Submission (November 1, 2001) to 'Journal of Systems and Software'

A Controlled Experiment on Inheritance Depth as a Cost Factor for Code Maintenance

Lutz Prechelt*, Barbara Unger, Michael Philippsen, Walter Tichy
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany
Phone: +49/721/608-4068, Fax: +49/721/608-7343
prechelt,unger,phlipp,tichy@ira.uka.de
http://wwwipd.ira.uka.de/EIR/

Abstract

In two controlled experiments we compare the performance on code maintenance tasks for three equivalent programs with 0, 3, and 5 levels of inheritance. For the given tasks, which focus on understanding effort more than change effort, programs with less inheritance were faster to maintain. Daly et al. previously reported similar experiments on the same question with quite different results. They found that the 5-level program tended to be harder to maintain than the 0-level program, while the 3-level program was significantly *easier* to maintain than the 0-level program. We describe the design and setup of our experiment, the differences to the previous ones, and the results obtained. Ours and the previous experiments are different in several ways: We used a longer and more complex program, made an inheritance diagram available to the subjects, and added a second kind of maintenance task.

When taken together, the previous results plus ours suggest that there is no such thing as usefulness or harmfulness of a certain inheritance depth *as such*. Code maintenance effort is hardly correlated with inheritance depth, but rather depends on other factors (partly related to inheritance depth). Using statistical modeling, we identify the number of relevant methods to be one such factor. We use it to build an explanation model of average code maintenance effort that is much more powerful than a model relying on inheritance depth.

Keywords: controlled experiment, inheritance depth, maintenance, cost model

1 Inheritance and complexity

Inheritance is one of the key elements that makes object-oriented programming powerful[Meyer, 1997]. Many design problems can be solved elegantly using inheritance and polymorphism and the resulting designs are often simpler, clearer, and more flexible than could have been achieved otherwise [Gamma *et al.*, 1995]. But it is also obvious that the construct of inheritance may introduce additional complexity if used inappropriately. In this paper, we investigate how the use of inheritance influences the effort required for subsequent program understanding and maintenance.

*Postal address for proof copies: Lutz Prechelt; Kilianstr. 18; 70327 Stuttgart; Germany

Reviewing the literature, one finds many positive program properties attributed to the use of inheritance, in particular less redundancy through code reuse and simplified extensibility due to polymorphism. However, as far as program understanding is concerned the literature tends to be pessimistic about the consequences of inheritance. [Chidamber and Kemerer, 1994, p. 483] mentions "more complex design" as a disadvantage of deep inheritance hierarchies. Basili, Briand, and Melo [Basili *et al.*, 1996] empirically found from three-person student projects that classes were more likely to exhibit defects during testing when they are deeper in the inheritance tree. The textbook [Sommerville, 1992, p. 200] states that *"[. . .] class inheritance is not essential and may sometimes confuse a design, because an object class cannot be understood on its own without reference to any super-classes."* [Wilde and Huitt, 1992, p. 1040] describes the actual effect: *"Understanding of a single line may require tracing a chain of method invocations through several different object classes and up and down the object hierarchy to find where the work is really getting done."* A more general view of this problem is provided by the notion of *delocalized plans* [Soloway *et al.*, 1988]. A delocalized plan is a set of design decisions whose consequences are spread out over different locations within a program. [Wilde *et al.*, 1993] argues that inheritance is one of the factors why object-oriented programs tend to have many delocalized plans. Soloway et al. observed in their experiments [Soloway *et al.*, 1988] that delocalized plans account for much of the effort and many of the mistakes during program understanding. As a partial remedy for these problems, modern texts on object-oriented design recommend to prefer object composition over inheritance and to inherit only interfaces, not implementations; see for instance Gamma et al. [Gamma *et al.*, 1995, Chapter 1]. [Dvorak, 1994] found that the distinctions between the concepts represented by classes are confused much more often as one goes deeper into a class hierarchy. On level 3 of a class hierarchy there was less than 30 percent agreement among the subjects what the immediate superclass should be; ill-designed class hierarchies result.

In summary, one may expect that maintenance tasks that are dominated by the program understanding effort become more difficult when the inheritance hierarchy of the program is deeper.

1.1 Previous experiments: PRIOREXP

Daly, Brooks, Miller, Roper, and Wood conducted empirical research in this direction [Daly *et al.*, 1996]. Using interviews and a questionnaire they found that 55% of object-oriented practitioners among 273 respondents *"agreed that inheritance depth is a factor when attempting to understand object-oriented software"*. 31% of the respondents *"indicated that between four and six levels of inheritance depth is where the difficulties begin"* (page 111). Based on these findings, they devised controlled experiments for assessing the influence of inheritance depth on program maintainability. The experiment tasks consisted of adding a new class to a program that was designed either with inheritance (experiment group) or without inheritance, i.e., "flattened" by inserting inherited code textually (control group). No polymorphism was used in the program.

The results suggested that inheritance indeed tended to slow down code maintenance when the hierarchy was five levels deep, but was actually *beneficial* when the hierarchy was only three levels deep. In the following, we will refer to these experiments as PRIOREXP.

[Cartwright, 1998] replicated the 3-level comparison with two 5-person undergraduate student groups and found the flattened program to consume 40% less time for completing the maintenance task. This result indicates that even three levels of inheritance are not necessarily beneficial. We will use the data from this experiment along with those of PRIOREXP in our analysis in Section 5.

Harrison, Counsell, and Nithi [Harrison *et al.*, 1999] performed a related experiment, also with undergraduate students, based on similar programs but different tasks: They asked the subjects (which worked only on paper) to first determine some program outputs, then draw an inheritance diagram, then identify where changes where required for a certain program enhancement. The time was fixed at 45 minutes total and only the correctness of the solutions was compared. Both, the 3-level and the 5-level programs, turned out to provoke more mistakes than their corresponding 0-level versions. Since several aspects of the setup of this experiment are somewhat artificial, we will not refer to these results in the remainder of the present article. Nevertheless they also suggest that the results of Daly et al. will not always hold.

