Do Dynamic Object Process Graphs Support Program Understanding? – A Controlled Experiment.

Jochen Quante
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
quante@informatik.uni-bremen.de

Abstract

Using automatic program analysis techniques for extracting architectural information and its visualization is widely considered useful for program understanding. However, it has to be empirically validated if a given technique is beneficial in practice. This is usually done by performing a set of case studies. To find out for sure whether a technique really has any effect, controlled experiments have to be conducted.

Dynamic object process graphs are one such technique. These graphs describe the control flow of an application from the perspective of a single object. In previous research, we conducted case studies which indicated that they may be useful for program understanding, but this assumption has not been validated so far.

We report on a controlled experiment which investigated this question: Does the availability of such graphs support program understanding or not? We describe the research questions that were investigated, the hypotheses, experimental setup, conduction, and discuss the results and lessons learned.

1 Introduction

Software maintenance is an expensive and time-consuming task. About 50% of the time in software development and maintenance is spent for gaining an understanding of the system to be modified [2]. Automated techniques for supporting a software maintainer in the program understanding task can potentially help to reduce this effort, making software maintenance more effective. Such techniques may even provide him with a *better* understanding of the system, reducing the number of errors introduced in maintenance. Many different techniques have been proposed for this purpose, and some of them are used in practice. Examples for such techniques are metrics calculation, program slicing, UML diagram extraction, and program visualization in general.

To assess the advantages and disadvantages of any such technique, empirical experiments have to be conducted. This is typically done in the form of case studies, observations, surveys, or controlled experiments. Di Penta et al. [4] give a good overview of this topic and the problems associated with it. They also cite a lot of examples where empirical studies in the context of program comprehension have been conducted. However, controlled experiments for assessing the benefit of a given tool are still performed very rarely. This paper describes one such experiment.

In a series of papers [6, 7, 5], we proposed a novel technique called "Dynamic Object Process Graphs" (DOPG). It extracts a projection of a program's control flow graph (CFG) from the perspective of a single object, and it only retains those parts of the CFG that have been executed. This technique is intended to provide an understanding of how a component is being used in an application. If the investigated object is of central concern for the application, these graphs may even give a first impression of the entire application's structure. An example for such an object is the socket in a chat application, or the file object in a database engine. In a couple of case studies, we collected evidence that such graphs can be helpful for program understanding.

It is yet to be shown that such graphs really support a software maintenance engineer in his work. Therefore, we conducted a controlled experiment to verify whether DOPGs really help in program understanding, or rather in certain tasks of program understanding.

Contributions. This paper describes and discusses a controlled experiment on the usefulness of DOPGs for program understanding: The research questions that were investigated, the experimental setup, conduction, results and lessons learned.

Overview. The remainder of this paper is organized as follows. Section 2 introduces Dynamic Object Process Graphs, Section 3 describes the experiment in detail, and Section 4 presents and discusses the results.

2 Dynamic Object Process Graphs

Dynamic Object Process Graphs (DOPGs) have been introduced in detail in previous papers of ours [6, 7]. In the following, we only provide a short summary of their construction and meaning.

An *interprocedural control flow graph* (CFG) is a well-known program representation that is commonly used in compiler technology and reverse engineering. A CFG's edges describe the order in which statements of a program (which are represented by the nodes) may be executed. A CFG is for example the basis for the program dependency graph, which in turn is used for program slicing. Extraction of the CFG is a standard static program analysis. However, the resulting graphs are extremely large and therefore not directly usable for program comprehension or even visualization.

The idea of DOPGs is to reduce a given CFG with respect to a given object. A DOPG retains only those nodes (and adjacent edges) of the CFG that comply to the following conditions:

1. The node has been executed in a given use case or set of use cases (dynamic analysis).
2. The node is *relevant* with respect to the given object. This is the case if
   - it represents an operation on the object (*atomic* node, such as creation, attribute access, or method call),
   - there exists a relevant node that is control dependent on it, i. e., it depends on this node whether the other node is executed or not. This applies for condition and call nodes.

Figures 1 and 2 show an example for the construction of a DOPG. Figure 1 contains all nodes of the CFG for a given program. After removing nodes that have never been executed (dashed) and nodes that are not relevant for the object (crossed out), only the DOPG remains. The final result is shown in Figure 2.

When generating DOPGs, the *object* is determined by choosing a *relevant class*. One DOPG is extracted for each static allocation point of this class that is executed. If multiple objects are instantiated at the same location in the code, they result in only one DOPG.

To make DOPGs available to programmers in practice, we created a DOPG viewer Eclipse plugin. It has the following features: Load DOPG, perform a spring layout algorithm, zoom in and out, stretch and tighten layout, pan (drag the viewport), move nodes, find *start* node (start of application) and find *create* node (object creation). Another very important function is the possibility to jump to the corresponding source code location of a node by double-clicking on it. A screenshot of the plugin is shown in Figure 3.

