A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance

Wojciech James Dzidek, *Student Member*, *IEEE*, Erik Arisholm, *Member*, *IEEE*, and Lionel C. Briand, *Senior Member*, *IEEE*

Abstract—The Unified Modeling Language (UML) is the de facto standard for object-oriented software analysis and design modeling. However, few empirical studies exist which investigate the costs and evaluate the benefits of using UML in realistic contexts. Such studies are needed so that the software industry can make informed decisions regarding the extent to which they should adopt UML in their development practices. This is the first controlled experiment that investigates the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of a real nontrivial system, using professional developers as subjects, working with a state-of-the-art UML tool during an extended period of time. The subjects in the control group had no UML documentation. In this experiment, the subjects in the UML group had, on average, a practically and statistically significant 54 percent increase in the functional correctness of changes $(\mathrm{p} = 0.03)$ and an insignificant 7 percent overall improvement in design quality $(\mathrm{p} = 0.22)$, though a much larger improvement was observed on the first change task (56 percent), at the expense of an insignificant 14 percent increase in development time caused by the overhead of updating the UML documentation $(\mathrm{p} = 0.35)$.

Index Terms—Empirical software engineering, UML, modeling, object-oriented programming, software maintainability, quasiexperiment.

1 Introduction

The Unified Modeling Language (UML) allows for the visual representation of a system's specification at various levels of design and is used to construct and document the artifacts of an object-oriented software system. This, in turn, aids in the communicating and understanding of various system properties. Advocates of UML often cite the following advantages: ability to handle the growing complexity of software development by working at higher levels of abstraction, traceability from requirements to low-level design, and more efficient communication. In fact, engineering designs are traditionally conveyed via two complementary notations, textual and visual, through domain-specific standardized notations. Since software designs must be expressed and communicated to many stakeholders, a visual language that is complementary to the code should be able to provide advantages as it does in other disciplines. Furthermore, the need to efficiently communicate design intent during development, maintenance, and evolution is an area in which improvements can have significant benefits.

Despite a growing popularity, there is little reported evaluation of the use of UML-based development [1], and many still perceive the development and maintenance of analysis and design models in UML to be ineffective [2]. Such practices are therefore viewed as difficult to apply in development projects where resources and time are tight. It is then important, if not crucial, to investigate whether the use of UML can make a practically significant difference that would justify the costs. This is particularly true in the context of software maintenance, which consumes most of software development resources as discussed in [3] and [4]: "Maintenance typically consumes 40 percent to 80 percent of software costs. Therefore, it is probably the most important life cycle phase of software" and "60 percent of software's dollar is spent on maintenance, and 60 percent of that maintenance is enhancement. Enhancing old software is, therefore, a big deal." Furthermore, the maintenance tasks are not usually performed by the original developers; thus, a lot of effort must be spent on understanding its functionality, architecture, and a myriad of design details of the large and complex existing system in order to change it correctly.

Having established the need for an empirical study where developers use UML during the maintenance phase leads us to the next issue: the manner in which UML should be used (i.e., the amount of detail that should be present in the diagrams and the necessary tool support). At one extreme, some argue for using UML at a very informal level, where diagrams are sketched on a white board in order to help communicate ideas and alternatives with colleagues; their emphasis is on selective communication rather than complete specification. These diagrams are either soon discarded or quickly become inaccurate (since they do not get modified along with the code). At the other extreme, proponents of the Model-Driven Architecture (MDA) believe that, thanks to MDA, future programmers will mostly deal with models instead of focusing on code (UML becomes the programming language) [5]. Since all changes occur via the models, these are always up to date, though the opposition claims that this is highly inefficient. This approach depends on tools that we do not yet possess.

This paper attempts to evaluate the costs and benefits of using UML at a given degree of formality, which represents

• *The authors are with the Simula Research Laboratory, Department of Software Engineering, PO Box 134, N-1325 Lysaker, Norway, and the Department of Informatics, University of Oslo, PO Box 1080, Blindern, N-0316 Oslo, Norway. E-mail: {jamesdz, erika, briand}@simula.no.*

Manuscript received 5 June 2007; revised 8 Jan. 2008; accepted 7 Feb. 2008; published online 25 Feb. 2008.
Recommended for acceptance by H. Muller.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-06-0180.
Digital Object Identifier no. 10.1109/TSE.2008.15.

a realistic compromise between the two extreme positions presented above. This is done through a controlled experiment performed at the Simula Research Laboratory, Oslo. The primary strength of this experiment is the level of realism: It involved 20 professional developers (intermediate to senior-level consultants) individually performing the same five maintenance tasks to the same real nontrivial system, where 10 of the developers worked with a UML-supported development environment and UML documentation, whereas the other 10 developers used the same tools but had no UML documentation to be read or updated. The developers took 1-2 weeks to implement the change tasks. An additional objective was to identify reasons for varying results and therefore identify plausible and necessary conditions for UML to be effective. Our decision to answer the above research question with a controlled experiment stems from the many confounding factors that could blur the results in an industrial context. Furthermore, this experiment builds on expertise acquired in the first two experiments of this type [6], with this one being the first experiment to ask these questions in such a realistic setting. A detailed structured comparison of this and the previous work is presented in Section 5.2.

The remainder of this paper is structured as follows: Section 2 reports on the planning of the controlled experiment and the results are presented in Section 3. Section 4 analyzes the threats to validity. Related work is discussed in context in Section 5. Improvements to the UML tool used in the experiment are suggested in Section 6 and conclusions are drawn in Section 7.

2 Experiment Planning

This section reports on how the experiment was designed and conducted.

2.1 Experiment Definition

We wanted to analyze the effects of UML for the purpose of evaluating the costs and benefits of modeling artifacts with respect to effort, functional program correctness, and the design quality of the solution. An important aspect was to decide what the baseline should be, against which the use of UML would be compared. There are, of course, an infinite number of possibilities here, given the wide variation in software development practices. However, in our experience, the most common situation can be defined as follows: 1) The source code is the main artifact used to understand a system, 2) the source code is commented to define the meaning of the most complex methods and variables, and 3) there exists a high-level textual description of the system objectives and functionality. This situation is therefore what we will use as a basis of comparison in order to determine whether the abstract representations captured by UML help developers perform their change tasks.

The experiment attempted to answer the following research questions:

1. Does the provision of UML documentation reduce the effort required in correctly implementing the change tasks?
2. Does the provision of UML documentation increase the functional correctness of the delivered solution? Since a fault found after the release of the software is significantly more expensive to fix than one found during development [7], [8], special attention must be paid to whether UML increases the probability of the change being functionally correct.
3. Does the provision of UML documentation improve the design quality of the delivered solution? Alternatively, does the use of UML decrease the decay of a system's design caused by maintenance tasks?
4. What are the shortcomings of the used state-of-the-art UML tool and how can it be improved?

2.2 Context Selection

The context selection is representative of situations where professional Java programmers perform realistic maintenance tasks for the duration of 1-2 weeks on a real nontrivial system. Furthermore, the system is initially unknown to the programmer and we are thus in the common situation where maintainers are not the initial developers of the system.

More specifically, 20 professional developers were recruited from Norwegian consulting companies and were paid the negotiated hourly wages. The advantage of using experienced professional developers is to avoid one of the main criticisms of most controlled experiments in software engineering: As opposed to student experiments, our results are representative of developers with industrial skills. Furthermore, unlike industrial case studies, which typically also use professional developers, this experiment controls for many extraneous factors that can impact our ability to analyze the effect of UML on software maintenance.

Since the 10 subjects working with the UML had various degrees of experience and knowledge of UML, they were all given a one-day refresher course that dealt with UML elements with which familiarity was necessary for the experiment. This time was also used to introduce the subjects to the selected UML-supported IDE: Borland Together for Eclipse (BTE) [9]. BTE was selected as the modeling tool due to 1) the advanced synchronization feature between the model and the code and 2) the tight integration with the Eclipse IDE.

2.3 Hypothesis Formulation

Our experiment has one independent variable (the use of UML documentation in a UML-supported IDE) and two treatments (*UML*, *no-UML*). It has six dependent variables, on which treatments are compared:

- T: Time to perform the change *excluding* diagram modifications.
- T': Time to perform the change *including* diagram modifications.
- Functional correctness in terms of the following:
  1. C: Number of submissions of a solution with a fault.
  2. C': Number of submissions of a solution with a fault, where the fault broke the existing functionality, a subset of C.
  3. C'': Number of submissions of a solution with a fault, where the fault stemmed from not taking into account all existing behavior C'', a subset of C. An example of a fault of type C'' would be a scenario where the developer must update two packages to correctly handle some new functionality, but, due to the lack of understanding of the system, only updates one of those packages.

TABLE 1
Tested Hypotheses

Dependent variable	Null hypothesis	Alternative hypothesis
Time excluding diagram modifications	H_0: T(UML) ≥ T(no-UML)	H_a: T(UML) < T(no-UML)
Time including diagram modifications	H_0: T'(UML) = T'(no-UML)	H_a: T'(UML) ≠ T'(no-UML)
Correctness – Num. of submissions of a solution with a fault	H_0: C(UML) ≥ C(no-UML)	H_a: C(UML) < C(no-UML)
Correctness – Introduced a fault breaking existing functionality	H_0: C'(UML) ≥ C'(no-UML)	H_a: C'(UML) < C'(no-UML)
Correctness – Introduced a fault stemming from not taking into account all existing behavior	H_0: C''(UML) ≥ C''(no-UML)	H_a: C''(UML) < C''(no-UML)
Design Quality	H_0: Q(UML) ≤ Q(no-UML)	H_a: Q(UML) > Q(no-UML)

- Q: Design quality in terms of following proper OO design principles [10]. This was calculated by first breaking each task into *subtasks* and rating each as either *acceptable* or *unacceptable* according to the predefined criteria elaborated upon in Section 2.7. Q counts the number of acceptable subtask solutions.

Following the example and logic in [6], when comparing the time spent on tasks across the UML and no-UML groups, one should, of course, account for the overhead involved in modifying UML diagrams. Bearing this in mind, T' is a priori a better measure than T when assessing the economic impact of using UML. However, we believe that it is still relevant to assess T as such results will provide evidence regarding whether UML, as a minimum requirement, facilitates the understanding and change of code. Furthermore, the time spent on modifying the models probably depends strongly on the modeling tool used and the subject's training in that particular tool. This is highly context-dependent and we therefore wanted to distinguish the time that developers spent understanding and modifying the code (with the help of UML diagrams) from the time spent on modifying the UML diagrams. The two measures of time are expected to provide interesting complementary insights.

Two subsets of faults with respect to functional correctness C are examined independently: C' and C''. C' only measures the number of faults that led to the existing functionality being broken (as opposed to the functionality that was added as part of the task), while C'' only measures the number of faults that stemmed from the developer not adapting the existing functionality to work with the newly added functionality. While both C' and C'' show a lack of understanding of the system being modified, faults of type C' may be prevented with the assistance of a complete regression test suite.

The hypotheses for testing the effect of UML documentation on our dependent variables are given in Table 1. The alternative hypotheses Ha state that using UML documents improves five out of the six dependent variables: less time to complete the tasks when *excluding* diagram modifications T, improved correctness in terms of C, improved correctness in terms of C', improved correctness in terms of C'', and improved design quality Q. Thus, Table 1 defines five of the hypotheses as one-tailed because we expected that using UML documentation would help people understand the system design better and hence provide better solutions faster. However, it is difficult to have clear expectations regarding the effect of using UML documentation on time when *including* the time spent on diagram modifications T' because the time taken to modify the diagrams might be greater than the expected time gains. Thus, the hypothesis on time including diagram modifications (T' in Table 1) is two-tailed.

The hypotheses will be tested on the results of each task that the subjects perform (at the task level) and on the aggregated results of all the five tasks (across all of the tasks at the subject level). In the case of design quality, the hypothesis will also be tested on each subtask (as defined in Section 2.7) of every task. Splitting tasks into subtasks allows for a comparison of the quality of solutions across developers while, at the same time, allowing for a large degree of freedom in the way that they implement their solutions.

2.4 Selection of Subjects

Subjects were recruited via a request for consultants being sent to Norwegian consulting companies. The request specified a flexible range of time, for which the consultants would be needed, along with the required education and expertise. Companies replied with résumés of potential candidates and these were then screened to verify that they indeed complied with the requirements. The subjects were required to at least have a bachelor's degree in informatics (or its equivalent), some familiarity with UML (use case, class, sequence, and state diagrams), and some project experience with the following technologies: Struts [11], JavaServer Pages (JSP) [12], Java 2 [13], HTML [14], the Eclipse IDE [15], and MySQL [16].

Note that the recruitment of all subjects could not be completed before the start of the experiment. This was due to several practical reasons:

1. The market for these skilled professionals is very tight.
2. We could not give the consulting companies definite start and end dates as to when the consultant would be working.
3. The consulting companies could not give us an exact start date for consultants.