1.2 Article overview

This article presents a follow-up to the work of Daly et al.; it may be helpful to read their article as well [Daly *et al.*, 1996]. We changed several parameters in order to broaden the external validity. We designed a new experiment, performed it twice, obtained results that *contradict* those of PRIOREXP, and investigated an explanation.

In the following, we will first discuss how and why our experiment design was different from PRIOREXP and then report on our actual experiments (design, subjects, tasks, procedure) and findings. We will refer to our own experiments as NEWEXP. Section 5 will then resolve the apparent contradiction between the two sets of experiments by explaining that the hypotheses investigated were misleading from the start. The section presents a model that explains the results of both sets of experiments *without* relying on inheritance depth.

2 Comparison of PRIOREXP and NEWEXP

Overall we find PRIOREXP well designed and described. Initially we had but one point of criticism: We believed the subjects should be given additional documentation of the inheritance tree, preferably in graphical form. Later we also found out that the programs used were rather simple, in both size and structure. Compared to PRIOREXP, the design of our own experiment exhibits the following main differences:

Combined 3-level and 5-level test: PRIOREXP used two completely separate experiments. The first experiment compared 3-level inheritance to a "flat" (0-level) program. It was a two-part experiment: PRIOREXP-1a used program "university" and PRIOREXP-1b used program "library"; PRIOREXP-1b was later replicated as PRIOREXP-1r. The second experiment (PRIOREXP-2) compared 5-level inheritance to a flat (0-level) program, using an extended version of the program "university". In contrast, we used 0-, 3-, and 5-level versions of a single program "Boerse" within one experiment; refer to Sections 3.3 and 3.4 for details. Rationale: This design allows for direct comparison of the 3-level with the 5-level version in addition to comparing both to the flat version.

Class diagram: The subjects of PRIOREXP were equipped with the program source code only.[1] In contrast, we also handed out a printed class diagram (in OMT notation, including all method names) which allowed studying the inheritance relations at a glance. Rationale: In a modern programming environment, such information is available easily.

[1] According to [Daly, 1996], the subjects of PRIOREXP-1a and -1b (but not -2!) were also given a sorted table of instance variables in every class.

Table 1: Overview of our experiments (NEWEXP-*G*, NEWEXP-*U*) and the previous experiments (PRIOREXP-1a, PRIOREXP-1b/-1r, PRIOREXP-2, Cartwright replication C).

	—NEWEXP—		—PRIOREXP—			C
	G	*U*	1a	1b/1r	2	
no. of subjects	57	58	31	29	31	10
no. of groups	3	3	2	2	2	2
no. of datapoints	57	58	50	27	30	10
0-level program or version:						
classes	20	20	3	4	8	4
method bodies	158	160	26	35	96	35
lines	2470	2465	273	370	1007	370
program files	1	1	7	9	17	9
3-level program or version:						
classes	27	27	5	6		6
method bodies	100	96	21	27		27
lines	1344	1317	252	323		323
program files	1	1	11	13		13
5-level program or version:						
classes	28	28			11	
method bodies	80	79			56	
lines	1200	1187			694	
program files	1	1			23	
programming language	Java		C++			
task types	2x/1x add class, change		1x add class			
available materials	file, listing, inherit. diagram		files			
submission procedure	subject decides		supervisor checks			
observed variables	elapsed time, correctness		elapsed time			

Program complexity: Our programs were significantly longer than those used in PRIOREXP (see also Table 1 and refer to Sections 3.2 and 3.3); the classes had more kinds of relationships and partly unobvious functionality. Rationale: The PRIOREXP programs are unrealistically simple. All PRIOREXP classes belong to a single inheritance tree, there are no other relations between the classes except for inheritance, and the function and implementation of each class can almost be deduced from its name alone. In contrast, the class hierarchy of the NEWEXP program is much more complex; see Figure 1.

Task complexity: The tasks in PRIOREXP always consisted solely of adding a new class, whose structure was similar to that of each existing class. In contrast, our tasks require understanding of classes with different internal structure and imply less straightforward extensions. The size of the tasks was also larger; see Table 2 and Section 3.5 for details. Rationale: In practice, maintenance tasks are not often as simple as those of PRIOREXP.

Changes vs. extensions: In addition to one task requiring adding a new class (as in PRIOREXP), our subjects also performed a task involving changes to multiple existing classes; refer to

Table 2: Characterization, by program version, of the typical solution effort for the tasks (the task used in PRIOREXP-1b and -1r and the Cartwright replication are similar to that of PRIOREXP-1a). *Investigated methods:* number of methods that must be analyzed and understood for solving the problem. *Hierarchy changes:* number of times one must switch to a subclass or superclass during that method understanding process if it is performed by dynamic tracing (i.e. following the execution call sequence). *Solution methods:* number of methods that are copied from existing classes (verbatim or with changes) or modified in order to create the solution. No methods needed to be written from scratch.

		—NEWEXP— (*G&U*)			—PRIOREXP— 1a		2	
inheritance depth		0	3	5	0	3	0	5
Task 1:	solution methods	16	16	8				
"Add class" task/	investigated methods	13	16	17	3	5	4	7
Task 2a:	hierarchy changes	0	17	21	0	2	0	5
	solution methods	17	9	5	10	4/5	15	4
Task 2b:	investigated methods	2	2	2				
	hierarchy changes	0	0	0				
	solution methods	17	10	5				

Section 3.5. Rationale: This is a frequent type of maintenance in practice.

Submission procedure: When a subject finished, PRIOREXP solutions were checked by a supervisor and the subject was told to continue if the solution was not yet correct. In contrast, NEWEXP subjects decided alone when they considered their solution complete. Rationale: In practice, an all-knowing supervisor is usually not available.

