Empir Software Eng (2006) 11: 227–267
DOI 10.1007/s10664-006-6403-7

Using patterns for the refinement and translation of UML models: A controlled experiment

Christian Bunse

Editor: Murray Wood

Abstract This paper presents a controlled experiment, conducted at the University of Kaiserslautern which evaluates an approach known as SORT, for the systematic refinement and translation of UML Diagrams. Specifically, the experiment investigates the effects of SORT, with respect to the mapping of object-oriented UML design models to source code, by comparing the effects of different approaches to such mappings (SORT and ad-hoc[1]) on the quality attributes understandability, verifiability, and effort (time). The experimental results demonstrate that OO systems developed by applying SORT are more understandable and verifiable. In summary, SORT can help to improve the quality of software systems, but its application alone does not guarantee quality.

Keywords Empirical evaluation · UML · Patterns · Understandability · Verifiability

1. Introduction

The success of the Unified Modeling Language (OMG, 2001) (OMG, 2005) reflects a growing consensus in the software industry that "modeling is a good thing." However, models created in the earlier phases of development (e.g., analysis and design) are of limited value, unless they can be readily mapped into correct and efficient executable forms, which in today's technology means code in high-level object-oriented programming languages. Any problem in the transformation path from models to code does not only have a negative impact on the quality of the

C. Bunse (✉)
Fraunhofer Institute for Experimental Software Engineering,
Fraunhoferplatz 1, 67663 Kaiserslautern, Germany
e-mail: Christian.Bunse@iese.fraunhofer.de

[1] The mapping strategy followed by most mainstream methods is based on personal experience and general mapping guidelines. Thus, mappings are performed on an individual basis without systematic guidance (i.e., these approaches can be characterized as "ad-hoc" mapping approaches).

delivered software system, but also hinders its future maintenance and/or reuse. Furthermore, this reinforces the widely held suspicion that modeling is "just paperwork" without any serious connection to the "real" business of code generation.

Ironically, the very richness and generality of UML is something of an "Achilles heel" in this regard. This is because power without control is dangerous. Developers attempting to use UML have such a wide selection of modeling concepts at their disposal, ranging from low-level implementation oriented concepts to very high-level abstract concepts, that they can easily lose their way and end up facing a daunting gap to span in order to translate their models into code. Not surprisingly, the wider the semantic gap to be bridged, the greater the chance of mappings that are inadequate or incorrect. It is not UML per se that is really responsible for addressing this problem, but rather the methods that are intended to support it. However, few, if any of the current UML-oriented methods pay much attention to this issue. The books that define the leading methods include at best a chapter discussing "implementation issues," usually in a general and ad hoc way (Rumbaugh et al., 1991; Booch, 1994; Kruchten, 1998; D'Souza and Wills, 1998; Cheesman and Daniels, 2000). Similarly, books on computer languages rarely spend more than a chapter on discussing how features of the language relate to modeling concepts such as those in UML (Stroustrup, 1993; DoD, 1983; Meyer, 1992). Newer publications such as Reed (2001) or Bruegge and Dutoit (2003) have recognized the importance and therefore address the mapping from models to code explicitly. Although, this goes into the right direction, the discussed mappings are either straightforward (mapping a UML class to a Java class) or do not address the context of a system (e.g., non-functional aspects) explicitly. As a result, the mapping of graphical models into code is an often neglected link in modern software development processes.

Perhaps one reason for this is the widely held view that case tools have already solved this problem. Widely advertised capabilities such as "round-trip engineering" give the impression that at the press of a button a case tool can translate a rich UML model into a complete, optimal, executable program (Medvidovic et al., 1999). However, although the code creation capabilities of modern case tools can be helpful when used appropriately, no tool is yet capable of handling all the nuances and trade-offs involved in creating an optimal and efficient implementation of UML models (Jeckle, 2005; Sendall and Küster, 2004; Holub, 2002). The "one-size-fits-all" mapping schemes[2] found in most case tools usually end up being incomplete and/or suboptimal for most situations. Some of these problems are also based on the insufficient semantics of UML. In domains such as automotive or aviation, using more "formal" modeling languages, such as SDL or ASCET, success stories of automatic code generation are known. However, code generation from UML models is not that successful. Thus, automatic code generation cannot be the solution for all problems (Herrington, 2003): "*You have first to study very well your project before you decide whether it makes sense to resort to code generation to produce any part of the project code*."

These problems all point to the need for a well-defined and flexible methodology for supporting the mapping of UML models into executable code in a way that takes into account prevailing non-functional requirements. One such approach, known as

[2] I.e., there is only one mapping rule for a given UML construct.

SORT (Atkinson, 2001; Bunse and Atkinson, 1999a,b), is based on the principle of cleanly decoupling refinement from translation by first refining it into a more concrete form at the implementation level of abstraction and second, translating the refined model into a tool comprehensible form (e.g., source code) with the help of patterns (i.e., pre-verified mappings). SORT promises to reduce the required effort for mappings and to have a positive impact on the understandability and verifiability of the implementation. However, an empirical validation is needed to objectively check whether this is true.

This paper describes the empirical evaluation of the SORT approach. After first outlining the principles underlying the approach under evaluation in Section 2, the remaining sections describe its empirical evaluation. Section 3 presents the details of the experiment by discussing hypotheses, experimental design, subjects, and materials. Section 3 summarizes the data collected and presents the data analysis results. Section 4 identifies and discusses possible threats to the validity of the experiment. Finally, a short summary and some conclusions are presented.

2. Pattern-Based Implementation of UML Models

2.1. General Problem

The general goal of the implementation phase of any object-oriented development method is to translate an abstract and informal representation of an artifact (commonly referred to as a design) into a more concrete representation that can be processed by automatic tools such as a compiler.

Most object-oriented methods provide little if any guidance on how to perform the implementation activity. Moreover, any guidance that is given almost always tries to bridge the gap between the abstract, informal representation and the more concrete, "formal" representation in one big step by directly mapping high-level design models, and even analysis models, into code with or without the help of CASE-tools (Webster, 1995; Bunse and Atkinson, 1999a,b; Bunse, 2001). Developers trying to perform such transformations therefore have to simultaneously concern themselves with different representation forms (e.g., UML models and source code) and different abstraction levels (i.e., analysis, design, and implementation). As a consequence, the implementation is typically one of the weakest aspects of many contemporary methods, and the source of many defects (Holub, 2002).

In practice, the step from an informal representation to a more concrete one (i.e., from design to code) involves two distinct transformations: *refinement* and *translation*. The first, refinement, is the description of a given phenomenon at a lower level of abstraction. In the context of UML, a refinement step takes one arrangement of pieces and rearranges or expands it to create another group connected via refinement relationships (D'Souza and Wills, 1998). The second transformation, translation, is the description of a given artifact in a different form (e.g., from implementation models to Java or C++ source code). In contrast to refinement, translation describes a relation between two descriptions at the same level of abstraction. Both descriptions are of the same thing, and contain the same amount of detail, but they use different languages or representation forms.

If refinement and translation are mixed, the relationship between a model and its executable description is complex, not intuitively understandable, and potentially misleading. For example, the implementation of a simple UML association between two classes does not only include a change of notation (from UML to source code), but also many implicit decisions (i.e., an association may be implemented by a reference via an array of pointers, by a dictionary object, etc.). Therefore, a developer implementing such an association has to make several, possibly undocumented, decisions, which make the link not only hard to understand but also difficult to maintain.

2.2. Basic Principles

Separating refinement and translation reduces the likelihood of errors during the implementation process, since individual steps are smaller and only address one dimension of concern (i.e., either abstraction level or representation form). Following this principle, the SORT approach (Bunse and Atkinson, 1999a) cleanly decouples refinement and translation by first refining UML models into a more concrete form at the implementation level of abstraction and second, translating the refined models into a tool comprehensible form (e.g., source code). This requires defining to what level of detail refinement proceeds until translation can start. The basic idea is to define a set of UML modeling concepts which correspond to the core constructs of object-oriented programming so that they can be mapped into elements of a program in a manner that approximates translation (i.e., without a significant change in abstraction level). This set is called the Normal Object Form, or NOF (Bunse and Atkinson, 1999b), because in a sense it represents a "normal" form, akin to that used in relational databases, to which UML models must be reduced before translation can begin.

The word "set" was chosen carefully in the preceding paragraph. At first sight it might appear that a subset of the existing UML modeling concepts would best satisfy the goal identified above, but on closer analysis this turns out not to be so. The UML certainly contains many low-level features which have a very close correspondence to core object-oriented language concepts, such as classes, methods, packages etc. However, there are also numerous fundamental object-oriented programming features which are not represent directly within the UML. Therefore, in defining the NOF it was also found necessary to add additional concepts to those predefined in the UML, using UML's in built extension mechanisms (i.e., stereotypes, tagged values and constraints). More precisely, the definition of the NOF consists of three distinct parts: (1) a subset of the predefined UML modeling features; (2) additional modeling features, defined through the UML extension mechanism, and (3) constraints on the use of (1) and (2). Strictly speaking this makes the NOF a restricted extension of the UML. A brief summary of NOF elements can be found in Appendix A.

2.3. Patterns

In order to be of practical value the NOF needs appropriate methodological support. This is the goal of SORT (Systematic Object-Oriented Refinement and Translation). SORT provides a practical technique for leveraging the NOF, and the concept of refinement/translation separation, by packaging useful refinements and translations.

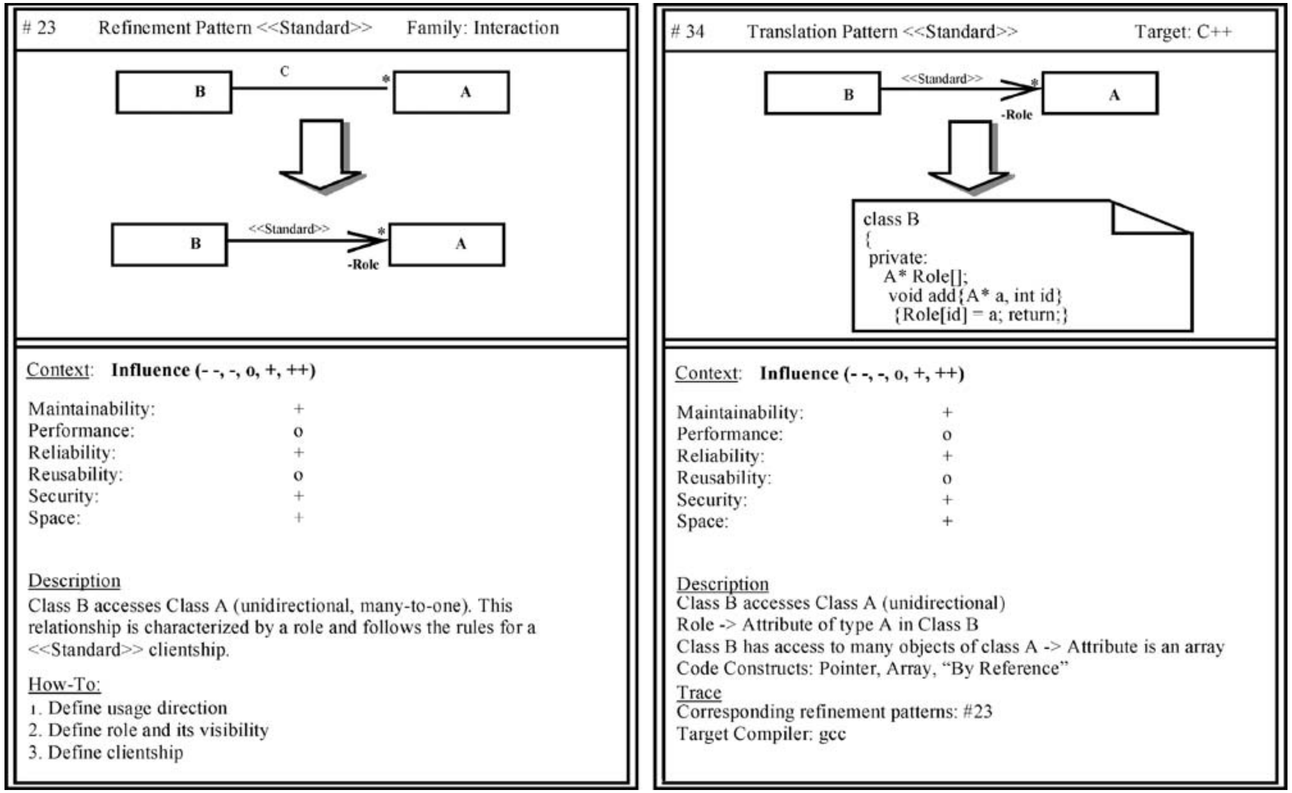
In view of the success of the pattern cataloguing approaches pioneered by Gamma et al., (1995) and Buschmann (Martin et al., 1997), SORT refinement and translation guidelines are packaged in a similar style. However, there is a subtle difference between the patterns defined in SORT and those of Gamma and Buschmann. Whereas the latter essentially capture good (i.e., useful) object-oriented structures/behaviors, SORT patterns capture good (i.e., useful) mappings between object-oriented structures/behaviors. Two forms of patterns are recognized in SORT: *refinement patterns*, which describe "good" refinements within the UML, and *translation patterns*, which describe "good" mappings of refined models to a specific object-oriented programming language (e.g., C++). The latter are similar to "idioms" (Coplien and Schmidt, 1995) in that they are language specific.