TABLE 2
Descriptive Statistics: Subjects' Background

Variable	Group	Mean	Standard Deviation	Min	Lower Quartile	Median	Upper Quartile	Max
Age	No UML	31.7	5.9	25	28	30	37	44
	UML	34.5	5.2	28	31	32.5	39	45
Degree (1=bachelors, 2=masters)	No UML	1.7	0.5	1	1	2	2	2
	UML	1.7	0.7	0	2	2	2	2
Graduation Year	No UML	1997.7	3.1	1991	1996	1998.5	2000	2001
	UML	1997.6	3.8	1990	1995	1998.5	2001	2002
Years of Study at University	No UML	4.9	1.9	3	3	5	6	9
	UML	4.7	1.2	2.5	4.5	5	5.5	6
Years of Study in Computer Science	No UML	3.7	1.4	2	2.5	3.5	4.5	6.5
	UML	4.0	1.7	1	3	4.75	5	6
Average Grade in Computer Science Courses	No UML	2.1	0.4	1.7	1.8	2	2.5	2.8
	UML	2.2	0.5	1.5	2	2.1	2.5	3
Years Of Experience With (YOEW) Java	No UML	4.9	1.4	2	4	5	6	7
	UML	4.8	2.1	1	3	5	7	8
YOEW Servlets / JSP	No UML	2.8	1.3	0.5	2	3	4	4
	UML	2.9	1.9	0.3	1	3	4	6
YOEW Struts	No UML	0.9	0.5	0.3	0.5	1	1	2
	UML	1.3	1.5	0	0.021	1	3	4
LOC in Java	No UML	187100.0	320433.2	6000	30000	50000	100000	1000000
	UML	255500.0	397649.0	5000	20000	75000	200000	1000000
LOC in C++	No UML	3200.0	6663.3	0	0	0	1000	20000
	UML	7000.0	11595.0	0	0	0	20000	30000
LOC in C	No UML	1300.0	3199.0	0	0	0	0	10000
	UML	1100.0	2079.0	0	0	0	1000	5000
LOC in C#	No UML	0.0	0.0	0	0	0	0	0
	UML	1300.0	3199.0	0	0	0	0	10000
Used Borland Together	No UML	40%	/	/	/	/	/	/
	UML	20%	/	/	/	/	/	/
Used a UML Tool	No UML	90%	/	/	/	/	/	/
	UML	90%	/	/	/	/	/	/

4. The consulting companies often could not guarantee that the consultant would be available.

Consequently, we assigned the first 10 subjects to the no-UML treatment and the next 10 subjects to the UML treatment. This assignment was also beneficial from a logistical point of view since, at a given point in time, all subjects followed the same experimental procedures. Though this assignment is clearly not "random," there is no reason to believe that the time at which the subjects were available was, in any way, related to their skills. It was, rather, determined by extraneous factors (e.g., contract terminations) and we therefore had no reason to expect any bias in the assignment process. This was confirmed by the analysis in Table 2, which provides background data on the subjects that participated in the experiment and clearly shows that the two groups are indeed comparable in terms of age, education, and experience. Furthermore, simple statistical tests on the data in the table confirmed that none of the differences between the groups is significant.

2.5 Experiment Design

The experiment was conducted on the BESTweb system [17]. The BESTweb system is a company-internal Web-based system developed in Java, using the Struts framework [11], written and documented by the first author of this paper. BESTweb supports research on software cost and effort estimation through the identification of relevant journal papers and conference proceedings [18]. The system is a database front-end client that gives access to information about all journal papers on software cost and effort estimation that have been coded according to the classification categories *research topic*, *estimation approach*, *research approach*, *study context*, and *data set*. Thus, each paper in the system is associated with codes based on this classification scheme; these codes are called the *BEST-codes*. Table 3 provides the basic metrics for the BESTweb system.

The experiment was conducted in two phases for reasons explained in Section 2.4. The subjects in the first phase worked without the UML environment/artifacts. The subjects in the second phase worked with the UML.

The UML documents provide information at a level of detail that one would expect at the end of the design phase

TABLE 3
BESTweb System Metrics

Number of Packages	6
Number of Classes	50
Lines of Java Code	2921
Number of JavaServer Pages (JSP)	17
Number of Attributes	107
Number of Overridden Methods	33
Number of Methods	275
Total Number of Children	13
Maximum Depth of Inheritance Tree	3
Number of Libraries Used	20

[10], including a use case diagram, sequence diagrams for each use case, and class diagrams. These correspond to the most commonly used diagrams in practice and we wanted our results to be as realistic as possible. For the same reason, all conditions in sequence diagrams were simply described in English. The subjects who received UML documentation had to keep it up-to-date. Tables 4 and 5 provide some basic metrics on BESTweb's sequence and class diagrams, respectively. Note that the largest diagrams in both tables are in bold, as we will refer to these in the discussion of the results.

Also, the no-UML documentation was provided to all of the subjects. This documentation included the user's manual, a high-level description of each package in the system and how it relates to the other packages, the third-party libraries, the database schema, and the system deployment instructions. The architecture description document was aimed at reflecting the type of document that exists in the industry for proprietary systems. (At the debriefing session, all but one of the subjects agreed that this documentation was at least as good as the industry standard and seven out of the 20 subjects thought it was better.) The developers also had access to the Javadoc documentation with which all the code was thoroughly documented.

TABLE 5
Class Diagrams: Metrics

Class Diagram	Num. of Classes
no.simula.bestweb.web	**36**
no.simula.bestweb.web.admin	10
no.simula.bestweb.pubmdl	6
no.simula.bestweb.parser	8
no.simula.bestweb.index	6
no.simula.bestweb.db	8

In order to maximize the realism of the experiment, the subjects were not informed of other participants. Furthermore, they were told that, since this work was being performed for a software research laboratory, we wanted to take advantage of the situation to collect data in order to learn how professional software developers work. The subjects were informed in advance that they would be working on a system that was also involved in a study. We deemed that these steps are necessary so that the developers would take the work seriously and not treat it as an exercise. Last, the consultants signed a nondisclosure agreement in order to make sure that they would not disclose information about their work to a potential future subject.

The subjects were also told that, for us to collect valid data, a few rules had to be followed. First, they needed to use the preconfigured development environment (e.g., the Eclipse IDE). Next, they had to work independently: They could not get help from colleagues or the experimenters. Technical questions to the latter had to be asked via e-mail. The reason for this was twofold: 1) so that the subjects would not engage in a technical conversation with the experimenters and 2) so that answers were carefully

TABLE 4
Sequence Diagrams: Metrics

Use Case	Num. of Objects	Num. of Messages
Login	9	17
Change Display Settings	4	3
Filter Publications List using Selected BEST-codes	10	11
Show All Publications	3	2
User's Search	3	2
Query Search	12	18
Sort Publications List	8	12
View Publication Details	7	7
Prepare for showStatistics.jsp	9	8
View Statistics of Publications Per Year	8	7
View Statistics of Publications Per BEST-code Category	13	16
Add User	7	15
Remove User	11	13
Upload a Library File	10	22
System Initialization	5	4
Load BEST-codes and Publications	**17**	**30**

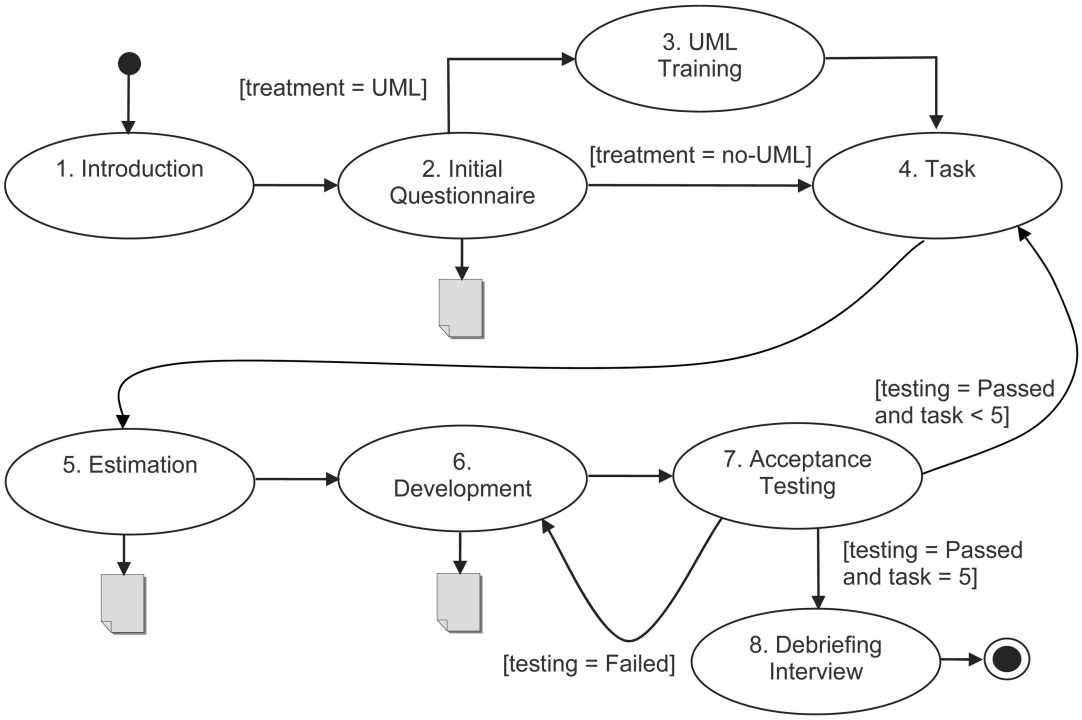

Fig. 1. The experimental process (the documents represent deliverables).

considered by the experimenters (to not give an unfair advantage to the subject). Furthermore, only one task would be given out at a time and the total number of tasks would not be disclosed: This ensured that the developers avoided budgeting their time. Finally, the introductory and debriefing sessions were to be audiorecorded.

The subjects went through the following procedure, as illustrated in Fig. 1:

1. The subject is given an introductory session explaining the manner in which he would be working.
2. The subject answers an initial questionnaire capturing the subject's background and experience (see Table 2).
3. If the subject is in the UML group, the subject receives a UML refresher/tool training session.
4. The subject receives the first task.
5. The subject submits an estimate of the amount of time that he/she thinks the task will take him/her.
6. The subject implements the task.
7. Upon completion, the task is sent in for acceptance testing. The system is then tested by an experimenter based on a system acceptance test plan.
   a. If the test fails, the subject is told about the problem and is asked to fix it to submit the solution again.
   b. If the test passes, the subject receives the next task and repeats the process from Step 4.
8. Upon the completion of all five tasks, debriefing takes place.

Since the experiment was conducted in a manner that was very labor intensive for the experimenters, a maximum of three subjects participated in the experiment at a time. Essentially, the first author of this paper acted as an advanced "customer" that received the intermediate results described above from the subjects, tested and accepted solutions, and, in general, ensured that they did not deviate from the prescribed process.

In the case of the tasks where the developer modified the existing functionality, the test plan was derived from the use case diagram and the sequence diagrams [19]. It ensured that, for every task, all relevant functionality was tested by covering all messages in every relevant sequence diagram. Exceptional scenarios and conditional flows were also accounted for (this sometimes resulted in the same path being covered several times). If the task required the addition of new functionality to the system (i.e., the addition of a new use case), checklist-based testing was applied [20]. The checklist was derived from the functional specifications and was checked: the main flow, exceptional scenarios, and existing functionality.

2.6 The Tasks

In total, the subjects performed five maintenance tasks in a fixed order. It was estimated that it would take the subjects 3-6 days to complete all of the tasks (assuming 7.5 hours/day, not including nondevelopment activities). The tasks consist of adding necessary features of practical value and affect every part of the system: At least one class in every package has to be modified. A summary of the necessary modifications is given in Table 6.

2.6.1 Task 1

The first task requires the addition of functionality to save a user's search query to persistent memory. Also, the user's last search query must be read and reexecuted automatically upon their relogon to the system.

The task requires that the developer understands how the system starts up, interacts with the database, and

TABLE 6
Minimal Modifications Necessary for Implementing Each Task

Task	Necessary Modifications
1	At a minimum, the task requires modifications to 3 classes containing respectively 5, 10, and 80 lines of code.
2	The task is unique in that it has only one correct solution which the developer can obtain in a limited and repetitive manner. The task requires a modification of a few lines of code in nine classes.
3	At a minimum, the task requires the addition of six new classes, two JSPs, and the modification of one class and one JSP.
4	At a minimum, the task requires a modification of four classes (30 LOC in total) and the addition of three other classes (80 LOC in total).
5	At a minimum, the task requires the addition of four classes and two JSPs, and the modification of one class and one JSP.

executes a query. In terms of coding, the task is small in size as one of its purposes is to introduce the subject to the system. However, initially, the task may seem complex because developers must gain a basic understanding of several parts of the system before the task can be solved.

2.6.2 Task 2

The second task requires the extension of the system to handle an additional piece of data from an input file (in XML format) used to update the publications in the BESTweb system. The system already partially handles the data: If it encounters the presence of the data in the file, it warns the user that the data is not supported. The developer is asked to add support for this data by extending the domain model, the GUI, and the search functionality.

The task requires that the developers gain an understanding of the system's domain model, GUI, and search functionality. The task can also be solved without a complete understanding of the implications of their changes by searching for every occurrence of a similar piece of data and then copying/pasting and modifying the code accordingly. For example, if the domain model has a *publication* and the system already handles the *title* property of the *publication* but not the *author* property, the developer could search for every occurrence of *title* in the system, then copy and paste the code that handles *title* and replace *title* with *author*.

2.6.3 Task 3

The third task requires the developer to add completely new functionality to the system (as opposed to making modifications to existing functionality). Recall that the BESTweb system was designed to aid in working with cost and effort estimation papers [18]. Each of these papers is classified according to a categorization scheme (using the BEST-codes). This task asks the developer to add functionality to the system that extends the manner in which cost and effort estimation metadata associated with each publication is dealt with, specifically the ability to add categorization categories and corresponding codes. Without such functionality, the user would have to manually add the code in the database and restart the system. The restart would be necessary so that the new code would be associated with the corresponding publications.

The task is complex as it requires the developer to understand most parts of the system. Furthermore, the developer has to create a GUI by using JSP and Struts (something that a large number of developers ended up struggling with).

2.6.4 Task 4

Task 4 asks the developers to add caching logic to the system so that if statistics for all of the publications in the system are requested, the cached results are used (so as to decrease the computational load on the system).

The task requires the developers to understand the statistical data and how it is generated by the system. The developers also have to ensure that when, for example, a new code is added to the system (via the functionality that they added in Task 3), these cached results are updated.