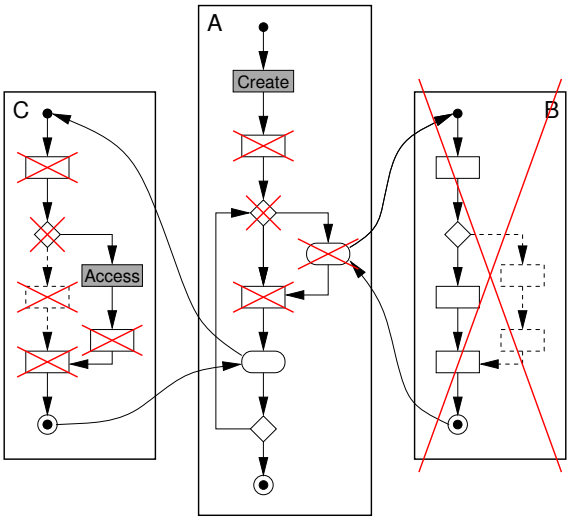

Figure 1. CFG based DOPG construction example. A/B/C are functions, gray nodes denote atomic nodes, and dashed nodes have never been executed. The crossed parts of the graph are removed in the next step.

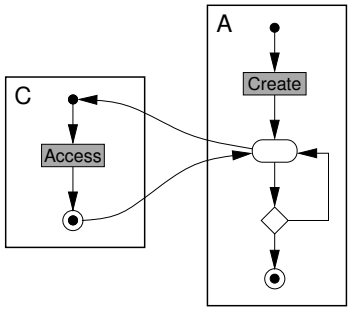

Figure 2. Final DOPG as constructed from the CFG in Figure 1.

3 Description of the Experiment

In the past, we conducted several case studies which indicated that DOPGs can be helpful for program understanding tasks [6, 7]. In particular, we investigated two IRC chat clients and found that DOPGs for the socket object give a good impression of the applica-

tion's overall structure. A similar result was obtained for the file handle object of a database system. This raised the question of whether such graphs are helpful for program understanding in general, or which tasks can be efficiently supported by using them. To clarify this question, we conducted a controlled experiment which is described in the following.

3.1 Hypotheses

We pose the following **research questions**:

- Does the availability of visualized DOPGs lead to faster answers to program understanding questions that are in some way related to a given component?
- Are these questions answered less erroneously?

Our **hypothesis** is that both questions can be answered positively:

H1 When DOPGs are available, questions that are related to a given component are answered faster.

Rationale: DOPGs represent an application's control flow from a single component's perspective. This information should be helpful for understanding how the component is being used. The maintainer can focus on those spots in the code that are relevant and sees their relation immediately. He does not have to browse through a lot of code in search for uses of the component or for the connection between these uses. DOPGs also contain call paths to the points where the component is used, which should give a good impression about the basic structure of relevant parts of the application. This should significantly reduce the time needed for finding out about component-related dependencies.

H2 When DOPGs are used, such questions are answered less erroneously.

Rationale: When the DOPGs are suited to the given question, i. e. represent an adequate use case, it provides detailed information about the component's relevant use. Since it concentrates all the neccessary information, the maintainer should get a very good understanding of the dependencies of the component. Consequently, he should be able to gain a deeper understanding of the component's use and to answer corresponding questions less erroneously.

The corresponding null hypothesis is:

H0 It does not make a difference in program understanding whether DOPGs are available or not.

Group	Size	Sys. 1	DOPG	Sys. 2	DOPG
1a	6	Gantt	yes	Argo	no
1b	7	Argo	no	Gantt	yes
2a	6	Gantt	no	Argo	yes
2b	6	Argo	yes	Gantt	no

Table 1. Experimental (sub)groups and sizes: Order of systems and DOPG availability.

3.2 Experiment Design

In order to validate or reject these hypotheses, we let two groups of subjects solve the same tasks. One of these groups (*control group*) worked with standard techniques only, while the other group (*experimental group*) additionally used DOPGs. The **independent variable** in our controlled experiment was the availability of visualized DOPGs. The experiment had a *between-participants after-only design* [1]. This means that each participant was either member of the experimental group or member of the control group.

Due to the relatively low number of participants that was to be expected, a *within-participants* design would have been preferable, but was difficult to implement in this case. Giving the same program understanding task twice would obviously lead to strong sequencing effects. It is difficult to find two different subject systems that are equivalent with respect to the tasks, or to find two different but still comparable tasks. Therefore, we decided to let each participant investigate two different systems with different tasks: For one system, the participant was assigned to the experimental group, and for the other system, he was assigned to the control group. The order of the two systems and which of the systems had DOPGs available was randomly assigned for each participant. This setup results in four combinations and therefore four subgroups as shown in Table 1. It can be considered as performing two interleaved experiments.