Refinement patterns define sound refinements from higher-level (i.e., non-NOF) UML model elements to UML-NOF elements. In general such a pattern defines how to remove higher-level elements, how to use NOF specific elements in NOF diagrams, and how to enforce the NOF modeling. To keep track (traceability) of refinements, the refinement relationships between elements must be made explicit by using the UML stereotype ≪refinement≫. This enables developers to traverse the refinement tree in order to identify those places where further work is needed. Translation patterns perform a similar job to refinement patterns, but for translations instead of refinements. Although the two are strictly separated, however, the relationship between the leaves of the refinement tree and the associated programming constructs must be clear. In other words, translation patterns are not solely oriented towards implementing a NOF modeling element, but also towards continuing the process of developing code from models seamlessly. An example of each of the two pattern forms can be found in Fig. 1.

However, simply defining the mapping is not sufficient. If there is more than one pattern for a specific construct it must be possible to choose the pattern which is most suitable for a given context. An essential part of a SORT pattern is therefore a list of the primary quality factors which the transformation influence (e.g., performance) and the level of this influence. For example, a transformation that implements a model element in a way that minimizes space may well have a detrimental influence on the final system's performance, whereas a refinement that maximizes performance may have a detrimental influence on space usage. Providing quantitative measures, on the influence a pattern has on quality factors, can be potentially misleading. On the one hand they allow patterns to be chosen on an objective basis, but on the other hand such measures are difficult, if not impossible, to collect and may rely heavily on the definition of the underlying metric (Grady, 1992; Satpathy et al., 2000). Therefore SORT adopts a qualitative scale which provides a subjective influence estimate. More detailed information on SORT and its patterns can be found in Atkinson et al., (2001).

3. Description of the Experiment

In general, empirical studies in software engineering are used to evaluate whether a "new" technique is superior to other techniques concerning a specific problem or property. This paper addresses the problem of mapping UML models to code. The benefits of a systematic mapping technique, such as SORT, are defined by a couple of



23 Refinement Pattern <<Standard>> Family: Interaction

Context: **Influence (- -, -, o, +, ++)**

Maintainability:	+
Performance:	o
Reliability:	+
Reusability:	o
Security:	+
Space:	+

Description
Class B accesses Class A (unidirectional, many-to-one). This relationship is characterized by a role and follows the rules for a <<Standard>> clientship.

How-To:
1. Define usage direction
2. Define role and its visibility
3. Define clientship

34 Translation Pattern <<Standard>> Target: C++

```
class B
{
 private:
   A* Role[];
    void add{A* a, int id}
     {Role[id] = a; return;}
```

Context: **Influence (- -, -, o, +, ++)**

Maintainability:	+
Performance:	o
Reliability:	+
Reusability:	o
Security:	+
Space:	+

Description
Class B accesses Class A (unidirectional)
Role -> Attribute of type A in Class B
Class B has access to many objects of class A -> Attribute is an array
Code Constructs: Pointer, Array, "By Reference"
Trace
Corresponding refinement patterns: #23
Target Compiler: gcc

Fig. 1 Refinement & translation pattern example

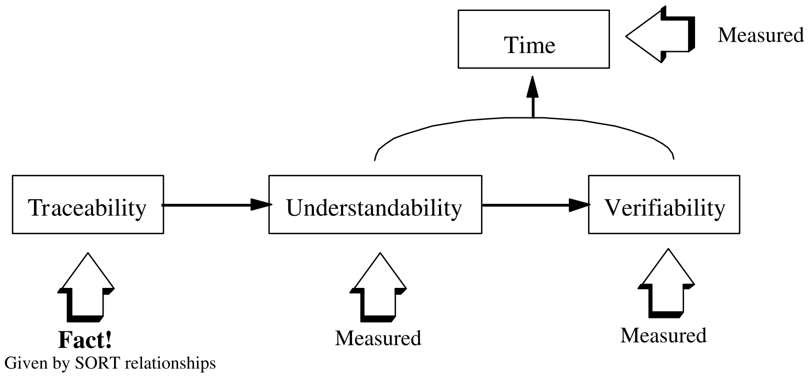


Fig. 2 Experimental model

research hypotheses. In detail, it is expected that such a technique would establish traceability between different representations of a system. The sequence of applied refinements and translations establishes relationships between model elements and between model elements and source code. Thus, these relationships have a positive impact on the quality of a system: They allow better understanding of the system, in such that the dependencies between models and code are known, and that specific non-functional requirements[3] are addressed. They also have a positive impact on the quality of the system in terms of contained defects and fulfillment of non-functional requirements, since patterns are pre-verified mappings (i.e., the generic mapping described by a pattern was checked for correctness and quality before it becomes a pattern). Finally, the guidance of such an approach will reduce development time. However, the approach does not guarantee correctness since it (1) depends on the quality of the original models, and (2) assumes that the correct pattern is applied. However, despite these limitations it is assumed that making refinements and translations explicit will have a positive impact.

The evaluation of SORT is based on a controlled experiment that investigates whether SORT truly fulfills the postulated benefits. The experiment compares two existing system documentations (i.e., SORT and non-SORT) and assumes that the application of SORT has established explicit links between a design and its implementation (i.e., traceability). Based on these relationships, the effects of different mapping techniques, SORT and Ad-hoc,[4] in terms of understandability, verifiability, and time are measured by letting the experiment's participants perform understanding and verification tasks on the design and its implementation. This is illustrated in Fig. 2. However, it must be explicitly stated that the results of this experiment cannot empirically evaluate the effects of SORT in full detail. Thus, the

[3] SORT patterns address the non-functional aspects of maintainability, performance, reliability, reusability, security, and memory, by providing an estimate of the impact of applying the pattern on the specific non-functional aspect.

[4] In this experiment SORT mappings are compared with mappings done in an Ad-hoc manner.The term Ad-hoc reflects "less structured" mappings of models to code which intermix refinement and translation steps. In Ad-hoc Mappings, models are either used as a reference for developers during implementation or are used to generate code-frames (neglecting non implementable modeling constructs such as association classes) that are manually completed.

results have to be further evaluated and extended by a series of additional experiments (Green et al., 1990).

3.1. Hypotheses

The concepts of understandability, verifiability, and time are of major interest within this study. As with many other concepts in software engineering, the first two are difficult to measure fully. In this study, understanding is captured by means of asking questions about the system documentation. Verifiability is captured by means of subjects performing verification activities on the design and code documents. Time is then simply captured in minutes for both activities. Standard significance testing was used to clearly examine these effects, the null hypothesis being stated as:

H_0 – There is no difference between SORT and ad-hoc developed OO systems in terms of their understandability and verifiability.

The alternative hypotheses, i.e., what is expected to occur, are then stated as

H_1 – Software systems developed by SORT permit "easier" understanding (i.e., a quicker and more thorough knowledge of a system) than software systems developed by ad-hoc approaches (i.e., non-SORT).

H_2 – The documents and techniques derived using SORT permit more efficient and more effective identification of defects (correctness and fulfillment of nonfunctional requirements) than ad-hoc approaches.

H_1 and H_2 are stated on the following basis: When SORT principles are applied they will improve the quality of a system in terms of increased understandability and verifiability.

3.2. Subjects

The experimental subjects used in the study were all master students[5] from the Department of Computer Science at the University of Kaiserslautern, who were enrolled in a one semester introductory class on Software Engineering. During the lectures, subjects were taught basic software engineering principles, object-oriented development techniques, UML, and basic programming skills. The lectures were supplemented by practical sessions in which the students had the opportunity to make use of what they had learned through completion of various software development exercises (Leite, 2000).

At the beginning of the course, subjects were informed that a series of practical exercises ("experiments") were planned and were asked to participate. Students were motivated by making it clear that they would gain valuable experience from participating during the subsequent sessions. As a result, 27 out of 30 subjects volunteered to take part. The subjects knew that data would be collected and that an analysis would be performed on the data, but were unaware of the experimental hypotheses that were being tested.

[5] All students already had passed their initial exams ("Vordiplom") and can thus be regarded as master students.

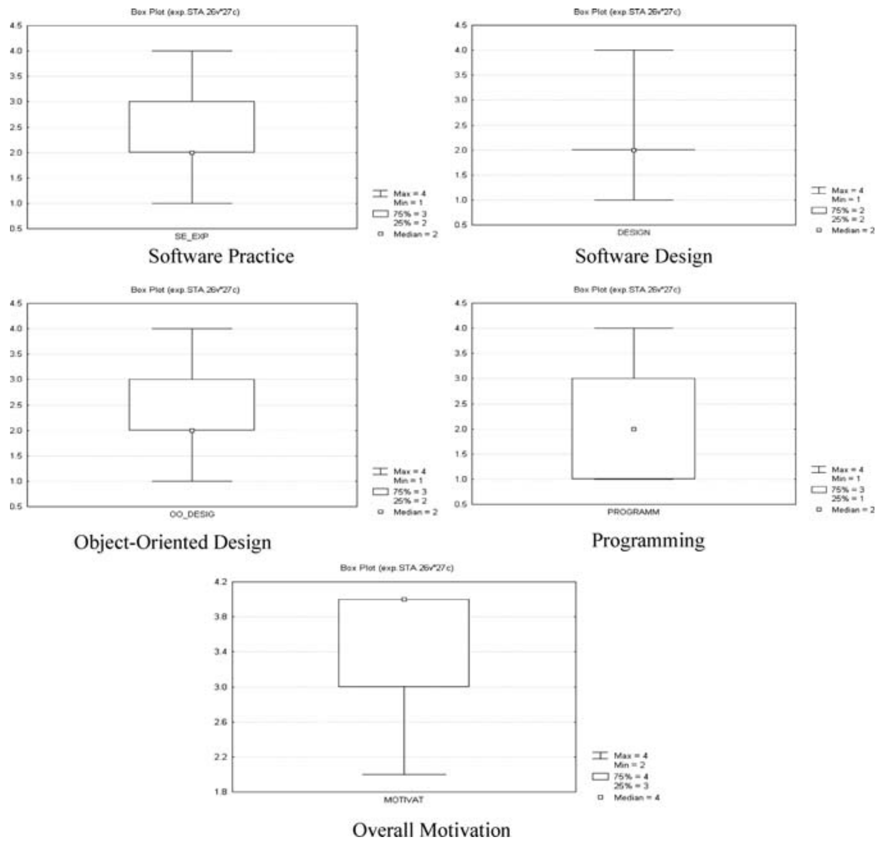


Fig. 3 Box–whisker plots—subjects' experience

As the German system allows students to take different classes at different times during their studies, the students were of varying degrees of experience (see Fig. 3). Before attending SE-1, the subjects had little experience[6] with (a) software practice–2 median response, (b) design documents–2 median response, (c) object-oriented design documents–2 median response, and (d) implementing object-oriented designs in C++–2 median response. However, the subjects were fairly motivated to participate (median 4 from an ordinal scale of 1—not motivated to 5—highly motivated).

3.3. Materials

To test the hypotheses, two different object-oriented systems were used. One system, in the following referred to as the "*Ad-hoc*" system, was taken from an OO textbook (Lee and Tepfenhart, 1997), which proposes a "simple" UML-based

[6] This information was captured by means of the debriefing questionnaire based on the ordinal scale of 1–no experience to 5–professional experience. See also Appendix E.

Table 1 Descriptive measures for each experiment system

Counts	Original SORT system	Refined SORT system	Ad-hoc system
Classes (Library classes[a])	19 (4)	14 (1)[b]	18 (3)
Attributes (total)	42	26	28
Methods (total)	37	28	23
Inheritance Relationships	11	9	10
Associations	2	2	3
Aggregations	7	4	7
CBO	1.63	1.36	1.66
NOC	0.57	0.65	0.61
DIT	1.1	1.1	1.1

[a] Library classes are predefined, ready-to-use classes from outside the system. Within the documentation, library classes are only described by an interface specification and a short, natural language description.

[b] The application of SORT resulted in several classes being removed from the original system. SORT patterns also use refactoring ideas to simplify a design such as those described in (Astels, 2002).

development process and ad-hoc mappings of models to code. The other system, in the following referred to as the "SORT" system, was developed for the experiment and documented in the same way as the Ad-hoc system, the only difference being that the design models were refined and translated by applying SORT.