The task forces the developer to deal with the *Best Codes Manager* (*BCM*) class. The BCM class is noteworthy, as developers frequently found it nontrivial to understand for the following reasons:

- It contains a cached copy of all the BEST-codes found in the database and this requires that the developers understand how the caching strategy works.
- It retrieves objects from a qualified association that points to another qualified association.
- It contains thread-safe logic.

2.6.5 Task 5

The last task is, in fact, a continuation of Task 3; thus, the comments for that task apply to Task 5 as well. Whereas in Task 3, the developers are asked to add functionality where the user could add new types of publications codes to the system, in this task, the developers are asked to add functionality where the users could delete existing publication codes from the system.

2.7 Instrumentation and Measurement

The instrumentation and measurement process was specified before the experiment began and outlined exactly how the interaction with the subjects would be performed; it also outlined how the data would be collected when interacting with the subjects.

The data sources are (see Fig. 1) listed as follows:

- for every subject, an initial questionnaire capturing the subject's background and experience (see Table 2),

- for every submission, a copy of their entire source code,
- for every submission by a UML subject, the estimated percentage of time spent on reading and updating the UML,
- for every submission, acceptance test reports (generated by the experimenters), and
- for every subject, an audiorecorded semistructured debriefing interview (conducted after the developer has finished all of the tasks; see Section 2.8.2).

An important point in this experiment was the fact that a subject's solution (submission) was not accepted until it passed all the functional tests. Such a setup ensures that 1) in their ultimate form, all tasks conform to the predefined specifications and 2) every subject completes every task. Full conformance to the specifications is important to be able to compare final solutions in terms of effort, for example. With some subjects being slower than others, it was important not to fix the time allocated for tasks to ensure 2) and to be able to observe differences in effort. The disadvantages of this decision are related to cost and logistics. Cost is higher, as the slower subjects will need to be paid more to complete the tasks. The logistics are more difficult, as it cannot be anticipated how long the subjects will take to complete the tasks.

The number of resubmissions and the reasons for their need were recorded. The resubmission problems were categorized as either *omissions* or *faults*, where a submission that did not fully implement the specified functionality was defined as an *omission*.

The solutions for each task were also assessed for their design quality in terms of following proper OO design principles [10]. This was done by first breaking each task into *subtasks* and then specifying all acceptable solutions for that subtask. Each *subtask* corresponds to a subset of an entire task's functionality; the level of granularity was set to one that is high enough to make it possible to compare the corresponding solutions (code) across all of the subjects. For example, in the case of a task that requires access to the database, the code that accesses the database could be placed almost anywhere, even though the proper place for it is in the package that specifically deals with the database interaction. Even though code that is placed in, say, the presentation layer, would pass the functional correctness tests, this would be an unacceptable solution (leading to code decay). Thus, each possible solution for each subtask was rated as either *acceptable* or *unacceptable* according to the predefined criteria. Specifically, a solution to a subtask with one of the following problems would be deemed as *unacceptable*:

- duplication (copying and pasting) of existing code instead of direct use of that logic (e.g., copying and pasting a sort method),
- addition of new design elements that current design elements could have handled (e.g., creating a new partial user class even though an existing user class is present),
- incorrect placement of logic in a class,
- distribution of logic throughout the application when adding it to just one place would have had the same effect, and
- use of the try/catch mechanism as a normal part of the application's logic.

2.8 Analysis Procedure

The analysis procedure included both quantitative and qualitative components. The quantitative data was the main source for testing the hypotheses, whereas the qualitative data was analyzed in an attempt to gain a deeper understanding of the work processes of the subjects, which, in turn, could potentially offer additional complementary evidence and, to some extent, explain the quantitative results.

2.8.1 Quantitative Analysis

Univariate analyses of the dependent variables were performed to test the hypotheses both individually for each task and across all tasks. For all dependent variables T, T', C, C'', C''', and Q, two-sample t-tests were performed [21]. In addition, to reduce potential threats to the validity of statistical conclusions resulting from violations of the t test assumptions, nonparametric Wilcoxon rank sum tests were also performed [21]. Additionally, with respect to design quality Q, Fisher's Exact Test [21] was used to test the difference in *proportion of subjects*, with solutions being scored as *acceptable* for each subtask (of each task).

The level of significance for the hypotheses tests was set to $\alpha = 0.05$. However, the reader should bear in mind that we perform multiple tests and, in order to allow for a stricter and more conservative interpretation of the results (e.g., using a Bonferroni procedure or one of its variants [22]), we provide p-values.

Furthermore, it is often useful to know not only whether an experiment has a statistically significant effect but also the *size* of any observed effects. Thus, for the dependent variables on time and correctness, the effect size was calculated using Cohen's d, which is defined as the difference between two means divided by the pooled standard deviation for those means [23]. In our case, we calculated the difference in means between the UML and the no-UML groups so that a positive value of d corresponded to the UML treatment being *beneficial*. To interpret the results, Cohen suggested that $d = 0.2$ is indicative of a *small* effect size, $d = 0.5$ a *medium* effect size, and $d = 0.8$ a *large* effect size.

Given the small number of subjects, we also fitted multivariate Analysis of Covariance (ANCOVA) models for the time and correctness data across all tasks and subjects. The average grade in computer science courses (see Table 2) of each subject was included as a *covariate* to adjust for individual differences between the subjects and *UML* and *Task* (and their interaction) were the independent variables. The use of the covariate, combined with the fact that we had a total of 100 data points for each dependent variable (20 subjects and five tasks), resulted in increased statistical power compared to the less sophisticated univariate analyses. However, since the observations of individual tasks for a given subject are correlated, the ANCOVA assumptions of independent observations would be violated. We thus resorted to a statistical technique known as Generalized Estimating Equations (GEEs) [24] to estimate the parameters (i.e., the effect of *Grade*, *UML*, and *Task*) of the models. GEE is an extension of Generalized Linear Models, developed specifically to accommodate data that is correlated within clusters (here being the individuals). The results of the GEEs were entirely consistent with the univariate results and are hence not included in this paper due to space constraints. Still, it

implies that the univariate results presented in this paper probably do not suffer from Type II errors as a result of the lack of power since the multivariate analyses of covariance with 100 data points showed consistent results.

2.8.2 Qualitative Analysis

A semistructured debriefing (interview) session was held with each developer immediately upon completion of the tasks. An interview guide with relatively open questions was prepared and all sessions were audiorecorded. The subjects were asked about their style of working (e.g., how they gained an understanding of the system) and the problems that they faced (e.g., what they suspect led to the introduction of every fault). The interviews lasted between 50 and 98 minutes for the no-UML subjects (70 minutes on average) and between 91 and 169 minutes for the UML subjects (123 minutes on average). The interviews varied in length due to several reasons. Each error had to be discussed and the total number of errors differed across developers. Furthermore, some subjects were more talkative than others. The interviews with the subjects in the UML group took longer since, additionally, the usage of UML was thoroughly discussed. The length of this additional discussion varied, depending on the extent to which the developer took advantage of the UML and the developer's prior experience with UML. Developers who did not take complete advantage of the UML diagrams could not comment as much as those who did.

The amount of information extracted from the subjects also varied due to other reasons. The developers spent 1-2 weeks on the experiment. The longer it took them to complete all of the tasks, the harder it was for them to remember details of what happened in earlier phases of the experiment. Unfortunately, the developer could not be interviewed after completion of each task as that would influence her (e.g., discussions examining why the developer chose a certain solution and not an alternative might have given the developer a deeper insight into the system). Furthermore, some people can provide insight into their thought process better than others and others have a very hard time forming an opinion [25].

Content analysis [26], a data reduction technique, was then applied to the audio recordings of the semistructured interviews in the following manner:

1. An initial set of codes was derived from the interview questionnaire.
2. The interviews were played back, and the subjects' answers and opinions were transcribed and coded. This made the coding traceable.
3. If a subject's opinion or answer was not possible to code with an existing code, a new code was declared.
4. Once this was done for all of the interviews, the codes were then reviewed, refined, and finalized.
5. All of the interviews were then replayed and recoded with the finalized coding scheme.

The coding schema (see the Appendix, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.15) encompassed the problems that the developers faced, their thoughts on the system, and how they worked. Additionally, the no-UML interviews were coded for problems that occurred due to the models not being available. The UML interviews were coded to capture the manner in which the UML diagrams were used. The purpose of the analysis was to better understand how access to UML documentation made a difference.

3 Experimental Results

The results from the analysis procedures described in Section 2.8 are now shown and dissected. The section first presents and discusses the descriptive statistics, univariate results, and the qualitative results, and discussion follows. Last, the main results from the experiment are summarized in Section 3.5.

3.1 Descriptive Statistics and Univariate Analysis

Recall that four hypotheses are tested with regard to the effect of UML on the following:

1. the time to perform the change task excluding diagram modification (using the variable T),
2. the time to perform the change task including diagram modification (using the variable T'),
3. submission correctness (using the variables C, C', and C''), and
4. design quality (using the variable Q).

These are examined respectively.

Note that the Wilcoxon test and the t-test provide consistent results in all but one case (C for Task 1 in Table 9 gave a p-value of 0.053). Due to the nonnormal distribution of the data, the Wilcoxon p-values are used in the discussions. The significant values also appear in bold in the tables.

3.1.1 Time

Table 7 shows the descriptive statistics and univariate results for time (in minutes) across all of the tasks and for each task, respectively. For each dependent variable (denoted by the Var column), the results are presented for each treatment (Treat): the mean of the subjects in the group, the standard deviation, the smallest data point (Min), the lower quartile (Lower Quart), the median (Med), the upper quartile (Upper Quart), the largest data point (Max), the effect size as a percentage difference between the two means (% diff), the effect size using Cohen's d, the p-value from the t test, and the p-value from the Wilcoxon test.

In terms of time excluding diagram modifications T, the no-UML group spent 1.4 percent more time in total. The development time is shorter on the first and the two largest tasks (3 and 5) for the UML group. The variance is also smaller for the UML group on the largest tasks (3 and 5), similar to Tasks 1 and 4, and larger on Task 2. The minimum is smaller for the UML group in the case of the two largest tasks and is larger in the case of the other tasks. The maximum is smaller for the two largest tasks for the UML group and is larger for other tasks. However, when including the time that the UML group spent on updating the UML documentation T', then, overall, the no-UML group finished the tasks 14.5 percent faster than the UML group. On the two largest tasks (3 and 5), the UML overhead is the lowest (around 10 percent, even though there is a lot of UML to update on those tasks). The variance is also smaller for these largest tasks for the UML group, similar to Tasks 1 and 4, and larger on Task 2. The

TABLE 7
Descriptive Statistics and Univariate Results: Time

Task	Var	Treat	Mean	Std Dev	Min	Lower Quart	Med	Upper Quart	Max	% diff	Coh-en's d	*t* test	Wilc-oxon
All	T	No UML	2030.3	921.6	950.0	1318.0	1913.0	2759.0	3458	1.4%	0.04	0.47	0.46
		UML	2001.6	602.8	1112.0	1618.8	2074.1	2429.0	2838				
	T'	No UML	2030.3	921.6	950.0	1318.0	1913.0	2759.0	3458	-14.5%	-0.36	0.43	0.35
		UML	2325.4	697.1	1410.0	1855.0	2314.0	2849.0	3610				
1	T	No UML	327.1	145.0	145.0	183.0	345.0	355.0	565	6.8%	0.16	0.37	0.41
		UML	304.7	142.1	154.0	198.5	298.8	355.0	622				
	T'	No UML	327.1	145.0	145.0	183.0	345.0	355.0	565	-16.1%	-0.36	0.44	0.64
		UML	379.7	151.2	194.0	265.0	331.5	480.0	685				
2	T	No UML	234.9	117.8	130.0	140.0	172.0	332.0	445	-23.3%	-0.39	0.81	0.88
		UML	289.7	161.9	173.5	210.0	220.0	315.0	721				
	T'	No UML	234.9	117.8	130.0	140.0	172.0	332.0	445	-24.0%	-0.40	0.39	0.23
		UML	291.2	162.3	180.0	210.0	222.5	315.0	725				
3	T	No UML	681.4	417.9	323.0	410.0	533.5	765.0	1471	9.0%	0.19	0.34	0.29
		UML	620.4	196.3	337.5	479.0	646.0	722.0	894				
	T'	No UML	681.4	417.9	323.0	410.0	533.5	765.0	1471	-11.1%	-0.22	0.62	0.25
		UML	757.3	237.6	435.0	535.0	800.0	910.0	1125				
4	T	No UML	318.0	157.8	135.0	155.0	325.5	477.0	552	-6.0%	-0.13	0.61	0.63
		UML	337.1	132.1	184.0	230.0	303.0	440.0	542				
	T'	No UML	318.0	157.8	135.0	155.0	325.5	477.0	552	-21.7%	-0.45	0.33	0.35
		UML	387.0	151.0	220.0	270.0	342.5	540.0	640				
5	T	No UML	468.9	295.0	197.0	220.0	377.5	639.0	1119	4.1%	0.08	0.43	0.49
		UML	449.8	200.6	140.0	301.0	454.8	550.3	746				
	T'	No UML	468.9	295.0	197.0	220.0	377.5	639.0	1119	-8.8%	-0.17	0.71	0.49
		UML	510.2	191.8	220.0	355.0	535.0	590.0	812				

TABLE 8
Descriptive Statistics: Percentage of Time Spent on UML

	Task 1	Task 2	Task 3	Task 4	Task 5	Average
Reading	20.9%	21.5%	10.1%	15.1%	6.5%	14.8%
Updating	20.8%	0.6%	17.7%	12.9%	14.6%	13.2%

minimum is always smaller and the maximum is larger for the no-UML group in the case of the two largest tasks (3 and 5); otherwise, this is smaller. None of the differences in time is, however, statistically significant at $\alpha = 0.05$. Furthermore, the effect size measure d is well below 0.5 for both T and T' and, as explained in Section 2.8.1, this indicates a *small* treatment effect.