The **dependent variables** measure the effect of controlled variation of the independent variable and should help answering the research questions. In our case, dependent variables were:

- the time needed to answer each question,
- the correctness of these answers, and
- the subjective user satisfaction/confidence/productivity.

The first variable was measured while the tasks were performed, the second one was determined after the experiment was finished, and the third one was acquired in a post-mortem questionnaire.

System	Files	LOC	SLOC	Class	APs	nodes	DOPG	edges
Jetris	12	1,885	1,527	Figure (subclasses)	14	16/21/94		21/31/144
GanttProject	475	61,892	43,100	GanttTask	3	66/293/661		84/409/963
ArgoUML	1,725	319,797	160,509	ClassDiagramGraphModel	1	167		237

Table 2. Characteristics of the subject systems. Jetris was only used for training. The middle part of the table shows the preselected relevant class and the number of used allocation points, and the right part contains the corresponding DOPG size measures (minimum/median/maximum counts).

Relevant **extraneous variables** were:

- the participant's experience in programming and software maintenance, in particular in Java;
- participant's familiarity with the subject systems;
- participant's familiarity with the environment;
- experimenter effects: how the participants are instructed, what the experimenter expects from the experiment, etc.;
- instrumentation: how the dependent variables are measured.

Differences between most of these extraneous variables were cancelled out by randomized assignment of participants to groups. Experimenter and instrumentation effects will be discussed in Section 3.5.

3.3 Subjects

All participants were computer science students from the University of Bremen who have already received the intermediate diploma ("Vordiplom"), which means that they have all participated in a one-year project with 5 to 7 people and developed a Java system of several thousand lines of code. Another precondition they had to meet was basic knowledge in using Eclipse for Java development. 35 students replied to a call for participation which was sent out to all computer science students at the University of Bremen. 27 of these participated in the experiment. Two of them took part in a pilot experiment which was conducted to adjust tasks and timing of the experiment. This left 25 participants for the main experiment.

The subjects had between 1 and 10 years (one outlier had 25 years) of programming experience (median 5, std. dev. 2.7) and between 1 and 8 years of Java experience (median 2.5, std. dev. 1.4). 60% of the participants rated their own programming capabilities as being "above average". The largest system the students had worked on before was between 1 and 200 KLOC in size (median: 17 KLOC). Six of the students reported maximum sizes below 10 KLOC. Half of the participants did not have much experience in working on systems written by others, but the other half did. The distributions of these properties were similar in the two experimental groups (experimental/control group per system). This was the result of randomization; it was not actively enforced (no *merging*).

3.4 Experiment Tasks

The choice of subject systems was performed based on the following considerations:

- Complex/large enough to be realistic.
- Application is roughly known to the participants from a user's perspective.
- Code is unknown to the participants.
- Written in Java, since this is the language that all students are familiar with.
- Must be executable (for dynamic analysis).

We chose two subject systems which meet all these requirements: GanttProject[1], a project planning tool, and ArgoUML[2], a tool for drawing UML diagrams. The Tetris game Jetris[3] was used for training. Table 2 shows the characteristics of these systems. LOC is the physical lines of code, while SLOC is the number of source lines of code as counted by `sloccount`[4] (i. e. excluding comments and empty lines). ArgoUML is four to five times the size of GanttProject, but it also seems to have a better comment ratio.

For each subject system, we identified one class that could be considered as being of central concern to this type of application. For GanttProject, we chose the `GanttTask`, which represents a task in a project. For ArgoUML, we chose the `ClassDiagramGraphModel`, which defines a binding between the underlying graph model and the graph editor. Finally, for Jetris, the `Figure` class was selected. These selections were purely based on investigating the class names and choosing classes that sounded promising. Table 2 shows the

[1] `http://www.ganttproject.biz/`
[2] `http://argouml.tigris.org/`
[3] `http://jetris.sourceforge.net`
[4] `http://www.dwheeler.com/sloccount/`

number of corresponding allocation points ("APs"), i. e. the number of locations where instances of these classes are created.

To extract DOPGs for the selected classes, use cases were executed for each of the systems. For GanttProject, a project file was loaded and a task's duration was changed. For ArgoUML, two classes were created using the (empty) class diagram editor. Then, they were connected by an association. Jetris was played for about a minute. The sizes of the resulting DOPGs are also given in Table 2.

In order to being able to maintain an unknown system, an engineer first has to understand it. If he is asked to extend or fix a certain feature, he first has to locate it. Feature location is a program understanding task that involves more than just telling a code location: The maintainer has to gain a basic understanding of certain parts of the system as well. Therefore, we consider this kind of task a good representative for program understanding and used them in the experiment.