Each system is described by a system document of comparable length,[7] which contains a problem description, an OO design (i.e., a UML class diagram and a textual description of the classes and their methods), and a source code listing. Both systems are comparable in terms of domain, complexity, and representation (UML diagrams and C++ code). The only difference is that the SORT system provides additional information in terms of SORT documents by providing refined (i.e., implementation level) diagrams and the patterns used for refining and translating it to source code. Appendix B provides examples concerning the application of SORT onto the original system. Both systems were taken from the domain of computer games, implementing a simple breakout game ("Ad-hoc" system) and a game of "Metamagical Themas" (Hofstadter, 1996; "SORT" system), respectively.

To further demonstrate the comparability of both systems, Table 1 provides simple counts related to artifacts and relationships. In detail, the data concerning number of classes, attributes, methods, and relationships show that there are no significant differences between the Ad-hoc and the refined SORT system. The distribution of attributes and methods in both systems reveal that the classes in both systems are comparable concerning size and complexity. Additionally, the table provides standard design measures for each of the three systems (original and refined SORT system, as well as the Ad-hoc system). In detail, these are DIT (depth of inheritance tree), NOC (number of children), and CBO (coupling between objects) measures

[7] The Ad-hoc system documentation has a length 31 pages incl. 474 lines of code. The SORT system documentation has a length of 39 pages incl. 9 pages of additional SORT documentation (developed within 0.5 person day), and 454 lines of code. The reference document (i.e., refinement & translation patterns) has a length of 21 pages.

(Chidamber and Kemerer, 1994; Briand et al., 1997a; Briand, 1999). Design measurement was performed to illustrate two things: First, to show that the refinement of the original SORT system has reduced the system's complexity. Second, to demonstrate that there are no significant differences between the original systems. Consequently, data provided by Table 1 demonstrates that a serious attempt was made to keep the differences between the SORT and Ad-hoc system documents to those caused by the technique (SORT, ad-hoc) applied.

3.4. Experimental Tasks

The experiment required each subject to perform two sets of tasks for each system documentation, whereby both tasks were pen-and-paper based and did not require the use of a computer (e.g., for executing code). First, subjects had to complete a questionnaire, which asked various questions about the system (see also Appendix C). The rationale was that the collection of all questions provides a test of a subject's understanding of the system. It also prepared them for the verification task. In detail, the questions are about:

1. The overall understanding of the system, to familiarize the subjects with the experimental systems and provide insight into whether they had understood the functional and non-functional aspects of the system.
2. The relationships between the design and its implementation, to test the traceability established by SORT in comparison to the Ad-hoc system (i.e., aimed at testing the difficulty of relating models and code).
3. More specific questions, answerable from the system documentation, to check the subjects' understanding of how the specified non-functional requirements of the system (e.g., response or reaction times) are addressed.

The questionnaire for each system contained exactly the same number of questions, with the questions being conceptually similar (i.e., directed to the same area of interest) and in the same logical order. In order to avoid subjects getting only a selective understanding of the system (i.e., only knowing about single classes), the questions required them to look at several elements in the design. The set of all questions was organized in a way that enables subjects to get an overview of the complete system.

The second task required the subjects to verify the system in terms of correctness and fulfillment of non-functional requirements (see also Appendix D). Subjects were asked to mark all places in the system documentation that are either incorrect or that are "wrong" in relation to the (non-) functional requirements. This was practically supported by providing the subjects with a small checklist, which told them how and what to check.

The empirical validation of the SORT approach focused on the mapping of UML models to code, therefore subjects had to work on both, UML diagrams and source code, in order to answer questions and identify defects. In general, the tasks were defined in such a way that comparisons of SORT and the Ad-hoc approach could be made (regarding defect type, numbers, and complexity). In addition, a "pre-test" with two student employees was performed. These students, who did not participate in the original experiment, worked on both systems and on a set of tasks. This was done in order to check (a) that the tasks could be performed in the given time, (b) that the questions can be answered, that defects can be found with or without the additional SORT information, and (c) that the complexity of questions and tasks is comparable. It appeared that the complexity of tasks and questions were comparable for both treatments and that the tasks for the SORT system could also be solved without using the additional information. Concerning the time needed for performing the verification task differences where found between systems. The reason is that it is not easy to insert defects (concerning correctness and quality) into a design that require an equal amount of time to be identified. However, both systems contained the same number of defects to be found (i.e., 22), which were comparable in their type and complexity (see also Section 3.5). Thus, a reasonable attempt was made to exclude any impact on the factors of interest.

After completion of both tasks, subjects had to complete a debriefing questionnaire (see Appendix E). This questionnaire, an adapted version of the questionnaire used in Briand et al. (1997b) captures (1) personal details and experience, (2) opinions with respect to a subject's motivation and performance, and (3) opinions on the experiment itself.

3.5. Defect Seeding

In the context of this experiment 22 defects were seeded into the correct UML document versions. Defects, typically encountered in model-based development, were seeded into the system documentation. The defects focused on the link between design and source code. Therefore, syntax or standard programming errors were not considered. However, all defects seeded to the design were also visible in the code. Example of these defects are missing/wrong associations, inconsistencies between diagrams and source code, missing/wrong sequences in sequence diagrams, missing/wrong states, etc. The distribution of the seeded defects, their type, and complexity is similar in both systems.

3.6. Procedure

The week before the experiment took place, subjects were given an additional practical session as training. The idea behind the training session was not to be a pilot study, but to familiarize subjects with the experimental setting and procedure, and, more generally, to answer any questions they had. In detail, the participants had a chance to practice experimental tasks using a system documentation (incl. UML-based design and source code), not related to one of the experimental systems, to avoid learning effects. At the end of the training session the subjects were told to review what they had learned before participating in the experiment. It is important to note that the training session was not planned to be a pilot study. The major lesson learned from performing the training session was that the types of tasks used (concerning understanding and verification) were manageable by the subjects.

The experiment was then performed over two consecutive days with each subject receiving a different set of system documents each day. Each experimental run took place in a classroom where the subjects had plenty of space to examine all the system documentation. Each subject sat next to a subject who was examining the other system documentation—this was performed to reduce plagiarism, although this was not a significant worry. Subjects were told that there were different systems being worked upon, but were not told anything about the nature of the study, i.e., which hypotheses were being tested, in order to avoid discussions after the first run, which might have had an impact on the results of the second run. The subjects were then given a maximum of one and a half hours to complete all tasks. In general, subjects were free to allocate their time to the tasks. However, after 45 minutes they were reminded that they had already spent half of their time and that they soon should start the second task. During the experiment subjects were told not to talk among themselves, but to direct any questions they had to the monitor. Questions directed towards the monitor were not answered if believed to assist a subject's performance. After completing their tasks, subjects completed the debriefing questionnaire and then returned this and all experimental materials to the monitor before leaving.

3.7. Design

As shown in Table 2, a standard within-subject 2 $\times$ 2 factorial design was employed [see, e.g., (Bortz, 1993)]. The two independent variables are the experimental run (X–run 1 and run 2) and the technique applied to the object-oriented system (Y–SORT and Non-Sort). An important advantage of using a within-subjects design such as this is that the error variance due to differences among subjects is reduced. From personal experience in performing software engineering experiments (Briand et al., 1997b,c) and from the literature (Curtis, 1980), it can be said that when dealing with small samples, variations in participant skills are a major concern that is difficult to fully address by randomization or blocking. At the same time, it can result in the independent variables becoming confounded with the order of presentation, which can lead to learning and fatigue effects. To control this, counterbalancing was introduced, i.e., half the subjects had to work on SORT system in experimental run 1 and then on the second system in run 2 (group A). The other half of the subjects did the opposite (group B).

To further control differences between subjects, random assignment to these two groups was performed. This was achieved by drawing a letter for each subject from a hat. Once this had been performed subjects were then shown to a desk with the appropriate system documentation. As the number of subjects was known before running the experiment, it was a simple procedure to create two groups of equivalent size, which is important to prevent the independent variables from becoming non-orthogonal. Each subject remained in the same group for the second experimental run.

Table 2 Experimental design employed

Variable X	Variable Y—implementation technique	
Run	SORT	Non-SORT
1	A	B
2	B	A

Since a within-subjects design was applied, blocking, used to neutralize background variables that cannot be eliminated by randomizing (Wohlin et al., 2000), is done via subjects (i.e., each subject sees both treatments). Further blocking has not been performed in order not to artificially restrict the sample. However, this might have had an impact on the results of this study and is therefore discussed as a threat to validity.

3.8. Dependent Variables and their Collection

The dependent variables (i.e., what is measured) are, on the one hand, oriented towards the subjects' understanding of the design and implementation, and, on the other hand, towards defect identification. The subjects' understanding was measured based on their accuracy in completing a questionnaire. Data for the verification was collected in two ways: (i) subjects had to mark the system documentation exactly where they thought a defect occurs, and (ii) subjects had to fill out a data collection form to summarize their findings. This allowed for cross-checking of the form. In addition, the time needed for completion of all tasks was recorded. In summary, the following dependent variables, enhanced by qualitative data from the debriefing questionnaire, were derived:

- **Understanding_Time**, the time (minutes) needed to understand the system in order to complete the questionnaire. It is expected that the completion of the questionnaire will require less time for the SORT system than for the Ad-hoc system.
- **Understanding_Correctness**, the correctness of understanding tasks (i.e., the number of correctly answered questions). The set of questions to be answered by each subject contained the same number of questions and was used to measure the subjects' understanding. Consequently, it is reasonable to use the number of correctly answered questions as a measure of understanding. Therefore, it is expected that a subject will answer significantly more questions correctly for the SORT system, showing his/her better understanding of the system, than for the Ad-hoc system.
- **Verification_Time**, the time (minutes) needed for identifying and documenting defects. It is expected that the verification of the SORT system will be performed in less time than the verification of the Ad-hoc system.
- **Verification_Completeness**, the completeness or effectiveness of the verification task, captured by the number of defects to be found. According to (Briand et al., 1997c), completeness, normalized on the scale of 0 to 1, can be calculated as:

$$\text{Verification\_Completeness} = \frac{\text{number of correct defects found}}{\text{total number of defects to be found}}$$

In the context of this experiment it is expected that the effectiveness of the verification task will be significantly higher for the SORT system than the effectiveness for the Ad-hoc system. In other words, subjects, when verifying the SORT system, will spot more of the inserted defects than subjects verifying the Ad-hoc system.

- **Verification_Accuracy**, the accuracy of the verification task captured by the number of places identified correctly. According to (Briand et al., 1997c), accuracy, normalized on the scale of 0 to 1, can be calculated as:

$$\text{Verification\_Accuracy} = \frac{\text{number of correct defects found}}{\text{number of defects indicated as found}}$$

  It is expected that subjects, when verifying the SORT system, will spot only true defects and not places where they thought a defect may be. In contrast, it is expected that subjects verifying the Ad-hoc system will find non-existing defects caused by the unspecific links between models and code.
- **Verification_Rate**, the verification rate or efficiency, which can be calculated as:

$$\text{Verification\_Rate} = \frac{\text{number of correct defects found}}{\text{Verification\_Time}}$$

  It is expected that the efficiency of the verification task will be significantly higher for the SORT system than the efficiency of the Ad-hoc system. In other words, subjects, when verifying the SORT system, will spot more true defects in less time than subjects verifying the Ad-hoc system.

3.9. Data Analysis

Data were collected for the 27 subjects over the two experimental runs (4 subjects were unable to return for their second run due to prior commitments). Therefore, 24 data points were available for analysis for the application of SORT and 26 data points were available for the ad-hoc approach; because the experimental design is completely within-subjects, repeated measures analysis can be performed. The first step of the analysis procedure is to analyze the descriptive statistic of the collected data and to discuss anomalies. Then the data has to be checked for normality—if the data is substantially non-normal, then non-parametric tests have to be used, otherwise parametric tests can be applied. To proceed with the analysis, a level of significance (i.e., the α-level) has to be preset. From a scientific perspective, it is necessary to work at a low α-level (Bortz, 1993). Therefore, the α-level for the experimental evaluation of SORT is set to $\alpha = 0.05$.

4. Results

Table 3 presents a descriptive summary of the dependent variables for the two different object-oriented systems. The columns represent the number of valid observations (N), the mean (x), the median, the minimum and maximum values, and the standard deviation (s). The rows provide these data for each of the dependent variables. In addition, Appendix F presents raw histograms of the dependent variables concerning correctness and completeness. These give a first impression that the results are in the predicted direction.

