Empirical Software Engineering, 7, 193–231, 2002.
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Review of Experimental Investigations into Object-Oriented Technology

IGNATIOS S. DELIGIANNIS igndel@it.teithe.gr
Technological Educational Institute of Thessalonici, Greece

MARTIN SHEPPERD mshepper@bmth.ac.uk
Empirical Software Engineering Research Group, Design, Engineering and Computing, Bournemouth University Royal London House, Christchurch Road Bournemouth BH1 3LT, UK

STEVE WEBSTER steve.webster1@btinternet.com
Semaphore Europe Ltd, UK

MANOS ROUMELIOTIS manos@uom.gr
University of Macedonia, Greece

Abstract. In recent years there has been a growing interest in empirically investigating object-oriented technology (OOT). Much of this empirical work has been experimental in nature. This paper reviews the published output of such experiments—18 in total—with the twin aims of, first, assessing what has been learnt about OOT and, second, what has been learnt about conducting experimental work. We note that much work has focused upon evaluation of the inheritance mechanism. Whilst such experiments are of some interest, we observe that this may be of less significance to the OOT community than experimenters seem to believe. Instead, OOT workers place more emphasis upon other mechanisms such as composition, components, frameworks, architectural styles and design patterns. This leads us to conclude that the empirical researchers need to ensure that their work keeps pace with technological developments in the fields they aim to investigate.

Keywords: Experiment, object-oriented technology, software architecture.

1. Background to the Review

Over the past decade the adoption of object-oriented technology (OOT) has greatly increased to the extent that it could now be regarded as the dominant software technology, certainly for non-legacy systems. It has been argued that software development has become too complex for structured methodologies to handle. An example of such a viewpoint is Riel (1996) who, amongst others, has suggested that the object-oriented (OO) paradigm, with its decentralised control flow, bi-directionally related data and behaviour, implicit case analysis (i.e., polymorphism), and information-hiding mechanisms offers a good opportunity for controlling complexity.

Presently, however, questions about the extent to which OOT has fulfilled its promises are answered more by intuitive feelings and anecdotal evidence, than by empirical and quantitative evidence (Basili and Burgess, 1995). Intuition may provide a starting point, but it needs to be backed up with empirical evidence. Without proper grounding, intuition can always be challenged. For this reason, over recent years, there has been a growing interest in empirical evaluation. Unfortunately, good examples of solid experimentation in computer science are comparatively rare (Tichy, 1998; Zelkowitz, 1998).

For the purposes of this review we consider an experiment to be a controlled empirical investigation into some phenomenon with a clearly stated hypothesis and random allocation of subjects to different treatments. A key motivator for using a formal experiment, rather than a case study, is that the results of an experiment can be more easily generalised than those of a case study. Another is that it provides the investigator with a much greater degree of control than is usually possible with case studies. The disadvantages tend to be in the size of artefacts and the laboratory type setting.

There is a formal terminology for describing the components of an experiment. *Object of study* is the entity that is studied in the experiment. They can be products, processes, resources, models, metrics or theories. *Treatments* are the different activities, methods or tools we wish to compare or evaluate. When we are comparing using a treatment with not using it, a *control* must be established, which provides a benchmark. A *trial* is an individual test run, where only one treatment is used. *Experimental subjects* are the people applying the treatment, for example using an OO programming language to solve a particular problem. The *response* or *dependent variables* are those factors that are expected to change or differ as a result of applying the treatment, for example, time taken or accuracy. By contrast, *state* or *independent variables* are those variables that may influence the application of a treatment and thus indirectly the result of the experiment. The number of, and relationships among, subjects, objects and variables must be carefully described in the experimental plan. *Criteria* for measuring and judging effects need to be defined, as well as methods for obtaining the measures. Finally, two important concepts are involved in the experimental design: *experimental units* which are the experimental objects to which a single treatment is applied, and *experimental error* which is the failure of two identically treated experimental units to yield identical results.

Fenton and Pfleeger (1997) suggest six steps of carrying out a formal experiment.

(1) Conception—deciding what we wish to learn more about, and define the goals of the experiment. From this, we must state clearly and precisely the objective of the study.

(2) Design—to translate the objective into a formal *hypothesis*. The goal for the research needs to be re-expressed as a hypothesis that we want to test. The hypothesis is a tentative theory or supposition that we think explains the behaviour we want to explore. Frequently, there are two hypotheses but may be more than two. The *null hypothesis* assumes that there is no difference between the *treatments* (that is, between competing methods, tools, techniques, environments, or other conditions whose effects we are measuring) with respect to the

dependent variable(s). The *alternative hypothesis* posits that there is a significant difference between the treatments. 'Testing the hypothesis' means determining whether the data is convincing enough to reject the null hypothesis, to accept the alternative one as true.

(3) Preparation—to make ready the subjects and the environment. If possible, a pilot study of the experiment should be conducted.

(4) Execution.

(5) Analysis—this phase consists of two parts. First, all the measurements taken must be reviewed in order to ensure that they are valid and useful. Second, there follows the analysis of the sets of data according to usual statistical principles.