The concrete tasks that had to be performed on GanttProject were the following:

1. GanttProject supports hierarchical tasks. Which code is responsible for updating the length of a parent task when a child task's duration is changed?
2. Which component is responsible for drawing the task (box) in the Gantt diagram?
3. Which class keeps the information about dependencies between tasks?

All three tasks are related to the GanttTask class: The first and third one are about the relation of tasks to each other, while the second one is about drawing this task. Tasks 1 and 2 have to do with the GUI: The change of a task duration is triggered by user action, and the task box is drawn for display to the user. The third question is about the internal data model.

For ArgoUML, the following questions were posed:

1. Which code has to be changed to make ArgoUML show an empty sequence diagram instead of an empty class diagram after startup?
2. How is the user's addition of an element to a diagram (e. g. adding a class to a class diagram) implemented?
3. Which class is responsible for recording and keeping a history of selections?

These three tasks have a relation to the ClassDiagramGraphModel: The model must be present when a class diagram is displayed or edited. Therefore, it is involved when an empty class diagram is created (task 1).

Introduction	10 min
Training (Talk)	15 min
Training tasks	25 min
Experimental task 1	25 min
Questionnaire	5 min
Experimental task 2	25 min
Questionnaire	5 min
Debriefing	10 min

Table 3. Session overview.

Adding an element to the diagram must also be reflected in the model (task 2). The third task is about selection management, which is more loosely related. All tasks have to do with the GUI.

3.5 Experimental Procedure

The participants were scheduled for one of a set of seven sessions. This was neccessary because only four identical workstations were available for the experiments. Each session participant was randomly assigned to one of the four different subgroups. This gave every participant the same 25% chance of being member of any of these groups. Each session lasted for two hours and had a structure as shown in Table 3. In each session, the participants were first given a short introduction, followed by a training of relevant methods. This covered using basic capabilities of Eclipse, such as code browsing, and demonstrated the *File* (textual) and *Java* (cross reference) search functions in more detail. The participants were also trained about DOPGs, i. e. their meaning, how to read and use them, and how to use the DOPG Eclipse plugin (see Figure 3). This training was conducted in approximately the same way for each participant group. This was achieved by using identical slides and an experimenter's handbook that described in detail what had to be done and said. However, questions of the participants did of course differ.

After that, each participant executed a set of six training tasks to get familiar with the environment and views. The tasks were similar to those that were given in the main part, but were much easier to accomplish due to the small size of the training system. The first three tasks were in parallel demonstrated by the experimenter. He showed how to solve each task in two different ways: By using Eclipse search functions, and by using the DOPG plugin.

For the main part, participants were told that they should use as much time as they needed for each task. They were not told in advance how many tasks would be given, and they were not allowed to go back to a previous task once they had finished it. Before the first

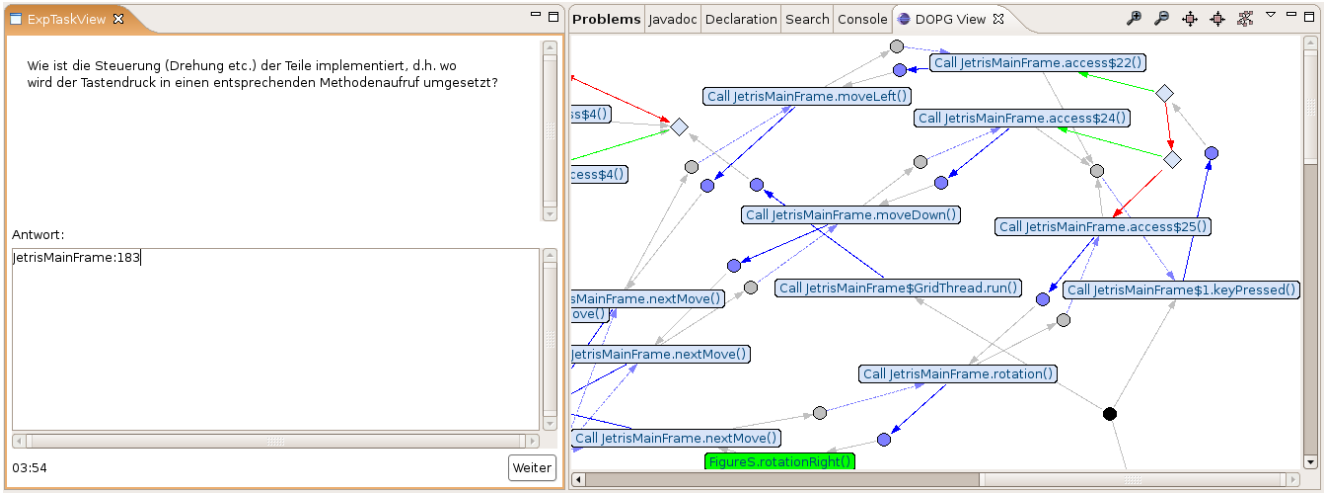

Figure 3. Eclipse plugins: The left view guides the participants through the experiments, giving information and tasks and accepting their answers. The right view is the DOPG viewer.

experimental task started, the experimenter demonstrated the two use cases of the two subject systems which had been used to generate the DOPGs. All participants were also told which was the chosen relevant class for each system.