A quick review of the table and the raw histogram (see Fig. 17) shows that with regard to understanding the subjects showed a deeper understanding, of the SORT

Table 3 Descriptive statistics

Variable	SORT system					
	N	$\bar{x}$	$\tilde{m}$	Min	Max	S
Understanding_Time	22	32.82	30	10	52	10.99
Understanding_Correctness	24	4.75	5	2	6	1.11
Verification_Time	22	48.91	50	30	80	11.15
Verification_Completeness	24	0.56	0.54	0.18	0.90	0.19
Verification_Accuracy	24	0.87	0.87	0.69	1	0.08
Verification_Rate	22	0.26	0.27	0.075	0.43	0.09
Variable	**Ad-hoc system (non-SORT)**					
	N	$\bar{x}$	$\tilde{m}$	Min	Max	S
Understanding_Time	24	34.62	34.5	20	75	11.06
Understanding_Correctness	26	2.85	3	1	4	0.92
Verification_Time	24	49.62	50	15	60	9.77
Verification_Completeness	26	0.36	0.36	0	0.64	0.15
Verification_Accuracy	26	0.46	0.46	0	0.81	0.15

system than of the Ad-hoc system by answering more questions correctly (5–median response for the SORT system and 3–median response for the Ad-hoc system). However, the time spent on the understanding task does not differ that clearly, since the median differs by 4.5 minutes for the SORT and Ad-hoc system, respectively. The statistical test of the concerned dependent variable has to show if this small difference is significant. Otherwise, it has to be discussed more deeply.

The descriptive statistics as well as the raw histograms (see Figs. 14–16) concerning the verification task illustrate a comparable situation. In principle, the subjects verifying the SORT system showed a higher effectiveness (0.54–median response) than subjects verifying the Ad-hoc system (0.36–median response). However, the mean value illustrates that approximately only 50% of the inserted defects were detected for the SORT system (0.56–mean response). However, the situation concerning the Ad-hoc system is even worse (0.36–mean response). A reason for this situation may be tight time constraints, which can be elicited from the debriefing questionnaire and from the mean values for the dependent variables concerning time. Concerning the remaining dependent variables, the data shows that subjects verifying the SORT system found defects more correctly (0.87–median response) and efficiently (0.27–median response) than subjects verifying the Ad-hoc system (0.46 and 0.17–median responses).

Table 3 shows that there are quite large differences between the minimal and maximal values concerning the time for understanding and verification tasks. Thus, time might have an impact on factors such as completeness or correctness. Thus, the time/accuracy trade-off has to be analyzed. A scatter plot can be a helpful tool in determining the strength of the relationship between the variables. The plots for this experiment (see Fig. 4) indicate increasing trends, therefore a regression test was performed to analyze if there is a relation between time and correctness/ completeness. The results showed that with a high probability, there appears to be

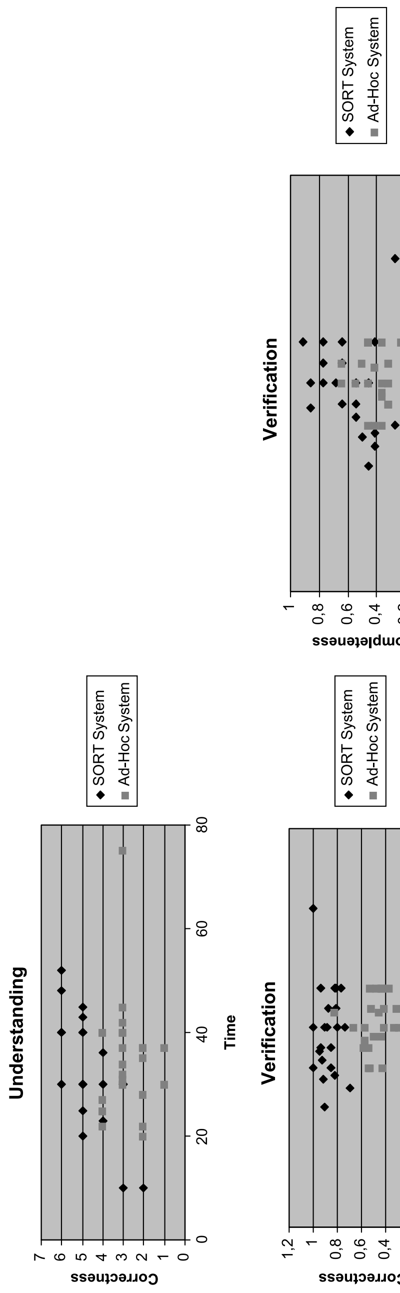


Fig. 4 Scatter plots—time vs. correctness/completeness regarding understanding & verification

no association between the variables. Thus, it can be excluded that there is a significant time/accuracy trade-off.

Before deciding which tests can be applied to the collected data, the data have to be checked for deviations from a normal distribution. The standard test is the Shapiro–Wilks' W test. This test was applied and its result indicates that for all dependent variables, except Verification_Completeness and Verification_Rate, the data is substantially non-normal. As a result, non-parametric tests are the appropriate tests to apply—as the data is within-subjects, the most appropriate of these is the Wilcoxon matched pairs test. However, to make sure a parametric test would not have led to different conclusions, the t-test for dependent samples [which is thought to be robust against minor deviations from normality (Welkowitz et al., 1976)] was applied in addition. In every case, the parametric test supported the findings of the non-parametric one.

4.1. Anomalies in the Data

Examination of Table 3 shows one strange result in the data set, it is the low number of subjects having valid times for the variables Understanding_Time and Verification_Time (i.e., N is far below 27 for the SORT and Ad-hoc systems, respectively). Subjects were told to record on their collection sheet at what time they finished the understanding tasks and moved onto the modification tasks. Many subjects (e.g., five subjects for the SORT system) unfortunately did not obey this instruction. Although the total time for performing both tasks is known, it is impossible to split the time between the time spent on understanding and the time spent on verification. Therefore, the missing data is discussed as a threat to validity. Another problem was that some subjects participated in only one experimental run, because they had to attend another exam. Although, valid data for one run was collected, the nature of the Wilcoxon matched pairs test prohibits them from further analysis (Bortz, 1993).

4.2. H_1

The results of the Wilcoxon matched pairs test applied on the two dependent variables concerned with H_1 are shown in Table 4. Column one represents the dependent variable, column two the degrees of freedom, column three the Z value of the Wilcoxon test, column four the critical value for $\alpha = 0.05$, which Z has to exceed to be significant, and column five provides the p value. This is further illustrated by the Box–Whisker Plot on the dependent variable "Understanding_ Correctness" shown in Fig. 5. An interesting fact in this regard is that the low scores for understanding correctness are mainly caused by the same subject (see also Appendix F, Fig. 17). Although the experience and motivation is average for this subject, he spent only very little time on the understanding task (10 minutes) and focused more on the verification task.

Table 4 Wilcoxon matched pairs test-H1

Variable	Valid *N*	Z	Crit. *Z*0.95	*p*-value
Understanding_Time	21	0.58	1.64	0.55
Understanding_Correctness	23	4.01	1.64	0.00

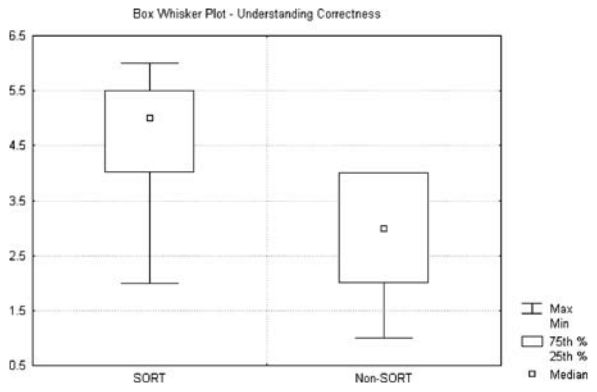


Fig. 5 Box–whisker plot—Understanding_Correctness

Table 4 shows that there is no significant difference in the amount of time spent on completing the understanding tasks, but there was a significant difference for the variable Understanding_Correctness, which measured the extent of a subject's understanding. Therefore, an analysis of the qualitative data, collected by means of the debriefing questionnaire, was performed. Regarding question 8 (if you could not complete the tasks, indicate why) it was found that 86% (20 subjects from 23) and 69% (18 subjects from 26) of the participants felt they ran out of time when performing the required tasks on the SORT and Ad-hoc system, respectively. This is an indication that the time allocated for the experiment was at best sufficient for them to complete their entire task. Often this is regarded as "normal" (i.e., subjects ran out of time in performing tasks with strict time restrictions), but the qualitative data from debriefing questionnaire indicate that the allocation of more time would have had different results. Another reason might be that subjects were reminded after 45 minutes to start the second task soon. The median responses concerning the dependent variable Understanding_Correctness differed by 2, although the time spent by both groups is nearly the same. Therefore it can be assumed that people working on the SORT system would have completed their understanding tasks quicker than people working on the Ad-hoc system. Consequently, the tight time[8] constraints explain why performance in terms of Understanding_Time was not significantly better for the SORT system and why the effects of SORT were only reflected in terms of Understanding_Correctness. Therefore, H_1 is accepted.

4.3. H_2

Table 5 presents a summary of the results of the statistical tests for the four dependent variables concerned with H_2. The results are further illustrated by Figs. 6–8, which display Box–Whisker Plots on the dependent variables Verification_Accuracy, Verification_Completeness, and Verification_Rate.

[8] The time for conducting an experiment within the SE-I lecture was limited to 90 minutes of training and 90 minutes per experimental run.

Table 5 Wilcoxon matched pairs test-H_2

Variable	Valid N	Z	Crit. $Z0.95$	p-value
Verification_Time	21	0.52	1.64	0.60
Verification_Completeness	23	3.24	1.64	0.00
Verification_Accuracy	23	4.10	1.64	0.00
Verification_Rate	21	3.73	1.64	0.00

The table shows that there was no significant difference in the amount of time spent on completing the verification tasks, but there was a significant difference for the variables Verification_Completeness, Verification_Accuracy, and Verification_Rate, which measured the completeness, correctness, and rate of the verification task. The lack of significance regarding Verification_Time can also be explained, as for hypothesis H_1, by tight time constraints during the experiment. An analysis of the debriefing questionnaires showed that participants felt they ran out of time when performing the verification task on the SORT and Ad-hoc system, respectively. This is an indication that the allocation of more time would have had led to different results concerning Verification_Time. Therefore, H_2 is accepted.

4.4. Analysis Summary

Overall the results show that the SORT system (i.e., the system developed by applying SORT technology) is significantly easier to understand and verify than the Ad-hoc system (i.e., the system developed in an ad-hoc way). This is supported by the data concerning correctness/accuracy, completeness,[9] and rate of the responses[9] and further supported by results obtained from analyzing the qualitative data from the debriefing questionnaire. Therefore, H_0 can be rejected.

Because of tight time constraints during the experiment ("ceiling effect"), differences in understanding and verification times could not be observed at a significant level. Consequently, future replications should allocate more time to each experimental run.

5. Threats to Validity

In this section the study's threats to validity and a discussion of what attempts were made to mitigate them is presented.

5.1. Construct Validity

Construct validity is the degree to which the independent and dependent variables accurately measure the concepts they purport to measure. The following possible threat was identified:

- Understandability and verifiability are difficult concepts to measure. In the context of this paper it is argued that the defined dependent variables are

[9] Only measured for verifiability.

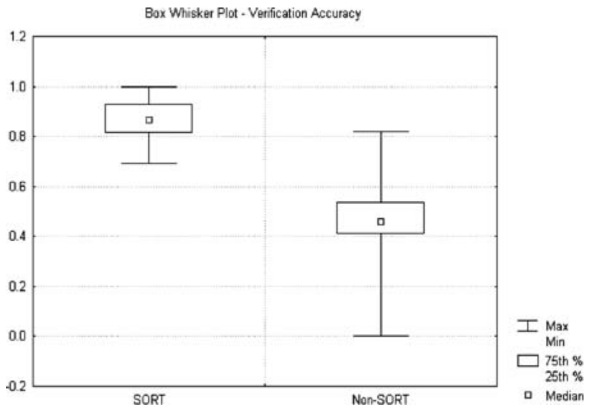


Fig. 6 Box–whisker plot—Verification_Accuracy

intuitively reasonable measures. Of course, there are several other dimensions of each concept.[10] However, in a single controlled experiment it is unlikely that all the different dimensions of a concept can be captured. In fact, there must be a focus on what can be realistically achieved. Since it is more careful to use several different measures to capture a concept, in the context of this experiment two and four measures were defined for the concepts of understandability and verifiability, respectively. Since these measures consistently support the stated hypotheses, as does the collected qualitative data, it is reasonable to have confidence in their construct validity. However, additional studies are required to investigate the other dimensions of understandability and verifiability.

5.2. Internal Validity

Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. The following possible threats were identified:

- A maturation effect is caused by subjects learning as an experiment proceeds. The threats to this study are that (1) subjects learned enough from the first experimental run to bias their performance in the second run, and (2) subjects learned from the understanding tasks to bias their performance during the verification task.