In order to gain a deeper understanding of how the UML subjects spent their time, Table 8 shows the percentage of time that the UML subjects have spent on reading and updating the UML documentation on each task. Note that, unlike the difference between T and T', which compares subjects in the two groups, this measure only deals with the subjects in the UML group and the amount of time that they self reported to have spent on UML. On average, 14.8 percent of the time spent on each task was spent reading the UML and 13.2 percent of the time was spent on updating the UML. Note that, in Task 2, there was virtually no UML to be updated due to the fact that the changes consisted of adding attributes and associations on class diagrams, and those are updated automatically. Also remember that Tasks 3 and 5 were the largest tasks and were fairly similar, while Task 4 was the most complex task. Keeping this in mind, the table shows that the subjects spent the most time reading the UML while working on the tasks that dealt with parts of the system that they were not familiar with (Tasks 1-4), as the amount of time spent on reading the UML in Task 5, the only task that dealt with functionality the developers were already familiar with, is significantly smaller. In particular, the largest percentage of time used for reading the UML was spent during Tasks 1 and 2 when the subjects were least experienced with the system. By Task 5, relatively little time is used on reading the UML: only 6.5 percent. Next, while the time used for updating the UML is dependent on the amount and types of changes introduced to the code, a general trend can be seen when looking at Tasks 1, 3, and 5. Task 1 requires relatively few changes in the UML, but, at this time, the developers have very little experience at updating the UML using the tool (BTE). While Tasks 3 and 5 require the most changes, the types of changes are very similar. Thus, one can see that the percentage of time that the subjects spend on updating the UML tends to decrease as they gain experience with this activity.

TABLE 9
Descriptive Statistics and Univariate Results: Correctness

Task	Var	Treat	Mean	Std Dev	Min	Lower Quart	Med	Upper Quart	Max	% diff	Cohen's d	*t* test	Wilc-oxon
All	C	No UML	5.3	2.7	2.0	4.0	4.0	9.0	9.0	54.7%	1.11	**0.0105**	**0.0313**
		UML	2.4	2.5	0.0	0.0	2.0	5.0	6.0				
	C'	No UML	1.2	2.4	0.0	0.0	0.5	1.0	8.0	50.0%	0.34	0.2390	0.4100
		UML	0.6	0.8	0.0	0.0	0.0	1.0	2.0				
	C''	No UML	1.0	0.7	0.0	1.0	1.0	1.0	2.0	70.0%	1.15	**0.0075**	**0.0191**
		UML	0.3	0.5	0.0	0.0	0.0	1.0	1.0				
1	C	No UML	1.6	1.0	0.0	1.0	1.5	2.0	3.0	50.0%	0.76	0.053	**0.040**
		UML	0.8	1.1	0.0	0.0	0.0	2.0	3.0				
	C'	No UML	0.3	0.7	0.0	0.0	0.0	0.0	2.0	0.0%	0.00	0.500	0.639
		UML	0.3	0.7	0.0	0.0	0.0	0.0	2.0				
	C''	No UML	1.0	0.7	0.0	1.0	1.0	1.0	2.0	70.0%	1.15	**0.008**	**0.019**
		UML	0.3	0.5	0.0	0.0	0.0	1.0	1.0				
2	C	No UML	0.1	0.3	0.0	0.0	0.0	0.0	1.0	100.0%	0.47	0.165	0.500
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
	C'	No UML	0.1	0.3	0.0	0.0	0.0	0.0	1.0	100.0%	0.47	0.165	0.500
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
	C''	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
3	C	No UML	0.5	0.7	0.0	0.0	0.0	1.0	2.0	80.0%	0.74	0.060	0.125
		UML	0.1	0.3	0.0	0.0	0.0	0.0	1.0				
	C'	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
	C''	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
4	C	No UML	1.4	1.8	0.0	0.0	1.0	2.0	5.0	50.0%	0.47	0.149	0.186
		UML	0.7	1.1	0.0	0.0	0.0	1.0	3.0				
	C'	No UML	0.8	1.5	0.0	0.0	0.0	1.0	5.0	62.5%	0.45	0.171	0.361
		UML	0.3	0.5	0.0	0.0	0.0	1.0	1.0				
	C''	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
5	C	No UML	1.7	1.8	0.0	0.0	1.0	3.0	5.0	52.9%	0.62	0.181	0.130
		UML	0.8	1.0	0.0	0.0	0.5	1.0	3.0				
	C'	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				
	C''	No UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0	/	/	/	/
		UML	0.0	0.0	0.0	0.0	0.0	0.0	0.0				

3.1.2 Correctness

Table 9 follows the presentation style used in Table 7 and introduced in Section 3.1.1, but deals with correctness instead of time. In terms of the number of tasks submitted with a fault C, we see that the UML group had 50 percent to 100 percent fewer faults in each and every task and 54.7 percent fewer faults overall. The variance is smaller on all but the first task (where it is almost equal) for the UML group. The minimum is zero in both groups across all tasks. The maximum is always smaller for the UML group except for on the first task (where it is equal). The differences are statistically significant in Task 1, with a p-value of 0.04, and across all the tasks, with a p-value of 0.03. Overall, there is a *large* positive effect of the UML treatment on correctness, as indicated by Cohen's d being well above 0.8 across all tasks.

The faults that broke the existing functionality C' occurred 0 percent to 100 percent less often in the UML group throughout the three tasks, where it was observed (1, 2, and 5) and 50 percent less overall. There were no significant differences at $\alpha = 0.05$.

Last, faults stemming from not taking into account all existing behavior C'' only occurred in Task 1 submissions (where the subjects are completely new to the system). The UML group had 70 percent fewer faults of this type, with a lower variance. The minimum is zero in all of the cases and the maximum is larger for the no-UML group. The difference is significant, with a p-value of 0.02. As for C, the effect size measure d is well above 0.8, indicating a *large* benefit of UML on C''.

TABLE 10
Descriptive Statistics and Univariate Results: Design Quality

Criteria	no-UML	UML	% diff	Fisher's Exact Test	*t* test	Wilcox
Task 1, Subtask 1, Acceptable	8	10	25.0%	0.2368	/	/
Task 1, Subtask 2, Acceptable	2	6	200.0%	0.0849	/	/
Task 1, Subtask 3, Acceptable	6	9	50.0%	0.1517	/	/
Task 1, Total, Acceptable	16	25	56.2%	/	**0.0006**	**0.0025**
Task 3, Subtask 1, Acceptable	7	8	14.3%	0.5000	/	/
Task 3, Subtask 2, Acceptable	9	10	11.1%	0.5000	/	/
Task 3, Subtask 3, Acceptable	10	9	-10.0%	1.0000	/	/
Task 3, Subtask 4, Acceptable	7	6	-14.3%	0.8251	/	/
Task 3, Subtask 5, Acceptable	10	9	-10.0%	1.0000	/	/
Task 3, Total, Acceptable	43	42	-2.3%	/	0.6077	0.5577
Task 4, Subtask 1, Acceptable	8	7	-12.5%	0.8483	/	/
Task 4, Subtask 2, Acceptable	9	9	0.0%	0.7632	/	/
Task 4, Subtask 3, Acceptable	6	3	-50.0%	0.9651	/	/
Task 4, Subtask 4, Acceptable	1	2	100.0%	0.5000	/	/
Task 4, Total, Acceptable	24	21	-12.5%	/	0.7268	0.7329
Task 5, Subtask 1, Acceptable	7	9	28.6%	0.2910	/	/
Task 5, Subtask 2, Acceptable	6	6	0.0%	0.6750	/	/
Task 5, Total, Acceptable	13	15	15.4%	/	0.2837	0.3707
All Tasks, Acceptable	96	103	7.3%	/	0.2288	0.2187

3.1.3 Design Quality

Table 10 shows the descriptive statistics and the univariate analysis results for design quality Q. The first column in the table indicates the design quality criteria, which can be one of the following: the number of subjects with an acceptable solution to a subtask (with a maximum of 10, in which case every subject in the group had an acceptable solution to the task), the subtasks aggregated (summed) to the task level, and the subtasks aggregated across all five tasks. Remember that, in Task 2, there was no flexibility in the implementation and there was essentially only one way in which the task could be solved to obtain a functionally correct change; thus, data for this task is absent from the table.

Overall, the UML group had 7.3 percent more acceptable solutions. Furthermore, a significant difference was found for Task 1 (overall), where the subjects lack familiarity with the system and are changing the existing functionality; the UML group's design quality score was 56.2 percent better, with a p-value of 0.0025. Otherwise, the differences in quality are relatively small and not statistically significant.

3.2 Discussion of Quantitative Results

Overall, when looking at the total time T that the subjects spent on the five tasks, we see that the UML group completed the tasks slightly faster (1.4 percent) than the no-UML group (Table 7). This difference is not practically or statistically significant. When we take the time that it takes to update the UML documentation into account, we see that the UML group spent 14.5 percent more time on the five tasks, though this difference is not statistically significant either (Table 7). The observed cost of keeping the UML documentation updated can be better understood if we look at Tasks 1 and 5. During Task 1, the UML subjects have very little experience in using the UML tool; thus, a certain learning curve can be expected. On the other hand, by the time that developers get to Task 5, the UML subjects are fairly comfortable with using the tool and are adding functionality to a system that they are fairly familiar with. This is clearly shown in Table 8. During Task 1, 20.9 percent of the time is spent reading the UML and 20.8 percent of the time is spent on updating it. By Task 5, a larger percentage of time is spent on maintaining the UML (14.6 percent) than reading it (6.5 percent). In Task 1, the UML subjects spent 16.1 percent more time on the task than the no-UML subjects updating the UML documentation (Table 7). This goes down by almost half in Task 5, i.e., to 8.8 percent. Thus, the time spent on the UML in Task 1 could be considered a worst-case scenario that can be expected in terms of time overhead. We use the term "worst-case scenario" since this is what the cost of the UML would be if there would be no other noticeable gains from the use of the UML. But, this is not the case since the UML group performed much better in terms of correctness on every task, i.e., 50 percent to 100 percent better. The difference is statistically significant in the first task and across all of the tasks (for C and C'''). This makes sense as, during the first task, the developers were least familiar with the system; thus, they were more likely to introduce a fault. This explains why the UML clearly helped the developers on the first task. Taking a closer look at correctness reveals that the UML group always did as well as or better than the no-UML group, even if not all of the differences are statistically significant. This is probably due to the fact that the developers in the UML group gained a deeper understanding of the system, thanks to the UML documentation, as further suggested by the qualitative analysis in Section 3.3.

Furthermore, in terms of the design quality Q, significant gains are found in Task 1. Our explanation is that the fact that the developers are completely new to the system is offset by the presence of the UML documentation, significantly so. The subjects in the UML group delivered a higher quality solution with lower complexity than did the subjects in the no-UML group. This is important as this result suggests that the UML can help prevent code decay

TABLE 11
Summary of Qualitative Results Applicable to All Subjects

Row	Code	no-UML	UML
(a)	In my opinion, the quality of the BESTweb system was below average compared to what exists in industry.	0/10	1/10
(b)	In my opinion, the quality of the BESTweb system was average compared to what exists in industry.	2/10	7/10
(c)	In my opinion, the quality of the BESTweb system was above average compared to what exists in industry.	8/10	2/10
(d)	In my opinion, the documentation that came with the BESTweb system was below average compared to what exists in industry.	1/10	0/10
(e)	In my opinion, the documentation that came with the BESTweb system was average compared to what exists in industry.	4/10	8/10
(f)	In my opinion, the documentation that came with the BESTweb system was above average compared to what exists in industry.	5/10	2/10
(g)	Reported that lack of Struts experience caused considerable problems.	5/10	8/10
(h)	Reported that lack of GUI development experience in JSP/Struts caused considerable problems.	5/10	5/10
(i)	Reported feeling rusty with the Java language.	0/10	3/10
(j)	Reported a lot of trouble understanding the *Best Codes Manager* (BCM) –related portion of the system.	5/10	2/10
(l)	Reported having trouble understanding parts of the system due to poor naming of variables or classes.	1/10	3/10
(k)	Reported only skimming the BESTweb architecture document (instead of thoroughly reading it).	6/10	6/10

[27] in real systems when developers are not familiar with the system.

Tasks 2 and 4 did not follow the general trend, where T was lower for the UML group (and, in fact, T' ended up being higher as well). We speculate that the reasons for this are task specific. Task 2 is unique in that it can be solved in a limited and repetitive manner. As discussed in Section 2.6.2, this was possible as all that the developer had to do was to extend functionality in the system by copying/pasting code and changing the variable name. Furthermore, once the changes were made, either the application worked or it did not. Hence, the developer could have solved the task by having a hunch as to how it can be done, doing the change in a mechanical manner (without fully understanding the consequences of the change), and testing the application to see if the hunch was correct. The no-UML subjects were more predisposed to solving the task in such a manner as they dealt directly with the code. Once they had identified a relevant piece of functionality, it was natural for them to perform a search in the code, instantly revealing the other relevant locations. The UML subjects were less likely to perform such a search as they primarily used the diagrams and, therefore, performing a search to find relevant occurrences is not as easy, natural, or possible to perform. Instead, the UML subjects tried to find the right place in the code by studying the diagrams. This procedure inhibited the use of the search/copy-and-modify approach by the UML subjects. Conversely, this made the UML subjects less likely to guess the answer. Unfortunately, no quantitative data can back up this hypothesis, apart from the large amount of time that the UML subjects spent on this task (even though, virtually, no UML had to be updated; see Table 8).