Definition of inheritance depth: [Chidamber and Kemerer, 1994] suggests the metric DIT(p) (depth of inheritance tree) to mean the number of edges on a longest downward path from root to leaf in the inheritance tree(s) of a program p. However, DIT(p) may change during program maintenance. Therefore, we define *inheritance depth* (ID) to be the DIT of the program version before or after the maintenance task, whichever is larger, because program understanding in our experiments involves the deepest classes, whether new or existing. PRIOREXP's definition of ID counts classes, not edges, on the path (i.e., it is DIT+1), but considers only the program version before the maintenance. But since the PRIOREXP tasks always involve adding a class at the bottom of the hierarchy, the two definitions turn out equivalent. Multiple inheritance is not used.

Furthermore, our programs are written in Java (as opposed to C++), come from a different domain, and have a graphical user interface in addition to textual stream I/O.

3 Description of the experiments

While reading the following sections, please refer to Tables 1 and 2 and to Figure 1 for further characterization of the experiment design, the program used, and the task complexity.

3.1 Hypotheses

The starting point of our work is PRIOREXP: Our work checks the results of this previous research. Therefore, we started with hypotheses closely tied to those used in PRIOREXP, but removed many of the weaknesses in the experiment itself, as explained in Section 2.

For the range of inheritance depths from 0 to 5 and for tasks where most of the effort goes into understanding (rather than changing), we investigate the following hypotheses:

Hypothesis 1: Programs with more levels of inheritance require *more time for code maintenance*.

Hypothesis 2: Programs with more levels of inheritance result in *lower quality of code maintenance*.

It will turn out that these hypotheses are misleading (Section 4.5). Fortunately, a meaningful result can still be derived from the data (Section 5).

3.2 Subjects and environment

We performed our experiment twice, with small changes. The first time we performed it with 57 graduate Computer Science students as the final exam of an optional 6-week intensive Java programming course with initially 70 participants. Participation required achieving 75 percent of all points in the previous course exercises, so that only course subjects with sufficient practical capabilities participated in the experiment. We call this experiment *G* (for "graduate course").

We replicated the experiment with 58 undergraduate Computer Science students at the end of their first year (mandatory lecture and lab course "Informatik II" with about 160 participants). Participation was optional, but resulted in a small bonus on the grade of the written exam to be conducted later. The course had used Java for all programming exercises. We call this experiment *U* (for "undergraduate course").

On average, the self-reported previous programming experience of the *G* subjects ⟨*U* subjects⟩ was 8.1 years ⟨6.1 years⟩ using 4.0 ⟨3.9⟩ different languages with a median largest program of 2750 LOC ⟨2000 LOC⟩. Before the course, 91% ⟨62%⟩ of the subjects had previous experience with object-oriented programming, 47% ⟨45%⟩ with programming graphical user interfaces (GUIs). With respect to Java AWT GUI programming, the course concentrated on JDK1.0.2-style ⟨JDK1.1-style⟩ event handling. The *U* course covered only a small amount of GUI programming.

3.3 Program used

Our experiment program, called "Boerse", was an interactive application for displaying two different kinds of stock exchange data in various ways. The data is taken from two text files and displays can be selected to have textual table form or graphical chart form and to cover different time ranges into the past (day, week, month). When the user selects a display and enters a stock code number (if required for that display), a window pops up with the desired presentation.

Three functionally equivalent versions of this program were used in the experiments: The 5-level program represents the original program design. See Figure 1 for its inheritance tree. The 0-level "flattened" version was created by inserting inherited attributes and methods directly into the source text of each subclass and removing the inheritance relation — actual polymorphism

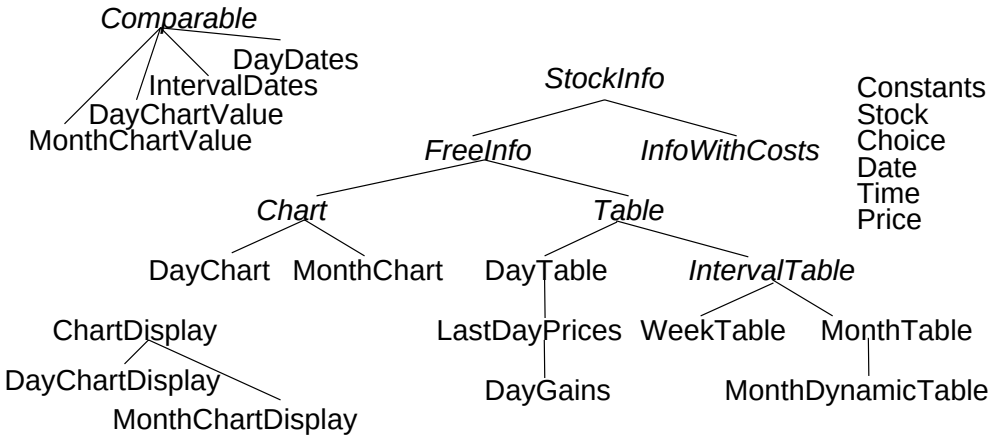

Figure 1: Classes and their inheritance relations for the 5-level version of the "Boerse" program.

is not used in any of the versions. The 3-level version was built according to the design rule "inherit only interfaces, not implementations." Therefore, in our 3-level version, subclasses were derived only from abstract classes, never from concrete classes. The rationale of this principle is maximizing flexibility for implementation change, see [Gamma *et al.*, 1995] for details. Thus, our three versions are not arbitrary variants of one program, but represent three different, but sensible design styles (although the 0-level program might be formulated more compactly). The programs were accompanied by two input files, whose format was relevant for solving the maintenance tasks.

For the *U* experiment these three versions were converted from their original JDK1.0.2 form (using `action()` methods for event handling) into a form using inner classes and JDK1.1-style event handling.

3.4 Experiment design

The independent variable in both experiments is the inheritance depth ID of the program. The variable has three levels, 0, 3, and 5, resulting in three experimental groups. The subjects did not know what the experimental variable was.