  These threats were addressed by using counterbalancing (i.e., half of the subjects worked on the SORT system in the first run and on the Ad-hoc system in the second run; the other half of the subjects did the opposite). Counterbalancing is known to eliminate ordering effects caused by the tasks, as well as any learning

[10] Answering questions is not the only important dimension of understandability. Developing system descriptions based on the overall understanding and detecting wrong assumptions are of equal importance.

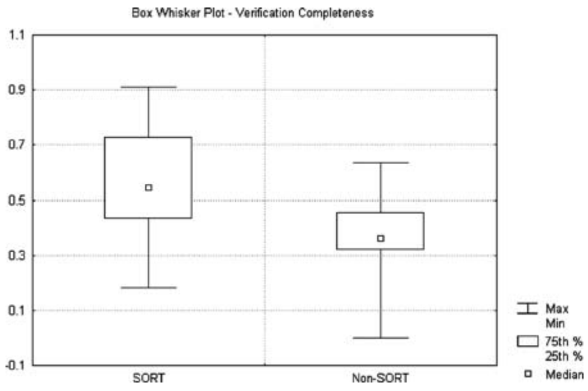


Fig. 7 Box–whisker plot—Verification_Completeness

and fatigue effects (Watt and van den Berg, 1995). In addition, counterbalancing would distribute such effects across both groups, hence reducing their impact. Furthermore, additional statistical tests (i.e., analyzing the impact of the order of execution onto the results) have not shown any learning effect.

- A subset of subjects did not follow experimental instructions, which resulted in an occurrence of missing time data. In general, this data loss may not have been random and thus had an influence on the results.

  However, this threat can be addressed by comparing the mean of the two variables Understanding_Time (32.82 and 34.62) and Verification_Time (48.91 and 49.62). In both cases, the mean values do not show significant differences, as one may have expected by non-random data losses. Thus, it can be assumed that this loss had no effect on the outcome of the study.

- A potential ceiling effect concerning time might have occurred due to the tight constraints, since significant differences in understanding and verification times were not observed.

  In the context of this experiment there are no significant relationships between completeness or correctness and time (see Fig. 4). Thus, it is unlikely that the allocation of more time would have led to different results concerning

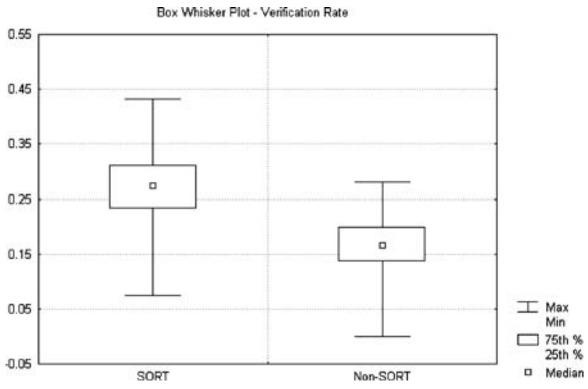


Fig. 8 Box–whisker plot—Verification_Rate

completeness or accuracy. However, this might be different for the dependent variables concerning time. On the one hand, according to the debriefing questionnaire, subjects felt they ran out of time. On the other hand, based on the discussion presented in (Arisholm and Sjøberg, 2004) the following assumption can be made: Subjects who work fast may spend more time on the last task than they would otherwise. Similarly, subjects who work slowly may have insufficient time to perform the tasks correctly. Thus, future replication may devote more time to the tasks and may exclude the last task (e.g., question 5 and the last five defects) from the analysis.

- An instrumentation effect may result from differences in the experimental materials employed (Briand et al., 2001). The threat to this study is that possible differences between the two systems and the tasks other than those controlled (i.e., applied technique, etc.) have caused differences in subjects' performance. In principle, this may have caused differences in cognitive complexity and task complexity.

  Concerning cognitive complexity, this threat was addressed by seriously attempting to keep the differences between both systems to those caused by the applied development technique (i.e., SORT and non-SORT). This is supported by the data points for simple system properties and design complexity measures as presented in Table 1. According to these data values, both original systems contain a similar amount of information and have a comparable complexity level. The application of SORT resulted in slightly different metric values for the refined SORT system, which is a further indicator that the differences between the two systems used are caused by the applied technique (i.e., the subject of interest). Concerning task complexity, this threat was addressed by a serious attempt to ensure that the tasks are highly similar (i.e., similar questions and defects in terms of type and numbers). Thus, it can be assumed that the "uncontrolled" differences between the systems (i.e., differences not caused by the applied technique) had no effect on the results of this study.
- The selection of subjects and their assignment to groups might have had an impact on the results. Randomization (random assignment) of group membership was used as a counter-attack against this threat. However, randomization is no guarantee for an equal experience distribution of subject experience between groups. Blocking can be used to neutralize this factor, but is not easy to perform given the small group size (Daly et al., 1996). Since a within-subjects design was applied, blocking was done via subjects. Each subject saw both treatments and the differential performance (performance on the SORT system, performance on the Ad-hoc system) of all subjects was averaged and compared. No preliminary test was administered to the subjects that might have assessed this bias in order to avoid problems of reactivity and sensitization.

5.3. External Validity

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. The following possible threats were identified:

- The subjects who participated in this study were students and are therefore unlikely to be representative of software professionals. However, the results can be useful in an industrial context for the following reasons: Industrial employees often do not have more experience than students since OO projects are often performed by persons who have just finished their university education or by people who apply object technology for the first time. Furthermore, laboratory settings, such as a university context, allow the investigation of a larger number of hypotheses at a lower cost than field studies. The hypotheses that seem to be supported in the laboratory setting can then be tested further in more realistic industrial settings with a better chance of discovering important and interesting findings. Conversely, laboratory experiments can be used to confirm results obtained in field studies, where control, and thus internal validity is usually weaker.

- The use of volunteers as subjects may affect the validity of the study (i.e., selection bias). Individuals who volunteer for an activity are almost certainly different from those who do not volunteer. Volunteers are, by definition, motivated to participate and presumably expect to receive some benefit from the intervention. These differences between study participants and non-participants limit the ability to generalize the results of this study beyond the research sample (Wohlin et al., 2000). Since 90% (27 of 30) of the students enrolled in the software engineering lecture volunteered to participate, it is highly probable that similar results will be obtained when performing the experiment in a similar way at the University of Kaiserslautern, and the results can probably also be generalized to other software engineering courses. However, future experiments have to be performed, varying in subjects, dependent variables, etc. to obtain more generalizable results.
- The materials used in this study (i.e., the software systems and tasks) may not be representative in terms of their size and complexity. However, experiments in a university context do not allow the use of "realistic" systems and tasks for several reasons, such as time, availability of such systems, etc. Another problem might be the defect seeding. Regardless of the distribution or origin of the defects, the generality of the results is threatened by the fact that these defects and their distribution across the system classes may not be representative of all situations.

It is important to stress the fact that the author views this study as exploratory. While these three threats limit generalization of this research, they do not prevent the results from being used in further studies. In addition, the weakness imposed by these threats may be eliminated if similar results can be obtained by using different empirical techniques (Briand et al., 2001). The idea is that the weaknesses of one study can be addressed by the strengths of another; see, e.g., (Kaplan and Duchon, 1988) or (Wood et al., 1999). In principle, this can be the application in an industrial context, the use of "realistic" software systems and tasks, or case studies that measure the effects of SORT during development.

5.4. Conclusion Validity

Conclusion validity is the degree to which conclusions about relationships in the data are reasonable. The following possible threats were identified:

- *Violated assumptions of statistical tests.* Every analysis is based on a variety of assumptions about the nature of the data, the procedures used to conduct the analysis, and the match between these two. Making wrong assumptions may lead to erroneous conclusions about relationships. One important issue in this regard is the distribution of the data (normal or non-normal), which is the major selection factor for the statistical test to be used. The application of the Shapiro–Wilks' W test, showed that for all dependent variables, except Verification_Completeness and Verification_Rate, the data distribution is substantially non-normal. Thus, non-parametric tests are the appropriate tests to apply. To make sure a parametric test would not have led to different conclusions, the t-test for dependent samples [which is thought to be robust against minor deviations from normality (Welkowitz et al., 1976)] was applied in addition. In every case, the parametric test supported the findings of the non-parametric one. Therefore, the violation of test assumptions can be neglected.

- *Fishing and the error rate problem*. This threat relates to the problem that if you play with the data long enough, you can often "turn up" results that support or corroborate your hypotheses (Trochim, 2002). In other words, this can be seen as "fishing" for a specific result by analyzing the data repeatedly under slightly differing conditions or assumptions. The probability assumption of the statistical tests used assumes that each analysis is "independent" of the other. This might not be true, since multiple analyses of the same data were performed (e.g., accuracy, completeness, etc.). However, since all statistical tests, except those concerning time, showed a high significance, chances are low that there is no real relationship in the data.
- *Low statistical power*. This threat is related to the probability that a statistical test rejects H_0 for a specified value of an alternative hypothesis. Thus, it can be regarded as the ability of a test to detect an effect, given that the effect actually exists. However, power analyses for non-parametric tests are not easy to perform. Although Singer et al. (1986) claim that some variants of power analyses for nonparametric tests can be conducted by adjusting the result obtained for the corresponding parametric test, an adaptation for Wilcoxon's matched pairs test is unknown. Therefore, a power analysis concerning the t-test for dependent samples was performed, which showed that the statistical power is higher than 0.8 concerning all dependent variables (except time). Therefore, the author believes that low statistical power as a threat to validity can be neglected.

5.5. Addressing Threats to Validity

In order to improve the experiment and address some of the threats to validity identified above, the following actions can be taken:

- **Increase the task time**. The time allocated for answering questions and performing verification should be increased because many subjects stated the reason they did not complete all the tasks were time constraints. Alternatively, the experimental tasks should be made less complex. The result of this improvement may mean that the experiment will find similar values for the dependent variables concerned with completeness and accuracy but shorter completion times and hence, increased modification rates for the good design.
- **Improve time data collection procedures**. The data collection procedures for collecting precise time data should be improved upon due to the experience with

subjects not fully obeying experimental instructions. One method for achieving this manually would be to give subjects only the understanding questionnaire to begin with. Once a subject has completed the questionnaire he/she indicates this to a monitor, who records the time and then provides the subject with the verification tasks. A second option would be to automate the whole experimental procedure, thereby making time collection trivial.

6. Summary and Conclusion

With the rapid rate of innovation in object technology, one might have expected to have seen significant improvements in the quality of object-oriented systems. In practice, however, this has not fully happened due to the primitive techniques still largely dominating the "implementation" phase of object-oriented development methods. Leading edge technologies such as distributed objects, components and the UML will never achieve their full potential until this problem is addressed.

In an ideal world software systems would be written in formal languages, with formally verified mappings between levels, but for object-oriented development this is a long way off in practice. SORT offers a practical approach to this problem which provides effective assistance in the mapping of UML analysis and design models to code. This is achieved through the application of three time-honored strategies for managing complexity:

- Reducing the size of the individual steps. A sequence of several smaller steps is easier to understand than one big step. With the SORT approach the task of implementing graphical models is broken into individual refinement and translation sub-steps.
- Separating concerns. Developers can concentrate on single activities and do not have to worry about several things at once. These single steps facilitate a more systematic refinement of graphical models to code-level abstractions before starting translation.
- Identifying and exploiting commonality. The SORT approach simplifies multiple implementations of a single model (i.e., porting).

To demonstrate that the promises of the SORT can be achieved in practice, the approach has to be empirically evaluated. This paper describes a controlled experiment, conducted at the University of Kaiserslautern that used a within-subject design to investigate the effects of refinement and translation patterns on the verifiability and understandability of object-oriented systems. The experiment used two systems, comparable in size and complexity, and let subjects perform understanding and verification tasks on these.

The results of the empirical evaluation show that applying the SORT technology in system development had a significant impact on understandability and verifiability. In detail, subjects had a significantly better understanding of the SORT system regarding correctness and completeness of understanding questions. In addition, subjects performing verification tasks on the SORT system performed significantly better, regarding verification rate, correctness and completeness. This supports the initial assumption (stated by the research hypothesis) that SORT allows the quality development of object-oriented software systems. However, the experiment did not allow evaluation of the impact of SORT concerning time aspects.

A number of threats to validity hinder the generalization of results. Thus, in future studies, improvements to the experimental procedure might include increasing the task time and improving the time data collection procedures. However, the author believes that the quality of object-oriented designs is sensitive to systematic and sound engineering style software development, as addressed by SORT. From a more general perspective, if the results of this study are further confirmed through external replication and complementary studies, organizations involved in object-oriented development should invest in the adaptation and application of SORT to ensure that their software products comply with the principle of separating and explicitly distinguishing between refinement and translation. A good example of such a complementary study is the evaluation of SORT in development. In contrast to this study, which evaluates the effects SORT had on a system, such a case study can evaluate SORT as a development methodology, by analyzing the development of a software system performed by two independent groups: one using SORT, and the other one following a different development/mapping approach.