In the case of Task 4, T is slightly higher for the UML group as well; this may be due to the fact that it was the most complex task to solve. The complexity stemmed from the fact that the most complex parts of the system had to be understood before the task could be solved correctly. Thus, if proper due diligence was not performed on the task, faults could be easily introduced.

In summary, the quantitative results show the following:

- The UML was always beneficial in terms of functional correctness (introducing fewer faults into the software).
- The UML was slightly more costly in terms of time if the UML documentation was to be updated (though, slightly less costly if it was not), though these results were not significant.
- The UML helped produce code of better quality when the developers were not yet familiar with the system.
- The largest gains were experienced during the first task. This is an important finding as developers in industry are often faced with the "first task" scenario due to high staff turnover and involvement on a very large system (where any one developer is only familiar with a small portion of the system).

TABLE 12
Summary of Qualitative Results Applicable Solely to the No-UML Subjects

Row	Code	Number of Developers
(a)	Reported missing models	5/10
(b)	Reported drawing own UML diagrams	2/10
(c)	Reported having problems gaining an overview of functionality	5/10
(d)	Specified that no benefit would be gained from the presence of UML	2/10

TABLE 13
Summary of Qualitative Results Applicable to the UML Subjects

Row	Code	Number of Developers
(a)	Took advantage of the Use Case Diagram	8/10
(b)	Took advantage of Sequence Diagrams	10/10
(c)	Took advantage of Class Diagrams	5/10
(d)	Took advantage of Page Flow Diagrams	5/10
(e)	Took advantage of the Statechart Diagram	1/10
(f)	Found the UML diagrams useful	10/10
(g)	Reported that UML was helpful in gaining an overview	7/10
(h)	Did not use UML to the maximum extent, reason: not used to working with UML	4/10
(i)	Did not use UML to the maximum extent, reason: did not want to risk wasting the client's time – a subset of (*h*)	3/10
(j)	Used the UML for navigation (browsing the system)	8/10
(k)	Used the UML diagrams to find exact code change places	6/10
(l)	Did not look at the UML diagrams during development	2/10
(m)	At some point was unable to gain an understanding of a piece of the system using UML	7/10
(n)	Reported problems comprehending the large Sequence Diagram (quantified in Table 4)	4/10
(o)	Reported problems comprehending the large Class Diagram (quantified in Table 5)	3/10
(p)	Did not trust the UML diagrams to be accurate	1/10
(q)	Reported needing Javadoc comments when looking at the diagrams	3/10
(r)	Found the UML tool to be problematic	9/10
(s)	Felt he completed the tasks faster thanks to the UML	2/10
(t)	Felt he completed the tasks slower due to the UML	2/10
(u)	Extensive experience of working with UML	1/10
(v)	Found the UML tool training adequate	10/10

Thus, one can conclude that, overall, using the UML can be beneficial when a developer must extend a nontrivial system that he/she is unfamiliar with, which is a very typical occurrence in industry.

3.3 Qualitative Analysis Results

First, results applicable to all of the subjects are presented and summarized in Table 11. Next, results applicable to the subjects in the no-UML group are presented and summarized in Table 12. Finally, results applicable to the subjects in the UML group are presented and summarized in Table 13. Note that a more detailed breakdown of the data presented in these tables can be found in Tables 17 and 18 in the Appendix, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.15.

Table 11 consists of two types of information: Rows a to f reflect the subjects' answers to questions that appeared on the debriefing questionnaire, whereas rows g to k reflect information offered by the subjects during the debriefing

session (e.g., while discussing the difficulties that they experienced).

One important topic of discussion during the debriefing interviews was the problems that all subjects faced while performing the tasks. With regard to language and framework, the main sources of problems were the lack of Struts experience (row g), the lack of GUI development experience in JSP/Struts (row h), and being out of practice with the Java programming language (row i). The UML subjects reported having more problems than the no-UML subjects in this area. Also, the UML subjects reported having more problems due to the poor naming of variables or classes (row l). Last, problems in understanding specific areas of the system most often point to the *BCM* (a complex part of the system discussed in Section 2.6.4). Half of the no-UML subjects reported problems in understanding this part of the system compared to only two subjects in the UML group.

In terms of the subjects' opinion of the BESTweb system's quality (rows a to c), most subjects in the no-UML group thought that the system was better than average, as opposed to just two in the UML group. The one (UML) developer who considered the system below average was used to developing safety-critical systems. In terms of the BESTweb system's documentation (rows d to f), again, more no-UML subjects thought that the system documentation was above average than their UML counterparts. The one (no-UML) developer who felt that the documentation was below average thought so due to the lack of any domain models in the documentation. Furthermore, the UML did not seem to have an effect on the effort spent on the architecture document (row k).

Information offered by the no-UML subjects is presented in Table 12. Half of the subjects reported missing the presence of at least some models (e.g., the domain model). Two subjects reported drawing their own UML diagrams to aid their comprehension of the system. Half of the subjects also reported having problems in grasping an overview of the functionality of interest in the system (two of these five subjects did not report missing models). Two of the subjects stated that they felt that they would have gained no benefits from the presence of the UML.

The qualitative results for the UML group, as presented in Table 13, suggest that the extent to which the UML documentation was used, as well as its impact, varied among the UML subjects. The experiment required that all subjects update the diagrams before they moved on to the next task. However, the extent to which they used the UML models to identify change locations prior to performing code modifications varied greatly among subjects.

First, in terms of the types of UML diagrams used, rows a to e show that all subjects used the sequence diagrams, all but two used the use case diagram, half used the class diagrams and the page flow diagrams (these diagrams modeled the Struts/JSP elements' interaction with the rest of the application), while only one subject used the statechart diagram. Even though most subjects used a subset of the available diagram types, all said that they found the UML generally useful (row f). Seven of the 10 also said that they found that the UML aided in getting an overview of the system (row g), even though two of these seven also stated that they used the UML to a lesser extent (see descriptions for rows h and i).

Taking a closer look at the extent to which the UML was used reveals that four subjects consciously limited their use of the UML (row h). One of these four subjects thought that taking advantage of the UML was optional in the case where he thought it would make him more efficient (thus, he chose not to use it "too much" to avoid wasting the client's time). Three of these four subjects reported that the low use was due to the anxiety that it would take more time (to complete the tasks) and, to a lesser extent, habit (row i). Furthermore, two of these three subjects (in row i) struggled with implementing the tasks; thus, the added burden of learning to use the UML in such a manner was too much for them.

Rows j to l deal with the manner in which the UML was used and reveal that most of the subjects used the UML diagrams to navigate around the system, more than half used it to find the exact places in the code that needed to be modified, and only two did not look at the UML once they started development.

Problems and difficulties experienced by the subjects while using the UML are covered in rows m to r. One of the most commonly reported frustrations was (row m) the inability to extract what they were looking for from the UML diagrams (at some point in time); three subjects explicitly expressed frustration that they had to go into the code (and leave the diagram) to look at comments (row q). This stemmed from three main issues: lack of experience with understanding nontrivial UML diagrams (it is one thing to understand individual constructs like a message on a sequence diagram and another to take these individual pieces of information and combine them into a complete mental model), lack of direct access to code comments from the diagrams (class and association descriptions in the comments may be crucial to understanding a cluster of classes on a class diagram), and lack of knowledge as to how a UML representation translates to code (e.g., composite aggregation). The last point was observed with respect to the statechart diagram. Even though the subject understood the statechart diagram, he could not see how this was implemented in the code due to a lack of familiarity with the state pattern [28] (note that the UML tool did not support the linking of the statechart diagram to code).

It is important to point out that the four developers who made lesser use of the UML (row h) are a subset of the seven (row m) aforementioned problems. The other commonly reported frustration dealt with the UML tool (row r); the main complaints were that it was very painful to update the diagrams and the presence of faults (bugs). Another commonly reported issue dealt with the largest sequence (row n) and class (row o) diagrams, which was reported by six subjects. Finally, it is interesting to point out that trusting the accuracy of the UML diagrams was not a problem (row p).

In terms of the subject's perception concerning time saving, including the time that it took to update the sequence diagram, two developers said that they felt that they finished faster thanks to the UML (row s) and two said that they finished slower because of the UML (row t). One of the two who said that the UML slowed him down was the developer who got discouraged when the UML did not help him as much as he thought it would on the first task. One developer said that he was "at least as efficient and [the UML documentation would] make people coming later on

the project more efficient. Could cut the time [that it takes to catch up] in half."

Last, while only one of the subjects (row u) in the UML group had extensive experience in working with UML (worked in a similar manner to what was asked of the subjects in this experiment), the remaining subjects had training that can be considered representative of what most practitioners who qualify themselves as experienced in UML-based developers have. Thus, nine of the subjects were learning to use the UML in such a comprehensive manner and none of the 10 had prior experience with the adopted UML tool. During the interviews, all subjects agreed that the one day training that they received at the start of the experiment provided all of the necessary information needed to use the tool, but they all also said that they would have liked to have more training (to various extents). Thus, the training that they received was adequate (row v).

3.4 Discussion of Qualitative Results

The qualitative results reveal the differences in experiences that the subjects had with the BESTweb system. Furthermore, subjects from the UML group provided insight into how they used UML and their opinions of working in such an environment. This section will look at these two issues using the presentation format from the previous section, starting with a discussion comparing the two groups, followed by a discussion on the no-UML group, and concluding with a discussion pertinent to the UML group.

The first large difference between the two groups is the UML group's seeming disadvantage (to the no-UML group) in terms of Struts experience and comfort with the Java language (rows g and i in Table 11). More subjects in the UML group claimed to have serious problems with Struts than did the no-UML subjects (eight versus five). This is significant as the BESTweb system is based on the Struts framework and this is a framework that has a productivity threshold. Thus, this could have negatively affected the overall performance of the UML group. Yet, we recruited people with the required background, so this was surprising. Taking a look at Task 3 provides some insight into this result since this was the first Struts-heavy task. The quantitative results do not point to the UML group having more problems. Thus, the UML may have helped them deal with lower Struts experience. Furthermore, the GUI shows that an equal number of subjects in each group are expressing problems with GUI development in Struts/JSP. This is further proof that the groups were probably well matched, despite the self-analysis of their Struts expertise. Furthermore, three subjects in the UML group reported that they were rusty in Java. This is a serious issue as it means that the subjects' efforts were not solely focused on solving the tasks. All of these observations suggest that our estimate of the impact of UML is probably a conservative lower bound.

Next, in terms of the subjects' perception of the system, the no-UML subjects had a higher opinion of the system in terms of the system's quality and documentation (rows a to f in Table 11). The most plausible explanation for this may be that the UML documentation allows one to see more problems and, therefore, the UML subjects were more critical. Also, more UML developers complained of poor naming of variables and classes in the system than no-UML developers, three versus one (row l in Table 11). The UML developers said that when they were looking at the sequence diagram, a poorly named variable or class made the diagram much harder to understand, forcing them to go to the code. This may have been less of an issue with the no-UML developers, as they were already in the code and could immediately look at associated comments to get the explanation behind the name (the system was completely documented with Javadoc comments).

With respect to system comprehension, one specific part of the system, namely, the BCM, caused particular problems (row j in Table 11). The no-UML subjects reported having more problems in understanding this portion of the system. This may indicate that the UML documentation aided in understanding this complex part of the system. Additionally, the BCM is one of the parts of the system that made Task 4 complex (as discussed in Section 2.6.4). The fact that the UML group took longer to complete Task 4, even without including the time that they used on updating the UML, may point to the fact that they took the time to understand that portion of the system, resulting in fewer erroneous submissions.

The key discussion point unique to the no-UML subjects was the lack of models in the documentation. This discussion varied greatly as each subject's opinion was largely influenced by their previous experience in using models (if any) and was completely out of our control. Keeping these points in mind, the qualitative results reveal that half of the subjects reported missing some kind of a model representation of the system (e.g., the domain model; see row a in Table 12) and two of these ended up drawing their own UML diagrams (both drew class diagrams and one also drew sequence diagrams) for the more complex parts of the system (row b in Table 12). Also, half of the subjects reported having problems in gaining an overview of functionality (row c in Table 12), three of which also reported missing models (row a in Table 12).

The UML subjects all found the UML to be generally useful (row f in Table 13), even though they used the UML to various extents (at the very least, they had to update it). Curiously, even though they all found it useful, only one subject used the UML to its fullest extent: the subject did not try to minimize the amount of time spent on the UML and made use of all of the diagram types and used the UML artifacts for navigating and locating code-change places. This varying use of UML among the subjects will now be examined, first in general terms and then in terms of specific types of diagrams.

Four of the subjects reported that they did not use the UML to the maximum extent out of habit (row h in Table 13). Furthermore, three of these four subjects also said that they did not take as much advantage of the UML as they could have for fear that it would take longer to solve the task in that manner (row i in Table 13). These subjects would use the UML to get an overview of the system but would then try to rely on the code to hasten the development time by, for example, checking a detail in the code rather than going back to the sequence diagram. This was unfortunate because, if the developer had an incorrect solution to the problem, either because they did not understand the specifications or because they formulated an incorrect (mental) solution, they did not give UML a chance to help them arrive at the correct solution. One of the subjects did not like to rely on the UML as he felt more confident with the code. He only used the parts of the UML that he found the most useful as opposed to trying to

take advantage of all of the UML documentation by reading and understanding it. Yet another subject spent a lot of time trying to use the UML on the first task. When he felt the UML did not help him solve the task, this discouraged him from using it to the same extent on the remaining four tasks. Thus, we can conclude that the primary reason for the lack of use of the UML in general is out of habit (or lack of experience with the UML) and fear of being inefficient. This is anecdotally confirmed by the fact that the only subject who took advantage of the UML to the full extent was the subject with extensive experience with UML (row u in Table 13).