We used a matched-between-subjects design [Christensen, 1994], i.e., the subjects were ordered by expected performance and then randomly assigned to the three groups in such a manner that one out of any three subjects with similar expected performance would be assigned to each group. For the *G* experiment we used the scores from the previous course assignments for sorting the subjects, for the *U* experiment we used the self-reported size of the largest program they had ever written, because no better information was available. We do not claim that these criteria provide perfect matches, but a pretest found that they resulted in groups with reasonably balanced average subject ability; see Section 3.7.

The dependent variables were the time required for each assignment (measured in minutes) and the quality of the delivered solution (measured on a defined discrete grading scale as described in Section 3.6).

3.5 Tasks and assignments

Overall there were three different tasks, which we call Task 1, 2a, and 2b. The *G* subjects performed all three; the *U* subjects performed only 2a and 2b. There were two assignments that were measured separately. For the *G* experiment the first assignment consisted of Task 1, the second of Tasks 2a+2b. For the *U* experiment the first assignment consisted of Task 2a, the second of Task 2b. In both cases the subjects only had to modify the program source code (and test their changes at their own discretion). They were not asked to update the class diagram, perform regression testing, perform configuration or release management, or other tasks that are part of a complete software maintenance cycle.

Task 1 calls for converting the program from 2-digit to 4-digit years ("solving the Year-2000-Problem"). No further information is given. Changes are required at all places where records from either of the input files are split into their individual fields. At each point of change, the character offset of the date field and all subsequent fields have to be corrected. The changes do not require any deep understanding of or changes to the inheritance relations in the program. Most subjects found out during program understanding that the points of change are exactly all occurrences of the `substring()` method, which could then be found automatically. There are only half as many change points in the 5-level program than in either the 3-level or 0-level program.

Task 2a requires writing a new class and extending the menu handling in the main class. The task consists of adding a new type of display (arbitrary time interval price table display), whose functionality involves parts from both, existing chart-style and table-style display classes. Most of the table-style functionality can be inherited in the 3-level and 5-level programs and copied in the flat program. The code for selecting an arbitrary time interval can be adapted from a chart class in all versions. The new code must also overwrite behavior inherited from superclasses. This behavior is most heavily distributed in the 5-level program. This task requires a good understanding of and an extension to the inheritance relations in the program.

Task 2b asks for adding yet another type of display, featuring arbitrary time interval selection and table format as in Task 2a, but displaying differently computed data. It turns out that, in the 5-level program, the solution from 2a can be reused entirely if it was designed well. One only needs to change the superclass from which to inherit the data computation. The flat and 3-level programs can also profit from 2a, but less so. Like Task 2a, this task requires a good understanding of the inheritance relations in the program.

3.6 Procedure and measurements

Each of the experiments was performed in a single session in June 1997. The subjects implemented their solutions using JDK1.1 on IBM RS 6000 Unix workstations running AIX.

The experiment had four parts, for each of which the materials were handed out and collected individually: first a background questionnaire combined with a short pretest for evaluating the subjects' knowledge of Java inheritance rules, then assignment 1, assignment 2, and finally a postmortem questionnaire.

For each assignment and each subject we measured the time between handing out and collecting the experiment materials. As a backup and double-check, we also intercepted each compilation and each program run and registered their time. For each task, we graded the solution produced by the subjects according to the degree to which they fulfilled the requirements. The grading is

expressed as the number of points achieved. It was based on a fixed classification of error types with corresponding point penalties. The error types are all very simple and were determined by black-box testing. There were 5 such types for Task 1 and 9 for Tasks 2a and 2b, for instance "the data from the last day of requested time range is missing" or "the price table does not appear, only the request dialog is implemented".

3.7 Threats to internal validity

There are two major threats. First, there may be program differences that are not directly related to inheritance depth but still influence code maintenance effort. Such differences could have crept in during the conversion of the original 5-level program into the flat and 3-level versions. However, the conversion process was quite simple and we do not believe we have produced unintended differences.

Second, the group abilities may be unbalanced by chance. In our pretest we checked for this possibility using two Java comprehension assignments. We compared the proportions of correct versus wrong answers. Neither the Fisher exact p nor the χ^2-Test indicated significant differences for any of the relevant group pairs; the smallest of the 24 p-values obtained were 0.20, 0.24, and 0.33. Our pre-experiment questionnaire also asked for various information about the subjects' programming experience, such as the number of years, number of lines of code written, number of different programming languages used, etc. None of these properties differed significantly between the groups.

3.8 Threats to external validity

There are two main differences between the experimental and real software code maintenance situations that may limit the generalizability (external validity) of the experiments: First, in real situations there are subjects with more experience and, second, there are programs and maintenance tasks of different complexity, structure, and domain.

Experience: The most frequent concern with experiments using student subjects is that the results cannot be generalized to professionals. This is certainly an issue for the *U* experiment, where the subjects were rather inexperienced. The subjects of the *G* experiment, on the other hand, perform quite similar to professional software engineers, in particular since our Java course has attracted predominantly individuals from the top half of our studentship, with an average of more than 8 years of programming experience. Furthermore, it is typically not the most experienced developers who are doing the maintenance in practical software organizations. We do not believe that the experiment effect was influenced by our subjects' short language experience in Java, because the use of inheritance in the experiment programs was rather straightforward. This argument applies to PRIOREXP as well.