A replication package (in German) is available for researchers interested in externally replicating the experiment.[11] An English version is currently under preparation.

Appendix A. Example NOF Elements

It is not possible in a paper of this size to provide a complete and detailed description of the NOF and its elements. However, in this appendix a brief overview of some of its major elements is provided, by addressing in turn the three separate parts of the definition.

UML Subset

At its core, the NOF contains those elements of the UML which embody the fundamental elements of object-oriented systems such as classes, attributes, links, associations and inheritance. Rather than list all the all the basic elements which are included, it is actually more interesting to identify some of the major concepts which are not deemed appropriate for the NOF. Some of the features of which do not correspond directly to programming level concepts include:

Specialized Compartments. These compartments are used to show specialized abstract properties of a class (responsibilities, business rules, etc.) By definition therefore, such compartments play their major role in the analysis phase to help developers understand the domain, but do not play an important role in implementation. Consequently they are not included in the NOF. The only compartments which are acceptable in the NOF are those for attributes, operations and exceptions.

Association Class. Association classes describe an association that is also a class. Although it is stated (OMG, 2001) that an association class is not the same as a class connecting two other classes, no existing object-oriented language

[11] The replication package can be obtained from the author by sending an email request.

supports any other implementation (Lee and Tepfenhart, 1997) (i.e., a dictionary class is used). Consequently association classes are not part of the NOF.

Class-in-state. Classes with a state machine may have many states. The class-in-state modeling element describes a state that objects of that class can hold. Due to its close relation to activity diagrams, it is another way to accomplish the same goal as dynamic classification. Therefore it not necessary as a part of the NOF.

Dependency. All dependencies which describe historical connections between elements (e.g., ≪trace≫) do not influence the implementation of a system and are therefore not part of the NOF.

Derived Elements. These are not part of the NOF because they are used for the purpose of clarity and do not provide additional semantic information.

N-Ary Associations. N-ary associations are associations between three or more classes. However, just as for association classes no currently existing object-oriented language provides direct support for such associations; they have to be "simulated" using multiple associations.

Qualifier. These are used to partition a set of objects connected with an object via an association. Qualifiers are not part of the NOF for two reasons. First, due to their definition as attributes of an association (see also association class). Second, they are clearly analysis elements, which model an important semantic situation, but do not influence the general strategy for implementing an association.

Additional Elements

Although the UML is a powerful tool for describing object-oriented software systems it is not possible to describe all properties of programs entirely in the UML subset within the NOF. Certain extensions (i.e., new elements) are needed. The NOF includes legal extensions to the UML defined using the in-built extension mechanism. Most of the UML extensions in the NOF occur in connection with associations. This is because the fundamental implementation variations for associations are not fully supported in the present version of the UML. Although associations can vary in many ways at the analysis and design level, such as in their arity (binary, ternary, etc.) and their multiplicity (e.g., one-to-one, one-to-many, many-to-many), at the implementation level there are far fewer variations. All inter-object relationships are essentially implemented by the same basic mechanism: one object holding a pointer to, or the value of, another object. Even the implementation of associations by 'Relation Tables' makes use of these mechanisms, by implementing the table as a class in its own rights which routes the communication.

Following ION (Atkinson and Izygon, 1995), this basic relationship between classes is called "*clientship*." Clientship is an asymmetric relationship; the client needs to be aware of the identity of the server class, but the server requires no knowledge of the client class. All clientship relationships are therefore represented with a UML navigation arrow indicating the direction of the client/server relationship. As with all program-level relationships, clientship implies a compilation (and thus static) dependency. In total there are four different, orthogonal properties of clientship relationships, each with two possible values. Each possible value for each property has a corresponding UML association stereotype:

Attached vs. *Detached*: One of the most important characteristics of a clientship relationship is whether the client holds a reference to the server or whether it holds the actual state (i.e., the value) of the server. When the client holds a reference to the server the clientship is said to be *detached*, whereas when the client actually holds the state of the server the clientship is said to be *attached*.

Permanent vs. *Transient*: Another important characteristic of a clientship relationship is how long the class has visibility of a particular instance of the server class. If the client holds a reference to, or the value of, the server in its main data structure, the clientship is said to be *permanent*. If, on the other hand, the client has visibility of a server object only for the duration of a single method, the clientship is said to be *transient*.

Proper vs. *Intimate*: Normally, a client class only has access to the "official", publicly visible methods of the server. This is termed *proper* clientship. Most object-oriented languages also allow client classes to be given privileged access to the server. This is termed *intimate* clientship.

Direct vs. *Indirect*: The final property of a clientship relationship is whether the client holds visibility (of the server), or whether the client relies on a second server for visibility of the first. The first situation is known as *direct* clientship, and the second *indirect* clientship.

Constraints

Most of the UML modeling elements can be used at different levels of abstraction. For example in the analysis phase an operation can be specified by a simple name whereas during detailed design it can be specified in more detail (types, parameters, etc.). In general, the UML allows the modeler to decide when and where certain pieces of information should be displayed. While this flexibility is fine during high-level modeling phases, it becomes a problem in the later implementation phases when the "as is" implementation needs to be described. This is because from an individual diagram is not possible to determine whether the absence of specific markings is because they have simply been

Table 6 Selection of constraints

Constraints
All attributes of a class must have visibility markings. In other words, it must be clear whether they are to be implemented as public, private or protected members (in the case of C++).
A method must have a visibility marking, a list of parameters (if existing), and a return type.
A parameter is specified in the following form: name:type=default-value.
Each class must have at least a constructor and destructor method.
The methods of a class have to be grouped by using one of the following stereotypes ≪constructor≫, ≪destructor≫, ≪update≫, etc.
The parameter of a template class must be bound to an actual value to be meaningful.
A clientship relationship has to be augmented with multiplicity markings.
A clientship relationship has to be augmented with roles.

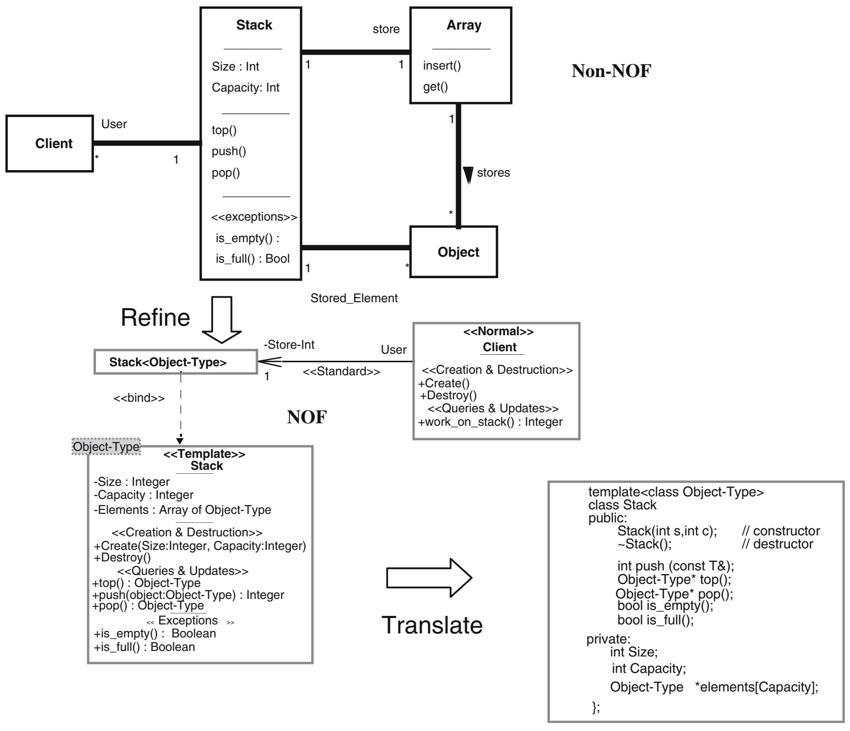
omitted or because the corresponding decisions remain to be made. A general goal of the NOF is to rule out such ambiguities by providing constraints, which clearly specify the level of detail to be represented. Due to the size of the UML and the limited space within this paper we cannot present the full set of constraints. However, Table 6 presents a short selection to give an overview of their nature.

Example

As an example of the application of NOF consider a simple stack. Figure 9 shows how the original stack model is refined to a more detailed UML before it is finally translated into equivalent C++ code.

Appendix B. Application SORT

This experiment uses a system documentation obtained by applying the SORT approach. This documentation was developed by applying SORT refinement patterns to the original design (see Fig. 10), resulting in a refined design (Fig. 11) (see Fig. 12), updated class descriptions (see Fig. 13) accompanied by the sequence of transformation steps including the applied patterns. The latter was also provided for the application of translation patterns concerning the system's



```
template<class Object-Type>
class Stack
public:
        Stack(int s,int c);      // constructor
        ~Stack();                // destructor

        int push (const T&);
        Object-Type* top();
        Object-Type* pop();
        bool is_empty();
        bool is_full();
private:
      int Size;
      int Capacity;
      Object-Type   *elements[Capacity];
 };
```

Fig. 9 Refinement and translation by using the NOF

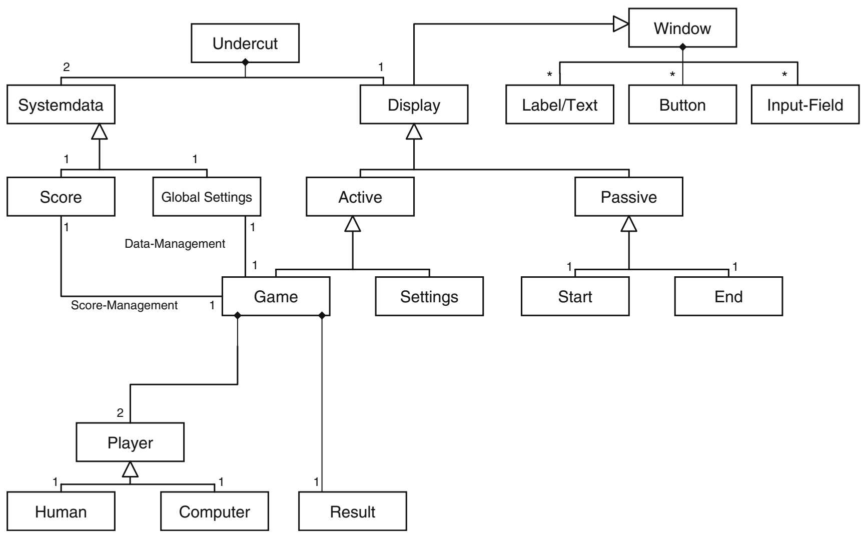


Fig. 10 Original SORT system—main architecture

implementation. Please note, that, due to size and complexity, the examples given in this appendix are a simplified/smaller version of the original models.

Appendix C. Generic Descriptions and Examples of Questions

Understandability is measured by asking the subjects to complete a questionnaire with five questions regarding the overall understanding, structure, and functionality of both systems. While keeping time constraints in mind, these questions were intended to capture the diversity of questions that a maintainer might ask about a system. Although the systems differ, the questions for both are comparable on a generic level (e.g., question 4 is aimed at the services a specific class offers). In the following we describe, at a generic level, each question in more detail.

- Question 1–Describe the overall functionality of the system.
  This question is intended as a starting point for the subjects to get

Class Undercut

Task: Provides the main control loop for the system and manages its aggregated parts. The main goal is to guarantee a smooth execution of the system.