During execution of the experimental tasks, measures of the dependent variables were recorded. These measures had to be taken in exactly the same way for each participant. We created and used an additional Eclipse plugin[5] that automatically led through the experimental tasks, presented the questions and recorded the subject's answers and reaction times. Figure 3 shows how the corresponding Eclipse view looks like. After each series of experimental tasks (one system), there was a post-mortem interview. The session closed with a debriefing.

3.6 Threats to Internal Validity

Given the described experimental setup and procedure, can we be sure that any observed effect is only caused by the variation of the independent variable, or are there any other extraneous variables that may have influenced the result? The following factors should be considered:

- **Individual participant differences:** Each subject has a different background regarding programming and software maintenance experience. These differences should be canceled out by the randomized assignment of participants to groups.
- **Instrumentation:** Measurement of the dependent variables may differ between participants. This threat was considered by automating the measurement process (see Section 3.5).
- **Session differences**: The training that the different groups received may have differed in details between sessions, although it was largely defined by the experimenter's handbook. There were also differences in the questions that were raised by different groups. However, since each session had one representative from each of the four subgroups, this effect should be canceled out.
- **Sequencing effects:** The effects caused by analyzing two different systems one after the other were equalized by counterbalancing the order of these systems.
- **Subjects' perception:** It could not be concealed that the subject of investigation were DOPGs, since this was the only unusual element in the experiment. The participants knew which technique was being investigated, and they could guess that this technique was invented by the experimenter.

3.7 Threats to External Validity

There are quite a number of factors that affect the generalizability of any results. Storey [8] discusses some of these threats in more detail.

- **Program representativeness:** It is unknown whether the programs chosen for this experiment are representative for real maintenance situations

[5] `http://sourceforge.net/projects/exclipse/`

or not. However, they should be large enough to be realistic.

- **Task representativeness:** The experimental tasks are not necessarily representative for real maintenance situations. Also, they were selected by the experimenter and may be of a kind that is particularly well suited for DOPG-based analysis. Generalization to different tasks is restricted.
- **Experience:** The participants were all computer science students, not professional programmers. Results will probably be different for experienced, professional Java programmers.
- **Familiarity with the system:** Being confronted with a completely unknown system is not the common situation in software maintenance. Usually, the system to be maintained is already well known to the maintainer. Results can not be generalized to such a sitation.
- **DOPG experience:** DOPGs were a new concept to all participants. They had to learn and understand this concept within a short period of time before using them. Subjects with more experience in using DOPGs will probably perform differently.
- **Experimenter effect:** The experimenter is the same person who invented DOPGs. This may have influenced any aspect of the experiment.
- **Choice of relevant classes:** The decision about the relevant class was not performed by the subjects. Would they have chosen the same relevant class, or some other class? The results are therefore not generalizable to the case that relevant classes are not preselected. Also, the effort for constructing DOPGs was not taken into consideration in the experiment. DOPGs were calculated outside the experiment.

4 Results and Discussion

This section presents and discusses the results of the experiment. The two subject systems and their associated tasks are not directly comparable and are therefore examined separately.

4.1 Response Time

First, we look at response times to questions, which were measured automatically during conduction of the experiment. The response time for each question was not limited, and several participants needed all available time only for the first task. Because of that, there is only complete data for the first task of each subject system. Response times for the other tasks are only available for a subset of participants. Also, the answer to the third question was in some cases already found while working on the second question, which further distorts response times for question 3. Therefore, a meaningful evaluation of response times is only possible for the first task of each system.

Results are summarized as barcharts in Figure 4. This figure is arranged as a matrix: The first column contains data from the GanttProject system, the second column from ArgoUML. The first row contains barcharts for response times to question one (*Q1*). This includes all answers, no matter if they were right or wrong. The second row displays response times for correct answers only (*Q1 [correct]*), and row three contains response times for wrong answers (*Q1 [wrong]*). Each box contains two barcharts: The upper one shows the control group's data, the lower one the experimental group's data. Each barchart contains the following information: The box indicates the 25% and 75% quantiles (*interquartile range*), the whiskers indicate the 10% and 90% quantiles, the thick line indicates the median, and each dot represents one data point.