Structure and complexity of program and task: It is unclear how the effects observed in the experiments relate to those that may occur with other programs and other code maintenance tasks. In particular the program *structure* and the quality of available documentation may make a large difference; for instance the number of relationships (besides inheritance) between classes, the availability of different types of design documentation, and previous familiarity with the program and its domain (which both may also be considered a kind of documentation). The availability of program analysis tools may also be relevant. Furthermore, as our results below show, the type of

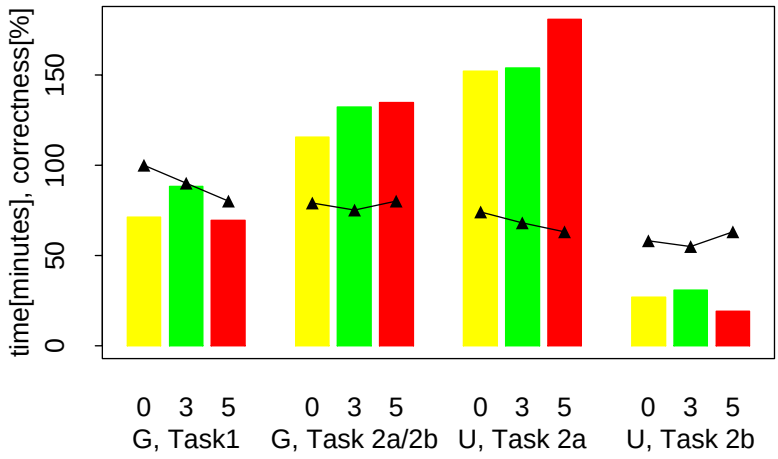

Figure 2: Average work times in minutes (as bars) and resulting average solution correctness in percent (as triangles and lines) for each task and each of the three program versions.

code maintenance task is an important factor. More research is needed before the relation between task type, inheritance depth, and code maintenance effort will be understood.

4 Results and discussion

The average times required by the groups for the various tasks is depicted in Figure 2. We report the results of one-sided, pair-wise statistical tests for identical means of these data which we performed using Bootstrap resampling percentiles [Efron and Tibshirani, 1993].[2] We report the p-values indicating the probability that the observed differences occured by chance alone; we call a difference significant iff $p < 0.1$.

4.1 Experiment *G*

Task 1 (Y2K-Problem): The subjects with the flat program version must perform a larger number of modifications than the group with the inheritance depth of 5 but they were almost as fast; the difference is not significant ($p = 0.200$). Also, the number of modifications is the same for the flat program compared to the 3-level program, yet the 3-level group was slower ($p = 0.075$) and hence was also even slower than the 5-level group ($p = 0.023$). These results are mostly, but not entirely, consistent with Hypothesis 1. The correctness of the solutions was perfect in the flat group (which obtained 100% of all points), very good in the 3-level group (90%), and good in the 5-level group (80%), which supports Hypothesis 2; the differences are not quite significant, though. The errors are mostly simple omissions, which is quite interesting, because it would suggest that the 5-level program, which has the smallest number of required changes, should come out best. It appears that some subjects did *not* use textual search for the `substring()` calls and then missed more of them in the more complex hierarchy. These results indicate that, for this task, the flat program is simplest to change. The relation between the 3-level and the 5-level version cannot reliably be

[2] We did not use the t-test because of severe non-normalities in our data. We did not use the Wilcoxon Rank Sum Test (Mann-Whitney U Test) because we want to compare means rather than medians.

concluded from the results because speed and correctness show opposite trends. Thus, the results provide modest support for both hypotheses.

Task 2a+2b (add two displays): In this task, the flat version was maintained significantly faster than the other two ($p = 0.038$ against the 3-level version and $p = 0.005$ against the 5-level version); these other two took about the same time ($p = 0.393$). Correctness is about equal in all three groups (79%, 75%, 80%, no significant differences). The relative frequency of different types of errors is similar in all three groups. Again, the largest amount of new source code needed to be produced for the flat version, but apparently it is easier to copy all of this code in one step and then modify it locally instead of tracing functionality up and down the inheritance hierarchy, independent of whether that is 3 levels or 5 levels deep.

According to these results, the flat program is again simplest to change and again the relation between the 3-level and the 5-level version cannot be decided. Thus, the results provide modest support for Hypothesis 1 and are inconclusive with respect to Hypothesis 2. However, separating the time for the subtasks 2a and 2b yields further insights as we will see below from the *U* experiment.

4.2 Experiment *U*

Task 2a (add interval table price display): For this class addition task there is little time difference between the flat and the 3-level version ($p = 0.456$). However, the 5-level version takes a lot longer to change ($p = 0.038$ against 0-level or $p = 0.055$ against 3-level). Apparently, the necessity for functionality tracing along the hierarchy as mentioned above is not yet harmful in the 3-level version, but becomes severe for the deeper 5-level hierarchy. The correctness of the solutions tends to decrease with increasing inheritance depth, but none of the differences are significant ($0.185 < p < 0.374$); this is also true for individual types of errors.

These results tend to support both hypotheses, but the differences are significant only for the time measure in the 5-level program.

Task 2b (add interval table gain/loss display): As mentioned above, the 5-level group can solve Task 2b by just duplicating the class written for the solution of Task 2a, changing its name and superclass, and augmenting the menu in the main program. The same was not possible for the other two versions. As a result, code maintenance of the 5-level program is significantly quicker for this task compared to the 3-level program ($p = 0.004$) or the flat program ($p = 0.006$); those two are about the same ($p = 0.263$). The correctness of the solutions or the types of errors made are not significantly different between any of the groups. There is a caveat for interpreting these results: some *U* subjects dropped out of the experiment during Task 2b. By comparison to the Task 2a results we find, not surprisingly, that the dropouts tended to be from the less capable half of the participants. Since the mortality was most pronounced in the 5-level group, the group abilities became unbalanced and thus the 5-level results obtained above are somewhat exaggerated.

4.3 Statistical robustness

All of the above conclusions about code maintenance time remain the same when possible outliers are removed by right-trimming, i.e., ignoring the largest 10% or even 20% of the time values in each sample. The same trends also still hold if we remove all solutions with substantial defects.

In both cases, the exact p-values merely change up or down a bit because the sample size and variance within the samples differ.