Attributes:

current_round:Integer

Methods:

start-Game() // Launches the Undercut Game

Fig. 11 Original class description

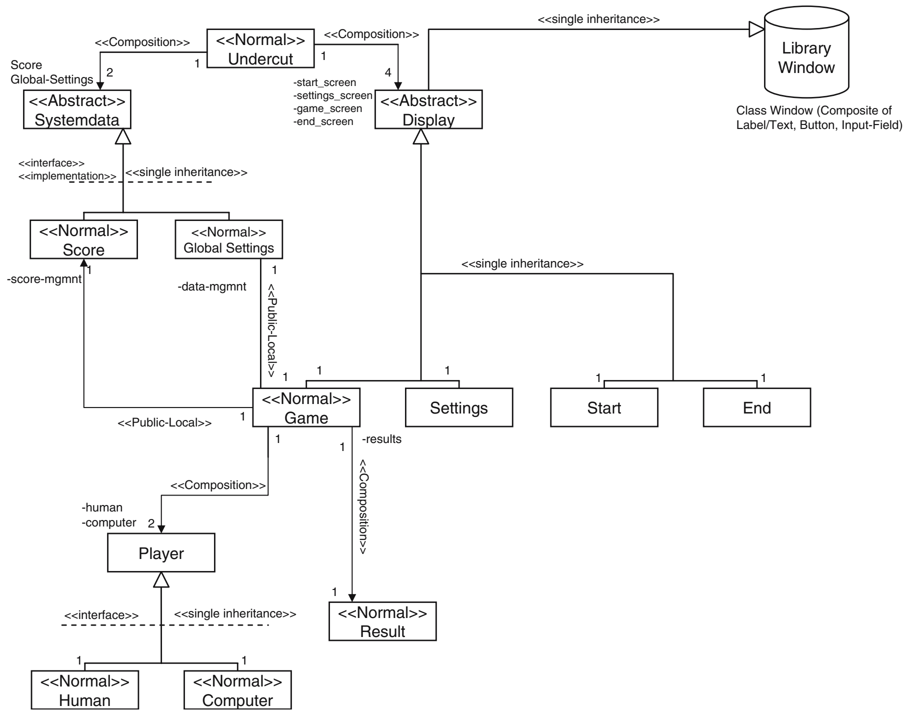


Fig. 12 Refined SORT system—main architecture

comfortable with the setting and the system. *Example: Please give a brief description of the overall system functionality.*

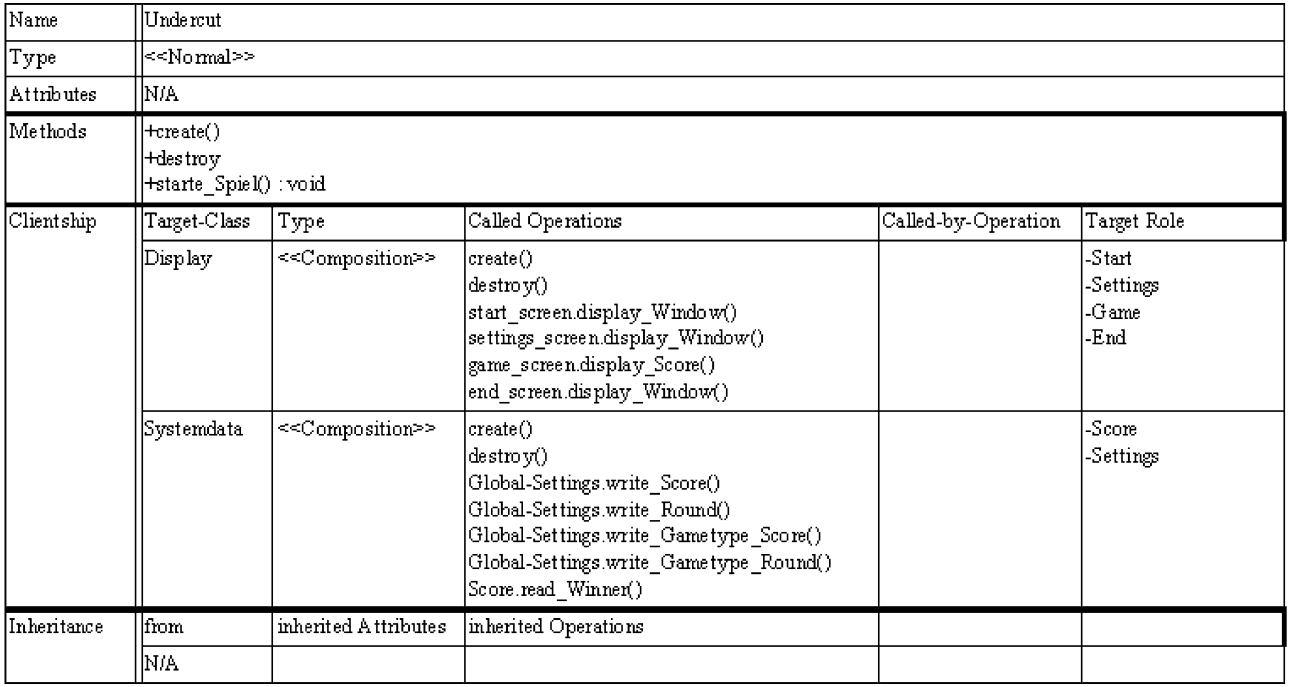
- Question 2–Map system functionality onto the architecture of the system (i.e., the subjects have to identify the classes that are outside the system scope).

  To answer this question subjects have to understand the structure of the system and its functionality before they can decide what is outside the system boundaries. *Example: Identify the library classes of the system and for each class describe how (Call, Inheritance, etc.) and by whom (class) it is used.*
- Question 3–Describe the services and dependencies of a specific class.

  For this question the subjects have to understand the functionality of a specific class in terms of operations, return values, etc. In addition, they have to understand the interactions between classes. *Example: Please, describe the dependencies and inherited attributes of the class* '*Scoring.*'
- Question 4–Describe the functionality implemented by a number of classes (i.e., inherited operations that were overloaded).

  This question requires the understanding of the systems architecture, components, classes, and their interaction. *Example: Please describe the functionality of the methods target-hit() in the system.*
- Question 5–Discuss how the non-functional requirements are addresses within the design and implementation of the system.

  To answer this question, subjects have to deeply understand the system documentation and have to trace non-functional aspects throughout design and



Name	Undercut				
Type	<<Normal>>				
Attributes	N/A				
Methods	+create() +destroy +starte_Spiel() : void				
Clientship	Target-Class	Type	Called Operations	Called-by-Operation	Target Role

	Display	<<Composition>>	create()
			destroy()
			start_screen.display_Window()
			settings_screen.display_Window()
			game_screen.display_Score()
			end_screen.display_Window()
					-Start
					-Settings
					-Game
					-End

	Systemdata	<<Composition>>	create()
			destroy()
			Global-Settings.write_Score()
			Global-Settings.write_Round()
			Global-Settings.write_Gametype_Score()
			Global-Settings.write_Gametype_Round()
			Score.read_Winner()
					-Score
					-Settings

Inheritance	from	inherited Attributes	inherited Operations		
	N/A				

Fig. 13 Updated class descriptions—example

implementation. *Example: Please, describe shortly how the non-functional aspect performance is treated and addressed within the design.*

Appendix D. Verification Tasks

Verification is measured by asking the subjects to check the design and code documents by performing pre-defined tasks, in order to identify defects (excluding syntax errors). Although the systems differ, the tasks for both are comparable on a generic level. In the following we describe, at a generic level, the tasks in more detail.

Goal: The goal of this task is to verify if the given C++ implementation is correct concerning its design.

Documentation: Verify the code by identifying all places within the design and code documents related to any defects (excluding syntax errors). In doing so, please:

- Enter the page and line number in the defect table.
- Mark the identified place within the documentation. Use the number given in the defect table to clearly identify the error.

Approach: Start with single classes of the design. If there are differences between design and implementation, please check if this is a defect and if yes, document it accordingly. In doing so, please perform the following activities for each class:

- Check if the functionality of a design class matches its implementation.
- Check if the implementation of a method (interface, visibility, etc.) matches its design.
- Check if the visibility, type, initialization, etc. of every attribute is correct.
- Check if the dependencies (Associations, Inheritance, etc.) between classes are correctly realized. Please pay special attention to overloading and inheritance of operations.
- Check if classes are correctly instantiated.

Appendix E. Debriefing Questionnaire (Translated from German)

The information you provide in this questionnaire may be very valuable to us. Please answer each question as honestly as you can. Anything you write down will be treated confidentially. Thank you.

Personal Details and Experience

Id. Number (e.g., A1): Qualifications (e.g., Informatik Vordiplom):

Please answer the following **four** questions based on this experience scale:

None	Little	Average	Substantial	Professional
1	2	3	4	5

Base your answers upon what you knew **before** attending the course.

1. What is your experience with software engineering practice? (circle number).

1	2	3	4	5

2. What is your experience with design and code documents in general? (circle number).

1	2	3	4	5

3. What is your experience with object-oriented documents (esp. UML & C++)? (circle number).

1	2	3	4	5

4. What is your experience in performing verification tasks (identifying errors)? (circle number).

1	2	3	4	5

Motivation and Performance

1. Estimate how motivated you were to perform well in the study. Please explain your answer:

Not	Poorly	Fairly	Well	Highly
1	2	3	4	5

2. Estimate how well you understood what was required of you.

Not	Poorly	Fairly	Well	Highly
1	2	3	4	5

3. What approach did you adopt to do the exercise? (tick only one).
   (a) Read the documents fully and then attempt the tasks.
   (b) Read the documents while thinking about the tasks.
   (c) Straight into the tasks, reading the documents as required.
   (d) Other—please specify:

4. Estimate the correctness (in %) of your answers to the understanding questionnaire.

0–20	21–40	41–60	61–80	81–100
1	2	3	4	5

5. Estimate the correctness (in %) of your answers to the verification tasks.

0–20	21–40	41–60	61–80	81–100
1	2	3	4	5

6. Estimate the accuracy (in %) of your answers to the verification tasks.

0–20	21–40	41–60	61–80	81–100
1	2	3	4	5

7. Estimate the completeness (in %) of your answers to the verification tasks.

0–20	21–40	41–60	61–80	81–100
1	2	3	4	5

8. If you could not complete all the tasks, please indicate why. (tick only one).
   (a) Ran out of time.
   (b) Did not fully understand the task.
   (c) Did not fully understand the design & code documents.
   (d) Other—please specify:

9. In your opinion, what caused you the most difficulty to understand the documents? (mark only one).
   (a) Nothing in particular.
   (b) The notation (UML, C++) used.
   (c) Inheritance.
   (d) Cohesion in classes, i.e., the relationships within a class.
   (e) Coupling between classes, i.e., the relationships between classes.
   (f) Level of detail of the documents.
   (g) Other — please specify:

10. In your opinion, what caused you the most difficulty to perform verification on the documents? (mark only one).
   (a) Nothing in particular.
   (b) The notation used.
   (c) Inheritance.
   (d) Cohesion in classes, i.e., the relationships within a class.
   (e) Coupling between classes, i.e., the relationships between classes.
   (f) Level of detail of the documents.
   (g) Other — please specify:

Miscellaneous

1. How do you judge the size of the documents you had? (please circle).
2. On a scale of **1 to 10** estimate, in terms of **understandability**, the quality of the documents you had. (1 – barely understandable; 10 – easily understandable).

Too small	Small	About right	Large	Too large
1	2	3	4	5

   Please specify number:

3. On a scale of **1 to 10** estimate the overall difficulty of the tasks you have been asked to perform. (1 – very easy; 10 – very difficult). Please specify number:
4. What do you personally think about using refinement and translation patterns for system design and implementation? Please specify:
5. Having performed the tasks, would you do anything different the next time around? Please specify:
6. Have you learned anything from participating in this study? Please specify:
7. Any additional comments?

   Thanks once again.