On first sight, the charts show that the median response time to question 1 was always shorter when DOPGs were present. On the other hand, the interquartile ranges for GanttProject do not show a big difference. To find out whether the means are really statistically different from each other and not result of pure chance, we performed several tests on the different data sets:

- Student's t-test: This test assumes that the data is normally distributed, but it is quite robust against violations of this precondition. We therefore apply this test anyway as an additional indicator.
- Mann-Whitney U test: A non-parametric test which only considers the ranking of data, not absolute differences. It assesses whether two samples come from the same distribution.
- Bootstrapping: A resampling method [3]. It takes several thousand random samples from the original data, calculates the mean differences from those samples, and analyzes the resulting distribution. The relative location of the zero-crossing of the mean difference values results in the p-value.

The results of applying the different statistical tests are displayed below each box in Figure 4 as p-values. A p-value is the probability of obtaining the observed effect when the null hypothesis is true. All p-values result from two-sided tests. The numbers show that only differences between means for *Q1* times for ArgoUML

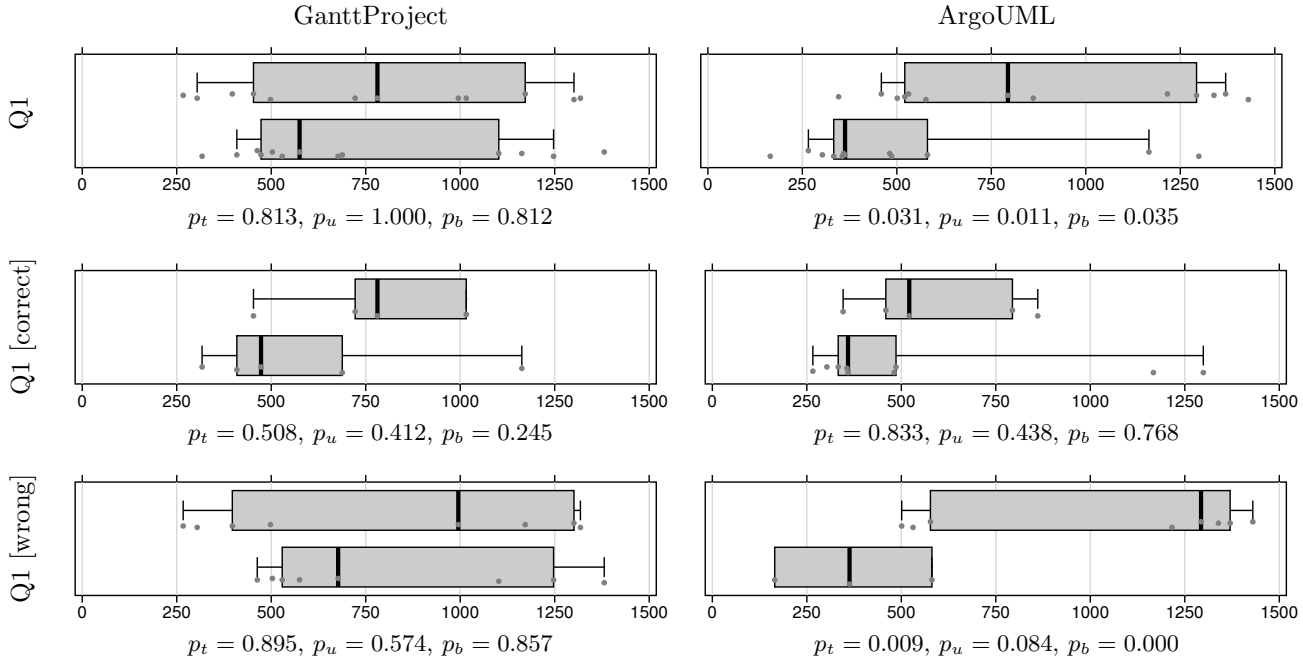

Figure 4. Statistical evaluation of dependent variable measures (response time for question 1). Within each box, the upper barchart represents the control group, and the lower barchart shows the experimental group's measures. The X axis represents the response time to question Q1 in seconds. Additionally, the p-values for different tests are given under each barchart: t-test (p_t), Mann-Whitney U test (p_u), and the bootstrap test (p_b).