In terms of the use of specific diagram types (rows a to e in Table 13), half of the developers took advantage of only two types of diagrams. Most subjects used the UML artifacts for navigation purposes and in order to obtain an overview of the system (rows j and g in Table 13) and this is confirmed by the fact that the use case and sequence diagrams were the most used (rows a and b in Table 13). Unfortunately, two of the developers did not use the use case diagram (row a in Table 13). This is unfortunate as all other developers that used the use case diagram reported that it was very beneficial. This is natural as it is the starting point from where the developer finds the use cases that need to be modified. These use cases are linked to the sequence diagrams that show which classes, objects, and methods are involved in the execution of the functionality specified by the use case. The class and the Struts-specific page flow diagrams (rows c and d in Table 13) were used by half of the developers and the statechart diagram (row e in Table 13) was only used by one of the developers (the one who was highly experienced at using UML). When asked why they did not take advantage of these UML diagrams, they said that either they did not feel the need or, since the other diagrams were much less integrated into the tool, it seemed troublesome to use them (not worth the effort). The fact that only half of the subjects used class diagrams was particularly surprising. While the class diagram is quite well integrated into the tool, the problem was the presence of too much irrelevant information at the same time (irrelevant for the specific task). Unlike in the case of sequence diagrams, which inherently only display the objects and interaction pertaining to a specific use case, the class diagrams display a "package view." In this view, the class diagrams display all of the classes/associations in the package. This often leads to unnecessarily complex diagrams since system learning occurs in an iterative manner. When a developer needs to modify the functionality of a use case, he/she first needs to focus only on the classes pertaining to that use case. This is very difficult when the classes are buried in a package view class diagram as they need to filter out the unnecessary parts of the diagram cognitively. One way of addressing this problem is to have a use case view of a class diagram. In this view, only the classes/associations that are used by the accompanying sequence diagram would be visible. Furthermore, class diagram views should also enable the creation of a special cluster of classes, for example, belonging to a design pattern.

The developers listed the following advantages when modeling:

- Traceability is the ability to quickly identify relevant parts of the system that need to be understood in order to implement a change (or determine the parts of the system that they needed to understand in order to solve the task). With the UML, this is accomplished by first identifying the relevant use case (on the use case diagram) and then looking at the according sequence diagram.
- Visualization and abstraction through modeling can convey information that is hard to retrieve from the code:
  1. Unnecessarily complex solutions are easily visible. For example, if a sequence diagram is very large (cannot be easily viewed on the computer screen), then it may help if the solution was simplified by decomposing it into subsystems. This, in turn, creates a more modularized and easier to understand solution.
  2. Composition is clearly seen. This helps prevent deterioration of the system's architecture (references will not be passed to objects that should not have them).
  3. All states and transitions are explicitly specified, helping the developer understand "the big picture" faster.
  4. With sequence diagrams, comments can be made on the dynamic view of the system (as opposed to the comments found in the code that only relate to the system from a static point of view).

Conversely, the developers revealed the following frustrations that they experienced:

- It was very painful and problematic to create and update sequence diagrams. This is primarily a problem with the tool and not the UML itself (row r in Table 13).
- The UML diagrams were useful for understanding parts of the system that they were unfamiliar with, but later, after gaining familiarity with the system, the developers thought that the UML was less useful overall due to the overhead of creating/keeping the UML diagrams up-to-date. Though the diagrams were still useful at determining if the solution was good, again a lot of this overhead stemmed from the developers that had to struggle with the tool.
- When a developer tried to understand the system from the UML diagrams and yet failed to do so, he/she reverted to the code. The developer then felt that time was wasted on the UML diagrams.
- The large sequence diagram and the large class diagram were overwhelming (rows n and o in Table 13). It is interesting to note that the developer who was very experienced with UML reported having trouble with the large class diagram but not with the large sequence diagram.

So, overall, we can see that the main problems faced by developers are either related to deficiencies in the tool or the need for further training and experience in using UML. This confirms further that the potential benefits of UML in the mid and long terms are probably larger than what was observed in this experiment.

Based on the discussions with the subjects, we also believe that the following items will help the adoption of the UML in practice:

- A refined UML tool that developers do not need to struggle with (row r in Table 13). The tool should enable developers to use a subset of its functionality, allowing for a gradual adaptation (see Section 6).
- A book on UML that, instead of describing the notation, draws on the best practices from experienced users of UML. This book would be analogous to the design patterns book for Object-Oriented programming [28] and the effective series book for Java and C++ [29], [30]. The book would deal with topics such as the following (raised during the training of the UML subjects):

  1. heuristics for sequence diagrams, like the maximum number of elements that can appear on the diagram [31], and
  2. when it is more appropriate to use the collaboration diagram instead of the sequence diagrams.

3.5 Summary of Results

In terms of time, the UML subjects used more time if the UML documentation was to be updated (though slightly less if it were not). With the total time T that the subjects spent on the five tasks, we see that the UML group completed the tasks slightly faster (1.4 percent) than the no-UML group (Table 7). This difference is not practically or statistically significant. When we take the time that it takes to update the UML documentation into account, we see that the UML group spent 14.5 percent more time on the five tasks, though this difference is not statistically significant either and may therefore be due to chance. On average, the UML subjects spent 14.8 percent of the total time reading the UML documentation and 13.2 percent updating the documentation.

UML was always beneficial in terms of functional correctness (introducing fewer faults into the software). The subjects in the UML group had, on average, a practically and statistically significant 54 percent increase in the functional correctness of changes $(\mathrm{p} = 0.03)$. UML also helped produce code of better quality when the developers were not yet familiar with the system. A significant difference was found for Task 1, where the UML group's design quality score was 56.2 percent higher $(\mathrm{p} = 0.0025)$, though, across all the tasks, there was an insignificant 7 percent improvement in design quality $(\mathrm{p} = 0.22)$.

All of the qualitative evidence suggests that the observed impact of UML on change quality and productivity is probably very conservative in this experiment. The UML subjects were at a disadvantage when it came to Struts experience and familiarity with Java. We also observed that half of the subjects only used two diagram types, with the use case and sequence diagrams being, by far, the most used. Four of the subjects did not use the UML to the extent that they could have due to concern that UML would make them less efficient and out of habit (not being used to using UML). The subjects also experienced severe problems when dealing with the tool and in understanding the large sequence and class diagrams. However, the qualitative evidence also explains the observed benefits of UML. The no-UML group had more problems in understanding a complex part of the system. All subjects found the UML to be generally useful: The largest benefits were the traceability of use cases to code and the ability to quickly get an overview of the system.

The results of this experiment, both qualitative and quantitative, can also be used to guide industrial adoption with respect to, at the very least, applications with similar properties (e.g., Web applications). In the case of developers who are not very experienced in using UML and who will perform maintenance tasks on a system that they are not familiar with, the use case diagram and the sequence diagrams seem to be the most cost-efficient parts of UML. This appears to be the case for two reasons. First, developers inexperienced with UML are overwhelmed by too many diagram types and will only use those that are easy to use. Next, these two diagrams help them quickly identify the relevant code for the specific functionality needed to perform the maintenance tasks. Given these advantages, these two types of diagrams can also be considered a cost-efficient starting point for introducing UML into the organization.

4 Threats to Validity

The reported experiment is generally very realistic and, in particular, when compared to previously reported experiments on UML. In fact, the main strength of this experiment lies in its external validity: Professionals worked on a real system, used real tools, and implemented real tasks. Furthermore, the fact that the developers worked until the tasks were implemented correctly ensures that this experiment does not suffer from construct validity problems with respect to correctness: How do you include unfinished or incorrect solutions in your analysis?

The hypotheses were formulated in such a way that the results obtained could be generalized to a target population of professional Java consultants performing real programming tasks with professional development tools in a realistic work setting. However, this is an ambitious goal, one that is difficult to achieve. For example, there is a trade-off between ensuring realism (to reduce threats to external validity) and ensuring control (to reduce threats to internal validity). This section discusses what we consider to be the most important threats to the validity of this experiment and offers suggestions for improvements in future experiments.

4.1 Statistical Conclusion Validity

Validity of statistical conclusions concerns 1) whether the presumed cause and effect covary and 2) how strongly they covary. For the first of these inferences, one may incorrectly conclude that cause and effect covary when, in fact, they do not (a Type I error) or incorrectly conclude that they do not covary when, in fact, they do (a Type II error). For the second inference, one may overestimate or underestimate the magnitude of covariation and the degree of confidence that the estimate warrants [32].

Recall that the individual level of significance for the hypothesis tests was set to $\alpha = 0.05$. No significant differences were found with respect to the dependent variable time. While the effect size between the time that it took to implement the tasks without accounting for the time spent on updating the UML documentation T was negligible (1.4 percent), the effect size for the time spent on implementing the tasks in total T' was -14.5 percent. Furthermore, the effect size measure d is well below 0.5 for

both T and T' and, as explained in Section 2.8.1, this indicates a *small* treatment effect. Considering the fact that the number of subjects that we used in the experiment was determined by budget constraints, it is illuminating to note that, for us to find a statistically significant difference with 80 percent power, the UML group would have to have a mean of 45 percent larger or smaller than the no-UML group (using the UML group's variance). Thus, given the effect sizes and our sample size, we were unlikely to find a statistically significant effect.

4.2 Internal Validity

The internal validity of an experiment is "the validity of inferences about whether the observed covariation between A (the presumed treatment) and B (the presumed outcome) reflects a causal relationship from A to B as those variables were manipulated or measured" [32]. If changes in B have causes other than the manipulation in A, there is a threat to the internal validity.

The main threat to internal validity in this experiment could have been the lack of random assignment to the two treatment groups: no-UML and UML. This was not possible due to practical reasons (see Section 2.4), thus making this a quasi-experiment. Fortunately, the groups were, in every practical aspect, equivalent, as discussed in Section 2.4. Furthermore, an ANCOVA was performed to adjust for the effects of grade, degree, and experience in terms of years and LOC written, all producing no different results (although grade and degree explained the variance best). Thus, we do not consider this to be a major threat. Note that, during the debriefing interview, three subjects in the UML group reported that they felt rusty in Java (see Section 3.3). Also, three more subjects in the UML group felt that the lack of knowledge in Struts caused serious problems. So, if there is any imbalance between the UML and no-UML groups, it is to the detriment of the former.

4.3 Construct Validity

Construct validity concerns the degree to which inferences are warranted from 1) the observed persons, settings, and cause and effect operations included in a study to 2) the constructs that these instances might represent. The question is therefore whether the sampling particulars of a study can be defended as measures of general constructs [32].

In the case of this experiment, we examine three such constructs. First, to investigate the effects of UML, the subject must have actually used the UML, as discussed in Section 4.3.1. Next, one of our dependent variables deals with software quality, a concept inherently without a precise definition [20]. In Section 4.3.2, we discuss the issues that are present in our definition of design quality, a small aspect of software quality. Last, no experimental setting can show the true cost of fixing a fault. The shortcomings of measuring fault cost via the time effort that it takes the developer to correct the fault are addressed in Section 4.3.3.

4.3.1 Usage of the UML

UML has many facets [33]: the choice of diagrams that are used, the level of detail of these diagrams, and the type of tool that is used (if any). In this experiment, we use five types of diagrams at the level of detail used in [10] (use case, sequence, class, statechart, and page flow). Also, the subjects were given a state-of-the-art UML development environment along with the printed UML documentation.

To ensure at least a minimum usage of the UML, the following steps were taken: The UML subjects were given training in the UML tool, the subjects were encouraged to take advantage of the UML (and not simply to ignore it, assuming that this way, they would save time), and the UML had to be updated before the solution was accepted. Even with all of these precautions, half of the developers took advantage of only two types of diagrams (see the Appendix, which can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2008.15). This is not too surprising given that nine out of the 10 UML subjects were new to such extensive use of UML and there is a learning curve before complete use of the artifacts can be made. In fact, only the subject who was highly experienced in using UML (in such a manner) took advantage of all the diagram types. Furthermore, the tool can be improved in many ways for both ease of understanding the existing design and ease of updating the artifacts (see Section 6). Consequently, the results of this study might be a conservative measure of UML's effectiveness since the developers probably did not reach their maximum level of efficiency during the experiment. To address this threat, future experiments should consider using developers who have passed the learning curve of using UML in such an advanced manner (note that the necessary subjects with such a background could not feasibly be found at the time of this experiment). In fact, such experiments always need to select a trade-off between assessing the maximum potential benefit of UML and realistic impact based on current common skills. We chose to focus on the latter in this experiment as we feel that UML experiments with highly trained students (in UML) give insight into the former [6].

4.3.2 Design Quality

We have chosen to focus on a limited aspect of design quality, a subtopic of the broad topic on software quality. As discussed in Section 2.7, arriving at a score for the solutions' design quality involved 1) breaking down each task into subtasks, 2) specifying whether a potential solution followed the proper OO principles [10], and 3) categorizing each subject's solution according to the scheme. Although this is only a small aspect of software quality, it was chosen as it met our two criteria: 1) It could be used to compare the solutions across all of the subjects and 2) it is repeatable. While the decision regarding the type of code that follows proper OO design principle is, to some extent, subjective, the process that we used to determine if a solution is acceptable or not is repeatable as each solution clearly maps to a defined category (ensuring reliability). Furthermore, the granularity at which this analysis was performed could be finer (e.g., we could also evaluate the variable names used), but then, it would not be easy to compare the quality of the solutions across all of the subjects. The main reason for this was the fact that the developers changed the system in a substantial manner and had a lot of flexibility in the manner in which they extended the design.

4.3.3 Cost of Fixing Faults

In this experiment, the cost associated with correcting a fault was the amount of time that it took the developer to fix

the specified fault. These faults were precisely pointed out to the subjects by the experimenters; the time spent by the experimenters to find the faults was not included in T and T'. This is not realistic as, in real-world scenarios, when a user finds a fault, the costs are much greater than just the correction effort. In [7], [8], experts mostly agreed that, for severe defects, "finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase." Some of the reasons for this increase in cost are listed as follows:

- The time to find the defect increases.
- There is a cost to package and deliver the fix.
- There is a potential cost to the customer due to downtime or data corruption.
- The company may suffer in terms of damage to reputation and share price.