Appendix F. Histograms on Experimental Results

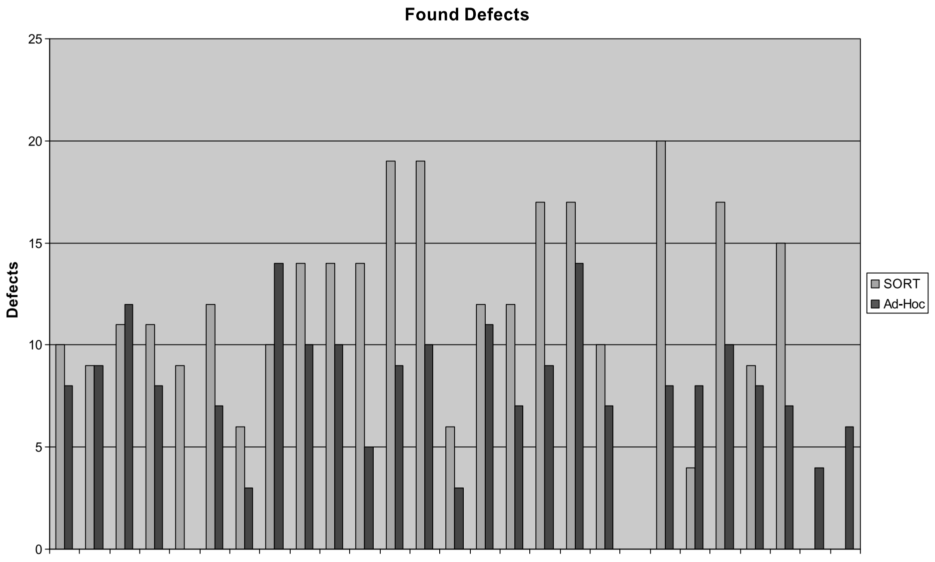


Fig. 14 Found defects per subject

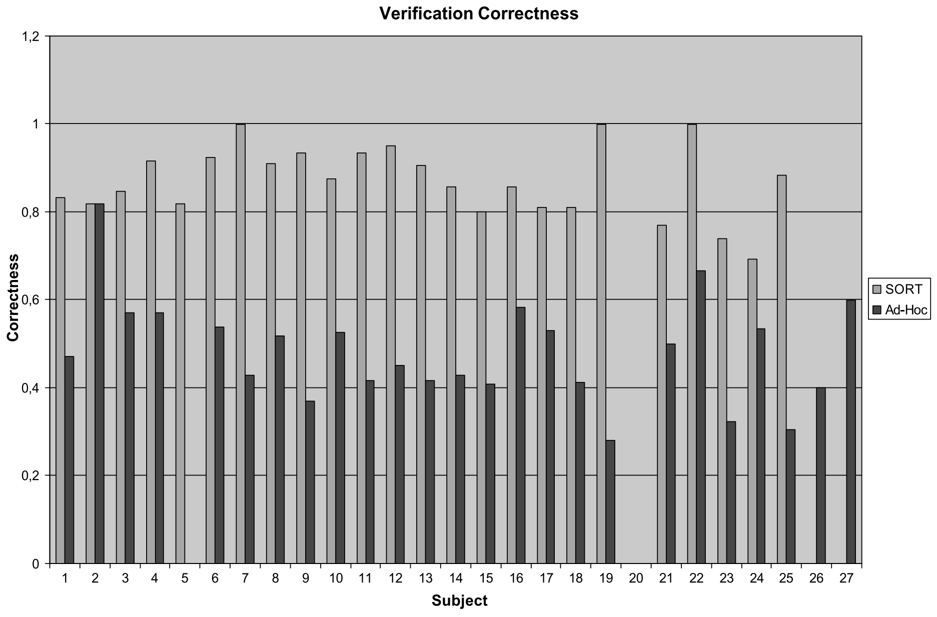


Fig. 15 Verification correctness per subject

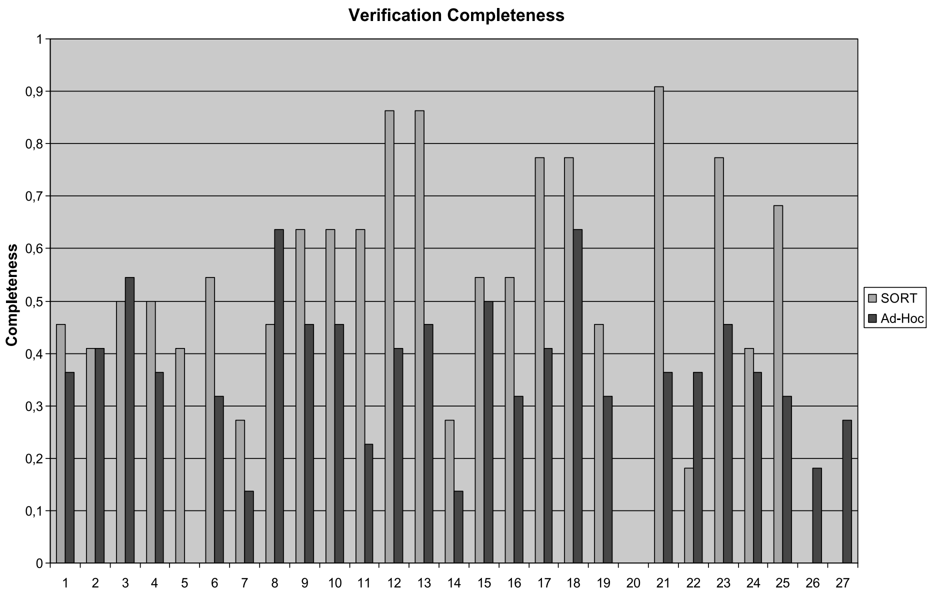


Fig. 16 Verification completeness per subject

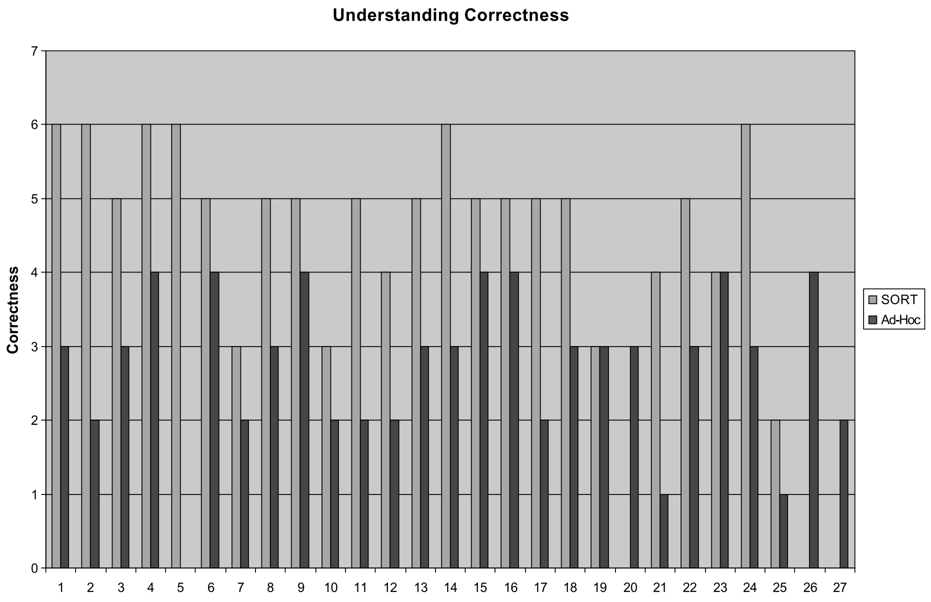


Fig. 17 Understanding correctness per subject

References

Arisholm E, Sjøberg, DIK (2004) Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. IEEE Trans on Softw Eng 30(8):521–534

Astels D (2002) Refactoring with UML. Proceedings of the 3rd International Conference eXtreme Programming and Flexible Processes in Software Engineering, pp 67–70

Atkinson C, Izygon M (1995) ION a notation for the graphical depiction of object-oriented programs. Technical report, University of Houston-Clear Lake

Atkinson C, Bayer J, Bunse C, et al. (2001) Component-based product line engineering with UML, Addison Wesley.

Bell AE, Schmidt RW (1999) UMLoquent expression of AWACS software design. Commun ACM 42(10):55–61

Booch G (1994) Object oriented analysis and design with applications (2nd edition). Benjamin/Cummings, Redwood City, CA

Bortz J (1993) *Statistik für Sozialwissenschaftler* (4th edition). Springer-Verlag, (in German)

Briand LC, Daly JW, Wüst J (1997a) A unified framework for cohesion measurement. Proceedings of the Fourth International IEEE Symposium on Software Metrics, Metrics'97

Briand LC, Bunse C, Daly JW (1997b) An experimental evaluation of quality guidelines on the maintainability of object-oriented design documents. In: Wiedenbeck S, Scholtz J (eds), Proceedings of the Empirical Studies of Programmers: Seventh Workshop ESP7, ACM Press, pp 1–19

Briand LC, Bunse C, Daly JW, Differding C (1997c) An experimental comparison of the maintainability of object-oriented and structured design documents. Journal of Empirical Software Engineering 2(3):291–312

Briand LC, Daly JW, Wüst J (1999) A unified framework for coupling measurement in object-oriented systems. IEEE Trans on Softw Eng 25(1):91–121

Briand LC, Bunse C, Daly JW (2001) A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Trans on Softw Eng 27(6):513–530

Bruegge B, Dutoit AH (2003) Object-oriented software engineering: using UML, patterns, and java. Prentice Hall

Bunse C, Atkinson C (1999a) Improving quality in object-oriented software: systematic refinement and translation of models to code. Proceedings of the 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99), Paris, France

Bunse C, Atkinson C (1999b) The normal object form: bridging the gap from models to code. Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, USA

Bunse C (2001) Pattern-based refinement and translation of object-oriented models to code, Fraunhofer IRB, ISBN 3-8167-5613-1

Cheesman J, Daniels J (2000) UML components: a simple process for specifying component-based software. Addison-Wesley

Chidamber SR, Kemerer CF (1994) A metrics suite for object-oriented design. IEEE Trans on Softw Eng 20(6):476– 493

Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F, Jeremaes P (1993) Object-oriented development: the fusion method. Prentice Hall

Coplien JO, Schmidt DC (1995) Pattern languages of program design. Addison-Wesley (Software Patterns Series)

Curtis C (1980) Measurement and experimentation in software engineering. Proc IEEE 68(9):1144–1157

Daly J, Brooks A, Miller J, Roper M, Wood M (1996) Evaluating inheritance depth on the maintainability of object-oriented software. Empirical Software Engineering, An International Journal 1(2):109–132

US Department of Defense (DoD) (1983) Reference manual for the Ada Programming Language

D'Souza DF, Wills AC (1998) Objects, components and frameworks with UML: the catalysis approach. Addison-Wesley

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns–elements of reusable object-oriented software. Addison-Wesley

Grady RB (1992) Practical software metrics for project management and process improvement. Prentice Hall, NJ

Green S, Kouchakdjian A, Basili V, Weidow D (1990) The cleanroom case study in the SEL: project description and early analysis. Technical Report SEL-90-002, NASA–SEL

Herrington J (2003) Code generation in action. Manning Publications Company

Hofstadter DR (1996) Metamagical themas: questing for the essence of mind and pattern. Basic Books

Holub A (2002) When it comes to good OO Design, keep it simple—feature-rich OO Design Tools fail where simple solutions succeed. Java World, a copy of the article can be obtained at http://www.javaworld.com/javaworld/jw_01_2002/jw_0111_ootools_p.html

Jeckle M (2005) Overview on UML CASE Tools, pages on the WWW which can be obtained at http://www.jeckle.de, last visited Spring 2005

Kaplan B, Duchon D (1988) Combining qualitative and quantitative methods in information systems research: a case study. MIS Quarterly, pp 571–586

Kruchten P (1998) The rational unified process: an introduction. Addison-Wesley

Lee RC, Tepfenhart WM (1997) UML and C++. A practical guide to object-oriented development. Prentice Hall

Leite J (2000) Lecture notes—software engineering 1 (SE-1). University of Kaiserslautern, Germany, Fall-Semester 1999–2000

Meyer B (1992) EIFFEL—the language. Prentice Hall

Object Management Group (OMG) (2001) OMG unified modeling language specification, version 1.5. Technical report

Object Management Group (OMG). (2005) OMG UML 2.0 superstructure specification. Technical Report, OMG

Martin RC, Riehle D, Buschmann F (1997) Pattern languages of program design 3. Addison-Wesley (Software Patterns Series)

Medvidovic N, Egyed A, Rosenblum DS (1999) Round-trip software engineering using UML: from architecture to design and back, 1999. Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR), Toulouse, France, pp 1–8

Reed PR Jr (2001) Developing applications with Java and UML. Addison-Wesley

Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W (1991) Object-oriented modeling and design. Prentice Hall

Satpathy M, Harrison R, Snook C, Butler M (2000) A generic model for assessing process quality, proceedings. of the 10th international workshop on software measurement (IWSM 2000), lecture notes in computer science, Volume 2006, Springer, pp 94–110

Sendall S, Küster J (2004) Taming model round-trip engineering. OOPSLA/GPCE: Best Practices for Model-Driven Software Development

Singer B, Lovie AD, Lovie P (1986) Sample size and power. In: Lovie AD (ed), New Developments in Statistics for Psychology and the Social Sciences. The British Psychological Society and Methuen, London, pp 129–142

Stroustrup B (1993) The C++ programming language (2nd edition). Addison-Wesley

Webster BF (1995) Pitfalls of object-oriented development. M&T Books

Trochim W (2002) Threats to conclusion validity, pages on the WWW available at: http://www.socialresearchmethods.net/kb/concthre.htm, Last visited in March 2005

Watt JH, van den Berg S (1995) Research methods for communication science. Allyn and Bacon, Needham Heights, MA

Welkowitz J, Ewen R, Cohen J (1976) Introductory statistics for the behavioral sciences (2nd edition). Academic Press

Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A (2000) Experimentation in software engineering—an introduction. Kluwer Academic Publishers

Wood M, Daly J, Miller J, Roper M (1999) Multi-method research: an empirical investigation of object-oriented technology. Journal of Systems and Software 48(1):13–26

Dr. Christian Bunse received the Phd degree in computer science from the University of Kaiserslautern Germany and his B.S. (Vordiplom) and M.S. degree (Diplom) in computer science with a minor in medicine from the University of Dortmund, Germany. From 1993 to 1995 he was a faculty research assistant of the Software-Technology-Transfer-Initiative at the University of Kaiserslautern, Germany. Currently, he is employed as department head at the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany. His current research focus is on model-driven and component-based approaches to (embedded) system development, and particularly, on pattern-based and aspect-oriented techniques in this area.