4.4 Subjective experience of subjects

The postmortem questionnaire asked for subjective judgements and had some interesting results. Judging whether the *use of inheritance* in the programs was adequate, a large majority of the flat version group (in particular the graduate students of *G*) found that inheritance was used *too little* or *much too little*. The other two groups by and large found this aspect OK. Still, however, more subjects in the flat program group than in the other groups believed they had a *correct solution*. With respect to the *clarity of program structure*, the *G* groups prefered the 5-level program over the other two, while the *U* groups prefered the other two over the 5-level program. Similarly, the *U* groups found the *task difficulty* lower for the flat program than for the other two. *G* had no clear differences. The answers to *"How well could you concentrate during the task?"* for the second task indicated lower concentration for the groups with deeper inheritance in both experiments.

4.5 Discussion

Both hypotheses stated in Section 3.1 are supported by these results. We must keep in mind here that Task 2b was unusual: For the 5-level program it required only cloning of an entire existing class, but no actual program changes whatsoever. This leads to shorter time and fewer errors compared to the 3-level program. With this exception, all group differences support the expectation that deeper inheritance hierarchies make program understanding and code maintenance both slower and more error-prone. Not all of the differences are statistically significant, but if we discount Task 2b as mentioned above, they all point into the same direction consistently.

These results are in sharp contrast to the results of PRIOREXP, which claimed that 3 levels of inheritance was better than both 0 and 5 levels. Our results are in line, however, with those of [Cartwright, 1998], who compared only 0 and 3 levels and found 0 to be faster. One may ask: "Which of these experiments are right?", but we think this is not a useful question. The ideal inheritance depth is likely to depend strongly on the purpose of the program and on the change task to be performed. The hypotheses stated in Section 3.1 should thus be considered misleading. Therefore, we will now search for better predictors of code maintenance effort than inheritance depth.

5 Building explanation models of code maintenance effort

As we saw above, if a practitioner asks "What is driving code maintenance effort? How can I understand, predict, and reduce it?", then inheritance depth is not useful. The purpose of the present section is to see whether the available data from NEWEXP (6 groups), PRIOREXP (8 groups), and the Cartwright replication (2 groups) allows better answers. Although we have data from 16 groups of subjects, the total data set is still rather small. Therefore, the subsequent analysis is only exploratory and the answers only tentative.

We will now search for prediction models of the form

$$t = f_{c_1 \ldots c_k}(P, T, M)$$

Table 3: Various prediction models for average code maintenance times. t is the time in minutes for the "add class" task, h is the number of hierarchy changes during understanding, m is number of methods to be understood, $exp = H$ and $exp = L$ are the groups with high or low levels of experience, d is the inheritance depth (ID). k is the number of coefficients (each model has 16 minus k degrees of freedom), r^2 is the fraction of variance explained by the model, p_{max} is the p-value of the least-significant coefficient in the model. Except for all coefficients d, all coefficients are significant in all models. $14[exp = L]$ denotes a separate constant offset of 14 for the low-experience groups only and $8.9m_{exp=L}$ denotes a coefficient for m that is 8.9 for the low-experience groups and 0 otherwise (an interaction term).

	prediction model	k	r^2	p_{max}
1	$t = f(m) = 6.9m + 28$	2	0.84	0.007
2	$t = f(h) = 4.5h + 61$	2	0.55	0.001
3	$t = f(d) = 7.8d + 70$	2	0.10	0.23
4	$t = f(d, m) = 7.4m - 3.3d + 30$	3	0.85	0.33
5	$t = f(m, exp) = 7.3m + 14[exp = L] + 22$	3	0.91	0.006
6	$t = f(d, m, exp) = 7.8m + 13[exp = L] - 2.9d + 25$	4	0.92	0.21
7	$t = f(m, exp) = 8.9m_{exp=L} + 5.9m_{exp=H} + 24$	3	0.94	0.001
8	$t = f(d, m, exp) = 9.3m_{exp=L} + 6.3m_{exp=H} - 3.3d + 27$	4	0.96	0.07

that is, functions that compute a prediction of the code maintenance effort t (the time for completing the task) based on input variables that are properties of the program P (such as the inheritance depth), properties of the maintenance task T (such as the number of classes that need to be changed), or properties of the maintainers M (such as their experience level). We will consider only the "add class" tasks. Each such function depends on one or several fixed parameters (coefficients) c_1 through c_k, which we will compute from the experiment data by least squared error optimization. Due to the large amount of individual variation, it is hopeless to predict the time for the individual subject, hence we will predict the group average time instead. Our models will focus on program understanding, because the given tasks did not require much actual change effort. We considered mostly, but not exclusively, linear models. Note that we are now considering the absolute effort, not only the relative effort for different versions of the same program, we are attempting a quantitative prediction instead of only a qualitative one, and we are using data from all experiments, not just our own. It turns out that the number m of methods investigated in the default solution strategy is the most powerful predictor of code maintenance effort. m is connected to inheritance depth, but the inheritance depth itself is hardly relevant.

5.1 Input variables: Properties of programs, tasks, and subjects

For deriving the models, we use three different sets of inputs which one may expect to have some influence on effort.

Program properties: The first set of possible inputs to our models are the quantitative program properties as described for instance in Table 1: inheritance depth (ID), number of classes, method bodies, lines, and files.

