: The subjects' perceived *time pressure* and task *difficulty*.

E Task Solution Oracles

The four oracles we used to grade the task solutions of our subjects are presented in the following.

E.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.

E.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.

E.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.

E.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.

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: _ (×5).

(e) Time split (repeated 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: _ (×5).

Impact Analysis Task A3

Evaluate the change impact of class **Utils** defined in package **org.gudy.azureus2.ui.swt**, by considering its callee 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: _ (×5).

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 14: Handout for Treatment 1 (1/2)

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 15: Handout for Treatment 1 (2/2)