4.4 External Validity

The issue of external validity concerns whether a causal relationship holds 1) over variations in persons, settings, treatments, and outcomes that were in the experiment and 2) for persons, settings, treatments, and outcomes that were not in the experiment [32].

The major strength of this study is, in fact, its external validity: The subjects were experienced professional software developers from various consulting companies who worked on a real nontrivial system where they implemented real and also nontrivial change tasks by using a real development environment (IDE) during an extended period of time (compared to other empirical studies on UML). Of course, the fact that some minimal degree of control had to be exercised (thus, the situation did not ideally recreate "a normal day at the office" for the developers) was a trade-off that had to be made to strike a balance between external validity and internal validity.

The scope of this study is limited to situations in which the developers have no prior knowledge of the system to be changed and it is possible that the results do not apply to situations in which the developers are also the original designers. Also, it may be the case that the results do not apply to systems in different domains.

5 Related Work

In what follows, we contrast the results from this experiment with the results from other experiments and studies that have investigated, in the context of program comprehension and maintenance, the costs and benefits of using UML and the impact of program documentation. This chapter is divided into two sections, where a general overview of the related work is given in Section 5.1 while an additional piece of related work in which two of the authors of this paper have been involved is discussed separately in Section 5.2 due to its close relationship to the work presented above. This latter section also discusses the principles of replication in software engineering experiments and shows how our work fits in this context.

5.1 Overview

One investigation evaluating whether using UML is cost-effective in a realistic context for a large project has been conducted in the form of a qualitative case study [1]. The participants in the case study acknowledged that, despite some difficulties (e.g., the need for adequate training), there are clear benefits to be derived from using UML (e.g., traceability from functional requirements to code). Although not all aspects of that work are comparable to this one, the ones that can be compared are in accordance: All of the subjects in the UML group found the diagrams to be useful and traceability was enhanced.

Another experiment was conducted to assess the qualitative efficacy of UML diagrams in aiding program understanding [34]. Fifteen subjects whose UML expertise varied (six beginners, eight intermediate, and one expert) had to analyze a series of UML diagrams (with access to code) and complete a detailed 60-minute questionnaire concerning a hypothetical software system. Results from the experiment suggest that UML's efficacy in supporting program understanding is limited by 1) unclear specifications of syntax and semantics in some of UML's more advanced features, 2) spatial layout problems (e.g., large diagrams are not easy to read), and 3) insufficient support for representing the domain knowledge required in understanding a program. This experiment only concurs with point 2). Note that, in this experiment, a Struts UML profile was used to adequately model the domain knowledge.

Furthermore, a controlled experiment investigated how access to textual system documentation (the requirements specification, design document, test report, and user manual) helped when performing maintenance tasks [35]. The results indicated that having documentation available during system maintenance reduces the time needed to understand how maintenance tasks can be performed by approximately 20 percent. The subjects who had the documentation available also showed better understanding and a more detailed solution to how the change can be incorporated as compared to those who had only the source code available. The results also suggested that there is an interaction between the maintainer's skill (as indicated by a pretest score) and the potential benefits of the system documentation: The most skilled maintainers benefited the most from the documentation. Although this work is not directly relevant, it is still relevant as UML can be considered as a form of documentation. Our experiment does not support the claim that UML decreases the time that it takes to perform the tasks nor that the most skilled maintainers benefited the most from the UML documentation. However, our experiment confirms that the presence of additional documentation in UML form gives the developers a better understanding of the system (via better correctness results).

5.2 Replication

In software engineering, as in other sciences, no single study can fully answer a large fundamental research question. Huxley [36] notes that "... in science, as in common life, our confidence in a law is in exact proportion to the absence of variation in the result of our experimental verifications." This problem is addressed via experiment replications, that is, the repetition of an experiment where some variables may or may not vary. Replications are necessary until such a time when additional verifications carry no further power of confirmation. The question of the validity of replications is addressed in detail in [37]. The authors persuasively argue that "... given the human

component and the rich variety of software and hardware technologies, it surely is beholden on the community to perform many many such verifications." Furthermore, the authors emphasize that "only under exceptional circumstances should one-shot studies involving subjects be relied upon." This point of view is shared by Curtis [38]: "... results are far more impressive when they emerge from a program of research rather than from one-shot studies."

Subjecting theory to experimental test is a crucial scientific activity, but researchers must be sure of their results before relying on them to take action. Popper [39] noted that "We do not take even our own observations quite seriously, or accept them as scientific observation, until we have repeated and tested them." Unfortunately, although it is agreed by the scientific community that replication is a crucial aspect of the scientific method, it is not widely practiced in software engineering. A systematic review of controlled experiments in software engineering showed that only 18 percent of the experiments were replicated [40]. This problem is not unique to software engineering, as noted by Collins [41]: "For the vast majority of science, replicability is an axiom rather than a matter of practice." This is also noted by Broad and Wade [42]: "How much erroneous ... science might be turned up if replication were regularly practiced, if self-policing were a more than imaginary mechanism?"

A replication can vary along three dimensions [37]: experimental method, tasks, and subjects. A basic finding replicated over several different methods carries greater weight, as stated by Brewer and Hunter [43]: "The employment of multiple research methods adds to the strength of the evidence."

Second, experimenters must decide whether a similar or alternative task should be used. Again, a basic finding replicated over several different tasks carries greater weight. Curtis [38] stated, "When a basic finding ... can be replicated over several different tasks ..., it becomes more convincing."

Third, the subjects must be considered. Not surprisingly, a basic finding replicated over several different categories of subjects also carries greater weight. This is especially true in the context of software engineering, where skills and experience have such extensive influence on the cost-effectiveness of technologies [44], [45].

This experiment can be a considered a *differentiated* [40] replication of two previous experiments [6], henceforth referred to as the Oslo/Ottawa experiment, since the same phenomena is studied, but all three replication dimensions are changed (maximizing the weight of the replication): experimental method, task, and subject. The experimental method itself is composed of various aspects related to the ways in which the studied constructs are measured, how the impact of the treatment (i.e., the UML) is analyzed, and how the randomization of subjects is performed. These differences will now be presented in a rigorous and structured manner.

Table 14 outlines the main differences in terms of the experimental method, whereas Table 15 describes the measurement of dependent variables separately for the sake of improving the tables' legibility.

Though both studies are looking to evaluate and study the impact of using UML on software evolution, the manner in which they do this greatly differs, as shown in Tables 14 and 15. The Ottawa experiment used a larger number of subjects and thus has greater statistical conclusion validity due to increased power. However, this experiment addresses the major weakness of the Oslo/Ottawa experiment, namely, external validity, by the increase of realism. Thus, with respect to validity threats, the two studies are complementary.

Realism is increased in several ways. Professional software developers are used instead of students. The system that the developers work on is a real-world system instead of artificial small systems. The settings are real world (the developers had their own office) instead of a university laboratory setting. The tasks took much more time to implement. The UML tools used in the Oslo/Ottawa experiment, namely, TAU [46] and Visio [47], were not as sophisticated and state-of-the-art as the one used in this experiment.

As opposed to the Oslo/Ottawa experiment, the developers in this experiment did not have time constraints. The drawback was that, because of the larger cost incurred, we could not recruit as many subjects as in the Ottawa experiment. But, the absence of time constraints allowed us to ensure that the developers had to submit a functionally correct solution before being allowed to proceed to the next task. Thus, time is measured differently in the two studies. Furthermore, we could also more closely monitor that the subjects followed experimental procedures throughout the experiment (the Ottawa experiment lost 11 subjects due to the subjects not following instructions properly). In the Oslo/Ottawa experiment, solutions were not checked for their correctness before being accepted. This is a crucial difference from the previous experiment as it makes the data analysis more reliable since we did not have to deal with partially correct solutions. In the Oslo experiment (which was, in a way, a pilot for the Ottawa experiment), correctness was measured by the percentage of correctly implemented tasks. In the Ottawa experiment, correctness was measured by the number of passed functional test cases. In this experiment, correctness was measured by the number of submissions (attempts) before arriving at the functionally correct solution.

Other differences include the fact that blocking was used in the Oslo/Ottawa experiment to ensure group equality. Although no blocking was performed in this experiment (due to practical reasons), a post hoc analysis of the subjects data demonstrated that the groups were equivalent, which is a result that we were expecting, given our recruitment strategy.

In the Oslo experiment, the subjects implemented the four tasks in one day for a duration of up to eight hours. The Oslo experiment had five change tasks that were completed during five course laboratories of three hours each, one task per laboratory, spaced a week apart.

Last, the design quality evaluation in the Ottawa experiment was at a finer degree than in this experiment due to the small size of the system and change tasks by counting the number of operations, attributes that should be added, modified, or deleted based on each identified solution. No design quality evaluation was performed for the Oslo experiment.

The results of both studies are contrasted in Table 16. The results of two studies must be compared carefully due to the already discussed differences in research method and measurement of variables. For example, effort was measured

TABLE 14
Study Comparison: Experimental Method

	Oslo/Ottawa	This Experiment
Experiment Design	Between- / Within-subject	Between-subject
Randomization Method	Blocking	No-blocking
Setting	Laboratory	Office
Population	Trained Students	Senior Professionals
Allocated Time to Complete All the Tasks	8 hours / 15 hours	Unbounded
Average Time Taken to Implement All Tasks (min)	202 / 492	2113
# of Tasks	4 / 4	5
Recruitment	Call for participation & financial incentive. / Part of coursework.	Call for participation & financial incentive.
Minimal Skills	3rd/4th year software engineering students having been exposed to UML	Professional software engineers with experience in UML along with other technologies and environments
Number of Subjects	20 / 78	20
UML Tool	TAU / Visio	Borland Together
System Type	Artificial	Real-world (in actual use)
# of Classes	7 / 12	50
# of Use Cases	8 / 5	16
Lines of Java Code	338 / 293	2921
Mortality Rate	9% / 14%	0%

TABLE 15
Study Comparison: Dependent Variables

	Oslo/Ottawa	This Experiment
Time Spent on Development	Amount of time used by the subject until a solution is submitted, correct or not.	Amount of time used by the subject until a **correct** solution is submitted.
Time Spent on UML	Only time spent on updating UML documents is kept track of.	Time spent on both updating and reading UML documentation is kept track of.
Measurement of Correctness	Percentage of correct solutions. / Number of functional test cases passed.	Number of submissions before arrival at the correct solution.
Code Quality Appraisal Method	None. / The design quality of a task solution was assessed on the basis of counting the number of elements that were correctly and erroneously added, changed, and deleted, based on a pre-defined optimal solution.	A described in detail in Section 2.7, each possible solution for each subtask was rated as either *acceptable* or *unacceptable*, according to the pre-defined criteria following proper object-oriented design principles [10].

TABLE 16
Study Comparison: Results

	Oslo/Ottawa	This Experiment	Comment
Time to Solve All Tasks *Excluding* Diagram Modifications	The no-UML group finished 25% slower. / The no-UML group finished 2.9% faster.	The no-UML group finished 1.4% slower.	Consistent: No significant differences.
Time *Including* Diagram Modifications	The no-UML group finished 27% faster. / The no-UML group finished 47.6% faster.	The no-UML group finished the tasks 14.5% faster than the UML group (not statistically significant).	Consistent: no-UML groups are faster.
Time Spent on Updating UML	35% / 30%	13.2%	Probably lower in this experiment due to the more experienced subjects using a more sophisticated tool. Also, subjects had more time to get used to the tool.
Correctness	Both experiments show that, for the most complex task, the subjects who used UML documentation performed significantly better than those who did not.	The UML group had 50% to 100% fewer faults in each and every task, and 54.7% fewer faults overall.	These results are consistent as all tasks in this experiment were complex, relatively speaking.
Design Quality	None. / UML solutions were better designed in terms of *correctly changed elements* (the average statistically significant difference was 0.50 on a five-point scale). With respect to *incorrectly changed elements,* a statistically significant difference was also found (average difference = 1.34).	Overall, the UML group had 7.3% more acceptable solutions.	Differences may be due to two factors: (1) the subjects in this experiment were experienced software developers and (2) the subject had to refine the solution until it was functionally correct before it was accepted. This may have resulted into a stronger convergence of the design quality of UML and no-UML groups.

in a different way. In Oslo/Ottawa, effort was equal to time "until *a* solution was submitted," while, in this experiment, effort was equal to time until a *correct* solution was submitted.

First, in terms of effort, Oslo/Ottawa reports that "When considering only the time required to make code changes, using UML documentation helps save effort overall." This is largely consistent with the results in this experiment: On average, the UML subjects spent less time, but the results in this experiment were not significant, perhaps due to the lack of statistical power. Next, Oslo/Ottawa reports that "When including the time necessary to modify the diagrams, no savings in effort are visible." In fact, in both studies, the no-UML groups finished faster.

In terms of the time spent on updating the UML documentation, the Oslo/Ottawa experiment reports an overhead of 35 percent and 30 percent, respectively. This is higher than the overhead in this experiment (13.2 percent). We believe that this is due to this experiment having more experienced subjects, using a more sophisticated tool, for a longer duration (having had more time to get used to the tool). In terms of the time spent on updating the UML documentation, the Oslo/Ottawa experiment did not collect quantitative data but reports that "Most people [*thought* that they] spent less than 25% of the time in laboratory sessions understanding UML diagrams, over all tasks." In this experiment, on average, the subjects spent 14.8 percent of their time understanding the UML documentation (see Section 3.1.1).

In terms of correctness, in Oslo/Ottawa, "... both experiments show that, for the most complex task, the subjects who used UML documentation performed significantly better than those who did not." These results are similar: We saw significant benefits of the UML in terms of correctness.