Task properties: Obviously, even if ID influences program understanding effort, it is not the ID itself that creates the effort differences, but rather ID's consequences. For instance, programs

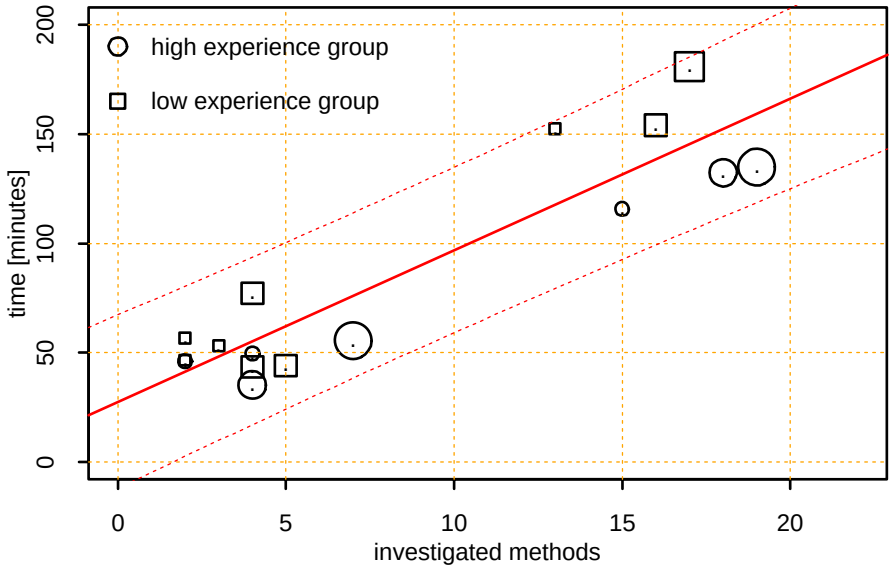

Figure 3: Visualization of Model 1 from Table 3. Each symbol represents one experiment group. Symbol size indicates inheritance depth. The lines show the prediction and a 90 percent confidence band of average work time as a function of the number of investigated methods, i.e. methods analyzed during understanding if a straightforward approach is used.

with deeper inheritance may have properties such as more heavily delocalized plans and a larger number of classes and methods to be understood, they may require more hops from one class into another during understanding, etc.

Such properties are currently rarely used to characterize programs, because they are hard to quantify. They depend not only on the program but also on the task and on the behavior of the programmer attempting the program understanding. But if the task is known and if one is willing to assume a certain reasonable "default strategy" is used to gain understanding, we can compute values for these properties, as shown in Table 2. We consider as input variables the number of investigated methods and number of required hierarchy changes.

The effort for the actual program change can be characterized by task properties such as the number of new or changed or cloned classes, methods, statements, declarations etc. Again, the assumption of a reasonable "default solution" is necessary to compute such values; in our case the number of new methods in the solution as shown in Table 2.

It is clear that these values do not always reflect reality, because some programmers will not use the default strategy for understanding or will not implement the default solution. Still, such assumptions might lead to useful predictions of *average* code maintenance effort. Our investigations find that indeed they do.

Subject properties: Third, we use a coarse binary classification of the expected skill level of each experiment group, because it makes little sense to ignore skill differences as large as those between our *U* and *G* groups. Note that this variable does not refer to individual subjects but rather to groups as a whole. The subjects of *G* are roughly comparable in experience to the subjects of PRIOREXP-1r and PRIOREXP-2. This level of experience is referred to as "high" ($exp = H$). The subjects of *U* are roughly comparable in experience to the subjects of PRIOREXP-1a, -1b, and the Cartwright replication. This level of experience is referred to as "low" ($exp = L$).

5.2 Models with two coefficients

The most reasonable models among all those we investigated are shown in Table 3. We will now discuss them, starting with Model 1. This standard linear regression model based on m is found to explain 84% of the variance among the average group work times. The model suggests a work time of 6.9 minutes per method that is investigated plus a constant effort of 28 minutes (for understanding the task itself, writing out the solution and so on). The quality of this model is rather surprising, given that we are talking of a rather heterogeneous data set that comprises four different programs from three different domains, five different groups of subjects with very different education and capabilities, and two different experimental conditions. The model and its underlying data are visualized in Figure 3.

In contrast, the model based on the number of hierarchy changes required during program understanding is less useful (Model 2 in Table 3). It explains only 55% of the variance and the constant effort of 61 minutes is unrealistically high. We conclude that the number of methods to be understood is a far better predictor of effort than the number of hierarchy changes required. When combining both (not shown in the table), the contribution of hierarchy changes becomes totally insignificant ($p = 0.98$).

Inheritance depth as the sole predictor is even worse, as is seen in Model 3. It explains only 10% of the variance and is not meaningful at all ($p = 0.23$).

5.3 Models with three or four coefficients

Adding inheritance depth as a second predictor variable to Model 1 does not significantly improve the prediction (Model 4, $p = 0.33$). In contrast, adding a constant time for lower experience levels does improve the prediction (Model 5, explaining 91% of the variance), but also adding inheritance depth leads to no significant improvement (Model 6, $p = 0.21$) again.

The most useful model results from providing different coefficients for m depending on the experience level of the group (high or low, Model 7). We thus obtain a model that has only three coefficients, yet explains 94% of the variance. It suggests a constant effort of 24 minutes plus 8.9 minutes per method for the less experienced subjects or 5.9 minutes for the more experienced subjects. The prediction quality of this model is shown in Figure 4.

Even based on this model, the predictive power of the inheritance depth as an additional predictor variable is weak (Model 8). Its contribution is small (at most 16 minutes) and only weakly significant ($p = 0.073$).

5.4 Discussion

We do not claim that the number of relevant methods is a universally good predictor of code maintenance effort. Quite on the contrary: we believe that it is valid only for tasks that are dominated by the program understanding effort. However, its high suitability for the given data makes clear that a search for models of code maintenance effort should consider (in addition to other factors) program properties *related to* inheritance depth rather than inheritance depth itself.