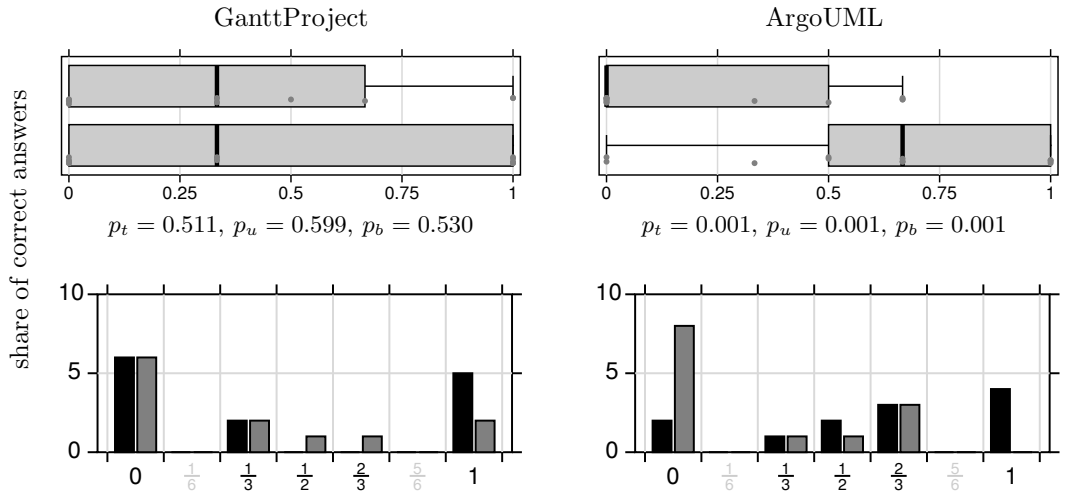

Figure 5. Statistical evaluation of dependent variable measures (correctness of answers). The upper barcharts represent the control group, the lower barcharts contain the experimental group's data. In the histograms, black bars represent the experimental group, and gray bars show the control group's counters. The X axis denotes the share of correct answers, the Y axis the number of occurences of each value.

are statistically significant (p=3.5%) for all three tests. Also, this is true for *Q1 [wrong]* times (ArgoUML), according to the t-test and the Bootstrap test (p=0.9%). All other differences between means have a high probability to occur under the null hypothesis.

These results suggest that answer times may be faster when DOPGs are available, but it depends on the system under investigation or the task. For the first GanttProject task, there was no significant indication that DOPGs helped, but for the first ArgoUML task, such indication could be detected. Therefore, hypothesis H1 is too general and must be rejected.

4.2 Correctness of Answers

The second dependent variable to be evaluated is the correctness of answers. Since participants finished different numbers of tasks, we look at the share of correct answers instead of absolute counts. Comparing absolute counts would additionally take into consideration response times, which we investigated separately in the previous section. The correctness of each answer was determined by first checking whether the answer (usually a source code location) complied to the sample solution. If it did not, the code was checked manually. If it was closely related to the sample solution, the answer was judged as correct, otherwise wrong.

Figure 5 shows the histograms for the share of correct answers for both subject systems. The bars for the experimental group are black, and those of the comparison group are gray. The charts reveal that there are no great differences in share of correct answers between the two groups for GanttProject. There were only a few more correct answers when DOPGs were available. This difference is not significant. For ArgoUML, the difference is obvious: Without DOPGs, about 60% of the participants did not give any correct answer – with DOPGs, two out of three answers were correct (mean). These mean differences are clearly statistically significant (p=0.1%).

The results are again twofold: For ArgoUML, we detected a strong tendency towards more correct answers when DOPGs were available, while no such difference at all could be detected for the GanttProject system and tasks. When only looking at the ArgoUML figures, we can reject the null hypothesis at the 0.1% significance level. For GanttProject, it definitely cannot be rejected.

It is interesting to note that no correlation between programmer's experience and response times or correctness of answers could be detected.

4.3 Questionnaire

After performing the tasks for one subject system, each participant filled out a questionnaire. This was requested to evaluate their subjective satisfaction, confidence and productivity. Among others, the following questions had to be answered:

- "Were you able to solve the tasks efficiently?" The average answer was "rather not", no matter if DOPGs were available or not.
- "Are your results correct?" The average answer was "rather yes". Again, the availability of DOPGs had no significant influence.

The participants were also asked to rate how helpful each of the available and relevant Eclipse features was for solving the tasks. Figure 6 shows the distribution of answers to this question. Answers may be missing when the feature has not been used. For GanttProject, ratings for the different features are similar, no matter if DOPGs were available or not. With DOPGs, they get the best rating of all features on average, but the *Java search* rating is quite close to it. Without DOPGs, the search functions have to be used more intensively; their rating varies. For ArgoUML, DOPGs were considered to be "very useful" when present. In their absence, the *Code browser* and *Java search* functions were used and found to be quite helpful.

The findings from the questionnaire confirm our earlier findings: DOPGs supported the ArgoUML tasks much better than the GanttProject tasks.

4.4 Discussion

The results are different from what we expected. When the experiment was designed, tasks for the two subject systems were considered to be of comparable difficulty. However, it turned out that participants had much more problems with GanttProject and its tasks than with ArgoUML. This is particularly astonishing when considering the fact that ArgoUML is more than four times the size of GanttProject. Some participants mentioned that ArgoUML's JavaDoc documentation was quite helpful, so this may have been one reason. Also, each of the six task was solved correctly by at least two students of each group, which means that the tasks were not infeasible.