Design quality was investigated in the Ottawa experiment, where it was found that "... using UML helped achieve changes with superior design quality, which would then be expected to facilitate future, subsequent changes." Across all of the tasks, this is inconsistent with our results, though it is consistent with the results on Task 1. This may be due to two factors: 1) The subjects in this experiment were experienced software developers and 2) they had to refine the solution

until it was functionally correct before it was accepted. This may have resulted in a stronger convergence of the design quality of the UML and the no-UML groups.

In summary, it is interesting to note that the results of the two studies show many similarities, despite the studies being very differentiated (differences in experimental method, tasks, and subjects). Thus, because we obtain similar results after using different measurements, both with trained students and with professionals, and systems of widely varying size, we can be confident that UML will bring practically significant benefits in a large number of conditions.

6 Suggested Improvements to the UML Tool

During the course of this experiment, from its design to the debriefing interviews with the subjects, ideas emerged with respect to possible improvements to the UML tool that we used, namely, BTE [9], in terms of usability, code generation, rule enforcement with respect to class and sequence diagrams, and means of gradual adoption of the UML tools into widespread use. Although these recommendations are targeted at BTE, most probably they also apply to other UML tools.

First, the subjects in the UML group expressed the need to return to the code to read comments. These comments can be made available directly on the diagram, saving the developer the need to go into the code, as that is highly distracting. One way of making these code comments available on the diagram is via tooltips. For example, on the class diagram, hovering over a class would bring up a tooltip containing its corresponding Javadoc comments. Hovering over a method in a class or an association would bring up its comment. On the sequence diagram, a tooltip with the code comment would appear when hovering over an object, a method, or a message. On use case diagrams, when hovering over a use case, a tooltip can show its description.

In the case of sequence diagrams, the following improvements to the UML tools are highly recommended:

- When scrolling down on a sequence diagram, the objects should never disappear from view.
- The updating of existing diagrams can be facilitated by 1) being able to add a message to an existing sequence diagram (in the appropriate place) directly from the code view and 2) the option of using dynamic analysis on the corresponding use case (with the appropriate filters being applied) and the (new) missing elements being automatically shown to the developer. The new additions can then be accepted or rejected from being displayed on the sequence diagram.
- It should be possible to generate a sequence diagram from dynamic analysis. Even though this is difficult to accomplish in a complete manner [48], an incomplete diagram that the developer could later refine would still be helpful.
- The developer should be able to ask the tool to display elements that are absent on the sequence diagram and then selectively choose to add elements that should appear on that sequence diagram.
- Upon selecting an object or method on the sequence diagram, the corresponding class (and method) should be highlighted in a different color on the class diagram so that the developer saves time when looking for it.

In the case of class diagrams, as discussed in Section 3.4, support for views is necessary to help developers focus on the important parts of a class diagram (a view on a class diagram would only show the classes and associations of interest). Next, in addition to the ability of visually specifying composite relationships and immutable (frozen) classes, safeguards can be put in place to ensure that these rules are not violated. Also, generation of the clone and equals method can be largely automated when this information is explicitly specified (composition and immutability). This is important as these methods are very tricky to implement correct, as discussed in [29].

In terms of additional UI support for class diagrams, a specific use of the class diagram was deemed as potentially being very useful: the possibility for an IDE to show a subset of a class diagram where the only classes that would be displayed are 1) the class of interest (e.g., being currently selected/modified) and 2) the class with which the class in 1) has immediate relations (that is, classes that exchange messages in sequence diagrams). This could also be used to gradually introduce UML into the development environment and let developers gradually adopt UML. Furthermore, this tool could also be used for visual dependency analysis. On a related note, a major inconvenience in the tool that was used in the experiment was the fact that the usage dependencies had to be specified manually, which is an unnecessary burden placed on the developers.

An investigation into a competing tool, namely, IBM's Rational Software Architect [49], revealed that it implements only one of the suggestions presented here: When scrolling down on a sequence diagram, the objects never disappear from view.

7 Conclusions

An experiment was conducted to investigate the costs and benefits associated with the UML during maintenance and evolution. This is the first such experiment performed on a real system, using professional developers as subjects and working with a state-of-the-art UML tool during an extended period of time. This paper provides very clear insights in terms of the kinds of (minimum) benefits that can be expected from using UML and the factors limiting or boosting such benefits. In turn, such information can be used to decide about whether and how UML can be introduced in a development organization. Although experiments are needed in other contexts as well (e.g., use of UML during initial development), we focused on software maintenance and evolution by a nonoriginal developer as this consumes the majority of the resources in a typical software organization.

The quantitative results show that UML did not have a significant impact on the time that it took to perform the change tasks, both excluding and including the time that it took to update the UML documentation. However, in terms of the functional correctness of the changes, UML had a practically and significantly positive impact, despite the fact that the UML subjects were not experts in UML and

encountered many problems with the modeling tool. Last, in terms of design quality, a post hoc analysis revealed a significant difference in the first task, where the UML subjects were not yet familiar with the system and delivered solutions of higher quality. However, significant differences were not observed on the remaining four tasks. The qualitative results explained the probable root causes of the observed benefits: traceability from functionality to code and an abstract overview of the system structure and functionality. It also provided evidence that the observed benefits of using UML were probably conservative and that better tools and even more experience would likely yield a larger return on investment.

In terms of related work, the results largely support those of similar experiments, especially [6]. Because we obtain similar results by using different measurements, both with trained students and professionals and systems of widely varying size, we can be confident that UML will bring practically significant benefits under a large number of conditions.

Last, even though such experiments are very costly and labor intensive (this experiment took 3 years of preparation, running, analysis, and write-up), we deem these crucial and well worth the cost and effort in order to mature the manner in which new techniques are adopted in software engineering. Usually, when researchers want to assess the effects of software engineering technologies with real tasks and professional developers, they resort to case studies for which there is much less control. Unlike industrial case studies, this experiment controls for many extraneous factors that can impact our ability to analyze the effect of UML on software maintenance. However, the conducting of controlled experiments with a high degree of realism introduces several issues that need to be tackled regarding experiment design, instrumentation and measurement, and practical execution. Thus, an additional contribution of this paper is that it serves as an in-depth example of how such controlled experiments can be conducted.

ACKNOWLEDGMENTS

The authors are grateful to Hans Christian Benestad, Magne Jørgensen, Tanja Gruschke, Marek Vokáč, and Kjetil Moløkken-Østvold for their valuable contributions to this work. They also thank the developers and organizers at the participating software companies. Last, they thank the anonymous reviewers for their insightful suggestions that improved the paper.

REFERENCES

[1] B. Anda, K. Hansen, I. Gullesen, and H.K. Thorsen, "Experiences from Using a UML-Based Development Method in a Large Organization," *Empirical Software Eng. J.*, vol. 11, pp. 555-581, 2006.
[2] K. Beck, *Extreme Programming Explained.* Addison-Wesley, 2000.
[3] R.S. Pressman, *Software Engineering: A Practitioner's Approach,* seventh ed. McGraw Hill, 2005.
[4] R. Glass, *Facts and Fallacies of Software Engineering.* Addison-Wesley, 2002.
[5] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model-Driven Architecture—Practice and Promise.* Addison-Wesley, 2003.
[6] E. Arisholm, L.C. Briand, S.E. Hove, and Y. Labiche, "The Impact of UML Documentation on Software Maintenance: An Experimental Evaluation," *IEEE Trans. Software Eng.*, vol. 32, pp. 365-381, 2006.
[7] T. McGibbon, "Software Reliability Data Summary," Data Analysis Center for Software, 1996.
[8] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned about Fighting Defects," *Proc. Eighth IEEE Int'l Symp. Software Metrics,* 2002.
[9] "Borland Together for Eclipse," Borland, 2004.
[10] B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java,* second ed. Prentice Hall, 2004.
[11] J. Holmes, *Struts: The Complete Reference.* McGraw-Hill, 2004.
[12] "JavaServer Pages 2.0 Specification," Sun Microsystems, 2003.
[13] J. Gosling, *The Java Language Specification,* second ed. Addison-Wesley, 2000.
[14] T.A. Powell, *HTML: The Complete Reference,* third ed. Osborne/McGraw-Hill, 2001.
[15] "Eclipse," IBM, 2004.
[16] M. Kofler, *MySQL.* Apress, 2001.
[17] "BESTweb," Simula Research Laboratory, http://simula.no/BESTweb/, 2004.
[18] M. Jørgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 33-53, Jan. 2007.
[19] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000.
[20] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* John Wiley & Sons/IEEE CS Press, 2005.
[21] J.L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists.* Duxbury, 1999.
[22] L.V. Garcia, "Escaping the Bonferroni Iron Claw in Ecological Studies," *Oikos,* vol. 105, pp. 657-663, 2004.
[23] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences,* second ed. L. Erlbaum Assoc., 1988.
[24] R.H. Myers, D.C. Montgomery, and G.G. Vining, *Generalized Linear Models: With Applications in Engineering and the Sciences.* J. Wiley, 2002.
[25] S.E. Hove and B. Anda, "Experiences from Conducting Semi-Structured Interviews in Empirical Software Engineering Research," *Proc. 11th IEEE Int'l Symp. Software Metrics,* 2005.
[26] O.R. Holsti, *Content Analysis for the Social Sciences and Humanities.* Addison-Wesley, 1969.
[27] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Trans. Software Eng.*, vol. 27, pp. 1-12, 2001.
[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[29] J. Bloch, *Effective Java Programming Language Guide.* Prentice Hall, 2001.
[30] S. Meyers, *Effective C++,* second ed. Addison-Wesley, 1997.
[31] G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Rev.*, vol. 63, pp. 81-97, 1956.
[32] W.R. Shadish, T.D. Cook, and D.T. Campbell, *Experimental and Quasi-Experimental Designs for Generalized Causal Inference.* Houghton Mifflin, 2002.
[33] M. Fowler and K. Scott, *UML Distilled: A Brief Guide to the Standard Object Modeling Language,* second ed. Addison-Wesley, 2000.
[34] S. Tilley and S. Huang, "A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding," *Proc. ACM SIGDOC '03,* pp. 184-191, 2003.
[35] E. Tryggeseth, "Report from an Experiment: Impact of Documentation on Maintenance," *Empirical Software Eng. J.*, vol. 2, pp. 201-207, 1997.
[36] T.H. Huxley, "We are All Scientists," *The New Treasury of Science,* H. Shapley, S. Rapport, and H. Wright, eds., Collins, 1965.
[37] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller, "Replication's Role in Software Engineering," *Guide to Advanced Empirical Software Eng.*, F. Schull, J. Singer, and D. Sjoberg, eds., pp. 365-379, Springer Science, 2008.
[38] B. Curtis, "Measurement and Experimentation in Software Engineering," *Proc. IEEE,* vol. 68, pp. 1144-1157, 1980.
[39] K.R. Popper, *The Logic of Scientific Discovery.* Hutchinson, 1968.

[40] D.I.K. Sjøberg, J.E. Hannay, O. Hansen, V.B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A.C. Rekdal, "A Survey of Controlled Experiments in Software Engineering," *IEEE Trans. Software Eng.,* vol. 31, pp. 1-21, 2005.
[41] H.M. Collins, *Changing Order Replication and Induction in Scientific Practice.* Sage Publications, 1985.
[42] W. Broad and N. Wade, *Betrayers of the Truth.* Oxford Univ. Press, 1986.
[43] J. Brewer and A. Hunter, *Multimethod Research: A Synthesis of Styles.* Sage Publications, 1989.
[44] E. Arisholm and D.I.K. Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 30, pp. 521-534, 2004.
[45] E. Arisholm, H.E. Gallis, T. Dybå, and D.I.K. Sjøberg, "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise," *IEEE Trans. Software Eng.,* vol. 33, pp. 65-86, 2007.
[46] "TAU," Telelogic, 2003.
[47] "Visio," Microsoft, 2002.
[48] L.C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Trans. Software Eng.,* vol. 32, pp. 642-663, 2006.
[49] "Rational Software Architect," IBM, 2007.

Wojciech James Dzidek received the BEng degree in computer systems and the MEng degree in software engineering from Carleton University, Ottawa. He is currently working toward the PhD degree in software engineering at the Simula Research Laboratory, Oslo. His PhD dissertation is focused on the empirical evaluation of the costs and benefits of UML in software maintenance. His research interests include model-driven development, testing and quality assurance, and empirical software engineering. He is a student member of the IEEE.

Erik Arisholm received the MSc degree in electrical engineering from the University of Toronto and the PhD degree in computer science from the University of Oslo. He has seven years of industrial experience in Canada and Norway as a lead engineer and a design manager. He currently heads a research project on software maintenance in the Department of Software Engineering at the Simula Research Laboratory and is an associate professor in the Department of Informatics at the University of Oslo. His research interests include empirical studies of maintainability, object-oriented analysis and design, and software quality measurement and prediction. He is a member of the IEEE and the IEEE Computer Society.

Lionel C. Briand is a professor of software engineering at the Simula Research Laboratory and the University of Oslo. He is currently on leave from the Department of Systems and Computer Engineering at Carleton University, Ottawa, where he is a full professor and the Canada Research Chair in Software Quality Engineering. Before that, he was the head of the Department of Software Quality Engineering at the Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany. He was also a research scientist in the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE, and ACM conferences. He is a co-editor in chief of *Empirical Software Engineering* (Springer) and is a member of the editorial boards of *Systems and Software Modeling* (Springer) and *Software Testing, Verification, and Reliability* (Wiley). From 2000 to 2004, he was on the editorial board of the *IEEE Transactions on Software Engineering*. His research interests include model-driven development, testing and quality assurance, and empirical software engineering. He is a senior member of the IEEE and a member of the IEEE Computer Society.