Note that the number of methods m that need to be understood for solving the task is hardly practical for predicting the effort in real maintenance situations, because m can usually not be

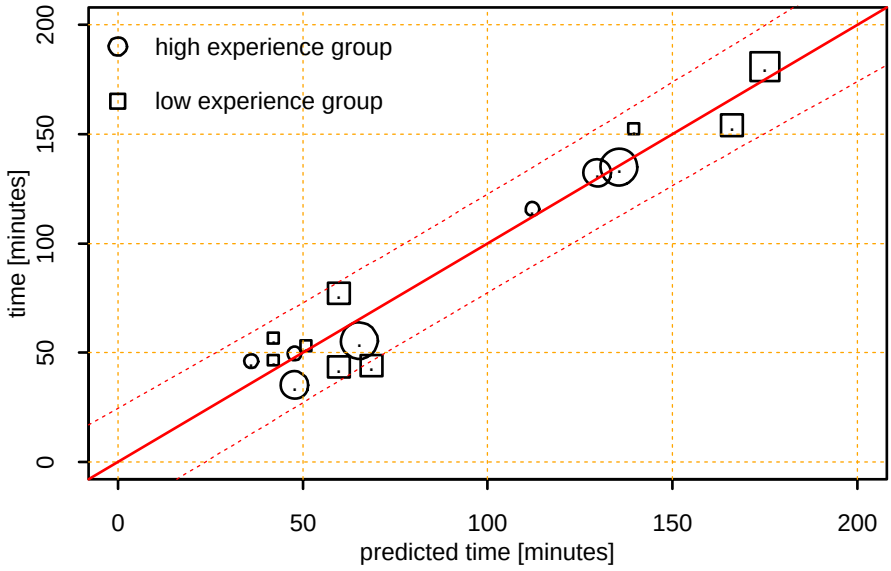

Figure 4: Prediction quality of Model 7 from Table 3. Since two predictor variables m and exp are involved, we cannot plot the model directly in two dimensions. Hence, the horizontal axis refers to the predicted value $8.9m_{exp=L} + 5.9m_{exp=H} + 24$ instead.

computed in advance. The models should therefore be called explanation models rather than prediction models.

Nevertheless, information about typical maintenance tasks may often be available during strategic decisions, so that m can be a useful indicator for comparing different design alternatives or for determining when reengineering may pay off for software that has to be maintained for a long time.

6 Conclusion

In our experiment setting, smaller inheritance depth resulted in smaller code maintenance effort when adding a class, contradicting the previous results of Daly et al. but corroborating those of the replication performed by Cartwright. However, this result is presumably task-specific because our tasks (as well as the previous ones) are dominated by program understanding effort. Probably the real advantages of inheritance become visible only for tasks involving complicated functionality modifications where less inheritance means a higher degree of code duplication and hence higher effort for changes and consistency checking.

For the tasks given in our and the previous experiments, we investigated several potential cost drivers of the average code maintenance effort by exploratory statistical modeling. We found the number of methods that need to be understood to be the most useful factor for predicting effort, far more general and reliable than inheritance depth. This result suggests that the assumption underlying, Daly et al.'s, Cartwright's, and our own experiments, is wrong. Inheritance depth is *not in itself* an important factor for code maintenance effort. Rather, we should investigate related program properties that are more directly connected to the actual maintenance procedure.

We speculate that the actual important factors may interact with inheritance depth, but are much more complex, such as (1) the match between the program design and the particular maintenance

task and (2) interactions with previous tasks (via knowledge gained therein). Although neither our nor the previous experiments were designed to identify or measure such factors, we could identify one of their components, the number of methods to be understood. Other plausible components, such as the degree of distribution of relevant items over the source code need to be investigated in future experiments that are designed specifically for that purpose.

Further details

This article cannot provide a complete description of the programs and tasks used. However, detailed information is available in a technical report[Unger and Prechelt, 1998] that includes the complete experiment materials, such as the task descriptions and source program listings, and also describes the grading scales, error types, and other details of the results. The experiment materials and raw result data are also available online at http://wwwipd.ira.uka.de/EIR/.

Acknowledgements

We thank Gerd Hillebrand for providing the Informatik II subjects and Arno Wagner for helping with mastering the technical infrastructure. We also thank all our experimental subjects for their patience.

References

[Basili *et al.*, 1996] Victor R. Basili, Lionel Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22(10):751–761, October 1996.

[Cartwright, 1998] Michelle Cartwright. An empirical view of inheritance. *Information & Software Technology*, 40(4):795–799, 1998.
http://dec.bournemouth.ac.uk/ESERG.

[Chidamber and Kemerer, 1994] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):476–493, June 1994.

[Christensen, 1994] Larry B. Christensen. *Experimental Methodology*. Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

[Daly *et al.*, 1996] John Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Wood. Evaluating inheritance depth on the maintainability of object-oriented software. *Empirical Software Engineering*, 1(2):109–132, 1996.

[Daly, 1996] John Daly. *Replication and a Multi-Method Approach to Empirical Software Engineering Research*. PhD thesis, Dept. of Computer Science, University of Strathclyde, Glasgow, Scotland, 1996.

[Dvorak, 1994] Joseph Dvorak. Conceptual entropy and its effect on class hierarchies. *IEEE Computer*, 27(6):59–63, June 1994.

[Efron and Tibshirani, 1993] Bradley Efron and Robert Tibshirani. *An introduction to the Bootstrap*. Monographs on statistics and applied probability 57. Chapman and Hall, New York, London, 1993.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Harrison *et al.*, 1999] Rachel Harrison, Steve Counsell, and Reuben Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. In *Proc. 3rd Intl. Conf. on Empirical Assessment and Evaluation in Software Engineering*, University of Keele, England, 1999.

[Meyer, 1997] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1997.

[Soloway *et al.*, 1988] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[Sommerville, 1992] Ian Sommerville. *Software Engineering*. Addison-Wesley, Wokingham, England, 4th edition, 1992.

[Unger and Prechelt, 1998] Barbara Unger and Lutz Prechelt. The impact of inheritance depth on maintenance tasks: Detailed description and evaluation of two experiment replications. Technical Report 18/1998, Fakultät für Informatik, Universität Karlsruhe, Germany, July 1998.

[Wilde and Huitt, 1992] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Trans. on Software Engineering*, 18(12):1038–1044, December 1992.

[Wilde *et al.*, 1993] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, January 1993.