There may be several reasons for the different DOPG performance with GanttProject and ArgoUML. Potential reasons include the following:

- For GanttProject, there were three DOPGs available, and two of them were quite large – much

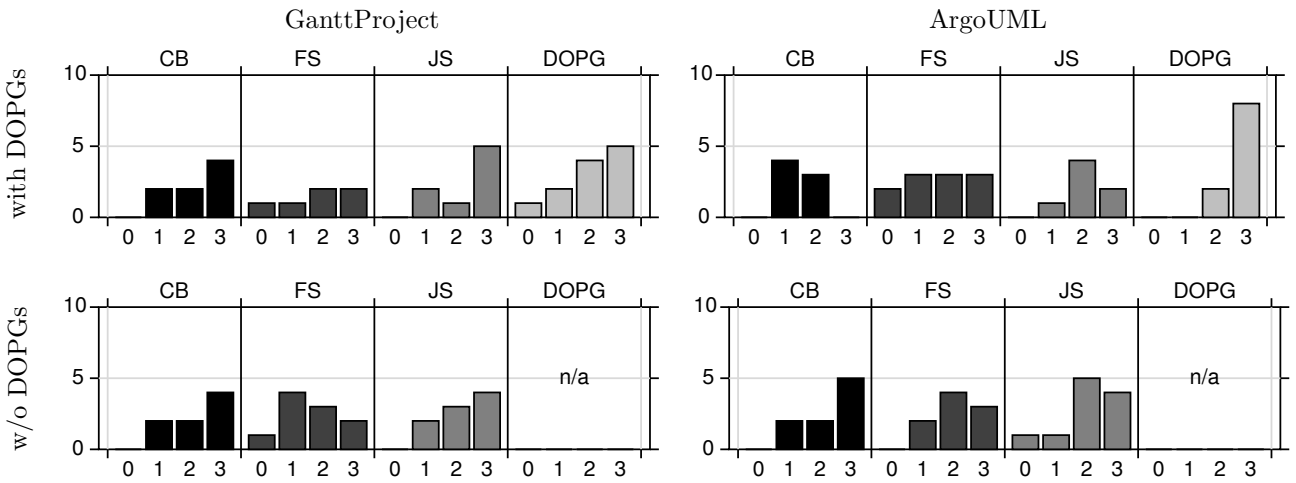

Figure 6. Rating of different tools. CB = Code browser, FS = File search, JS = Java search. The numbers on the X axis denote the rating: 0 = not helpful, 3 = very helpful. The Y axis shows the number of occurences of each rating.

larger than the ArgoUML graphs. For ArgoUML, there was only one DOPG of medium size. Maybe the effort for choosing among several DOPGs is higher than expected, or the graph size has an important impact.

- ArgoUML's first task was better solvable with the given DOPGs, or it was just easier.

We currently do not know what the real reasons are, but it is nevertheless an interesting finding that DOPGs improve program comprehension performance for certain tasks and systems – but for others, they do not. If we had chosen only one of those two systems for the experiment, the outcome would have depended only on this choice: For ArgoUML, the null hypothesis would have been rejected, but for GanttProject, we would not have been able to reject it.

5 Conclusion

We designed and performed a controlled experiment to find out whether DOPGs support program understanding. As a result, we could not reject the null hypothesis for our research question, which means that DOPGs do *not* support program comprehension *in general*. However, we found out that it depends on the subject system, tasks, and possibly the choice of DOPG objects whether they are helpful or not. For one of the investigated systems, DOPGs were clearly beneficial, while for the other system, this was as clearly not the case.

These findings illustrate the general problems one has in designing or appraising empirical studies of this kind: Even if an adequate subject system and representative tasks have supposedly been chosen, you cannot be sure what the results would have been with a different system or different tasks. This should be kept in mind when dealing with this kind of study.

Acknowledgements

Thanks to Lutz Prechelt for comments and helpful hints on the design of this experiment. Also, special thanks to the 27 student participants without whom the experiment could not have been conducted.

References

[1] L. B. Christensen. *Experimental Methodology*. Allyn & Bacon, Boston, USA, 8th edition, 2001.

[2] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[3] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall/CRC, 1998.

[4] M. D. Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Proc. of 15th ICPC*, pages 281–285, 2007.

[5] J. Quante. Online construction of dynamic object process graphs. In *Proc. of 11th CSMR*, pages 113–122, Mar. 2007.

[6] J. Quante and R. Koschke. Dynamic object process graphs. In *Proc. of 10th CSMR*, pages 81–90, 2006.

[7] J. Quante and R. Koschke. Dynamic object process graphs. *Journal of Systems and Software*, 2008. Accepted for publication. doi:10.1016/j.jss.2007.06.005.

[8] M.-A. D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proc. of 13th IWPC*, pages 181–191, 2005.