(6) Dissemination and Decision-making—to document the experimental materials and conclusions in a way that will allow others to replicate and confirm the conclusions in a similar setting. The experimental results may be used in three ways. First, by using them to support decisions about how to develop or maintain software in the future. Secondly, to allow others to suggest potential improvements to their development environments. Thirdly, to perform similar experiments with variations in experimental subjects or state variables.

Over the last decade various researchers have conducted a range of experiments and empirical studies attempting to evaluate the practical benefits, drawbacks and other aspects of OOT. We have identified 27 such experiments,[1] however, we consider that only 18 of them belong to the category of controlled experiments in which we are most interested (see Table 1). The review is potentially of value for two reasons. First, it provides a good foundation for the design of further experiments. Secondly, the review enhances our understanding of the benefits (or otherwise) of OO technology.

The remainder of the paper is organised as follows. First—since it is evident that many experiments have focused upon class inheritance—we summarise recent developments in OO architecture. Next we describe and review published experimental work in the field of OOT. Finally, the paper concludes by discussing the current state of play in empirical evaluation and identifies potentially fruitful avenues for further investigation.

2. Recent Developments and Issues in OO Architecture[2]

We now briefly consider the current state of play in OO architecture. Software design is considered to be the skeleton of a software system, thus its quality significantly impacts the quality of the final products. Many people argue that the secret of good OO design is to end up with a class model that does not distort the conceptual reality of the domain. Success in this, as it is argued, helps lead to maintainable systems, because such models tend to be comparatively easy to understand, and therefore comparatively easy to modify sensibly (Pooley and Stevens, 1999).

Table 1. Summary of published experimental research into OOT.

Investigators	Included?	Area of Investigation	Reason for Exclusion
Abreu and Melo (1996)	N	Metrics evaluation	Not a controlled experiment
Agarwal et al. (1996)	Y	OO versus procedural technology	
Agarwal and Sinha (1996)	Y	OO versus procedural mindset	
Agarwal et al. (1999)	Y	OO versus procedural technology	
Basili et al. (1996)	N	Metrics evaluation	Not a controlled experiment
Briand et al. (1997)	Y	OO versus procedural technology	
Briand et al. (2001)	Y	OO design principles	
Cartwright (1998)	Y	Inheritance	
Chatel and Detienne (1994)	N		A case study
Corritore and Wiedenbeck (2000)	Y	OO versus procedural technology	
Cunis (1997)	N		A case study
Daly et al. (1996)	Y	Inheritance	
Harrison et al. (2000)	Y	Inheritance	
Henry and Humphrey (1990)	Y	OO versus procedural technology	
Laitenberger et al. (2000)	Y	Inspection techniques	
Lee and O'Keefe (1996)	Y	OO versus procedural technology	
Lewis et al. (1991)	Y	OO versus procedural technology	
Moynihan (1996)	Y	OO versus functional technology	
Pant et al. (1996)	N		Uses a single subject
Prechelt et al. (2001)	Y	Design patterns	
Prechelt et al. (2001)	Y	Design patterns	
Ramakrishnan and Menzies (1996)	N		Experiment unfinished
Shoval and Frumermann (1994)	N		Doesn't focus on the OO paradigm
Unger and Prechelt (1998)	Y	Inheritance	
Wiedenbeck et al. (1999)	Y	OO versus procedural technology	
Wiedenbeck and Ramalingam (1999)	N		No explicit hypothesis
Yida et al. (1998)	N		A case study

Class inheritance and object composition are mechanisms for extending a design. They are also the most common techniques for reusing functionality in OO systems. However their use must be carefully applied since they can be dangerous when used incorrectly (Riel, 1996). It is essential to view a system from two perspectives, seeing it as a 'kind-of' hierarchy as well as a 'part-of' hierarchy (Lieberherr et al., 1991). The challenge lies in applying these mechanisms to build flexible and reusable software. A number of design heuristics and design patterns have been proposed over recent years that help to achieve this aim.

Inheritance is a class-based relationship best used to capture the 'kind-of' relationship between classes. Its main purposes are twofold: it acts as a mechanism for expressing commonality between two classes (generalisation), and it is used to specify that one class is a special type of another (specialisation). Effectively it is just a mechanism for extending an application's functionality by reusing functionality in parent classes. Generally speaking, reuse (white-box) is the major motivation for inheritance (Amstrong and Mitchell, 1994).

It is often argued that inheritance should be utilised to model commonality and specialisation (Lieberherr and Riel, 1989; Capretz and Lee, 1993; Firesmith, 1995). Inheritance can also be used for subtyping, when substitutability is guaranteed (Amstrong and Mitchell, 1994) or when kind-of roles, transactions and devices are being modelled (Coad and Mayfield, 1997).

Inheritance is defined statically at compile time for most "popular" languages, and is straightforward to implement. Since it explicitly captures commonality it can facilitate modification. It is usually clearly shown in the architectural model and in the code structure (Amstrong and Mitchell, 1994; Coad and Mayfield, 1997). It reduces redundancy (Amstrong and Mitchell, 1994) and permits proper polymorphic substitution (Firesmith, 1995; Gamma et al., 1995).

Unfortunately, there are also disadvantages with inheritance. First the hierarchy becomes a compromise between classification and implementation purposes which can lead to classification problems (e.g., a square is a rectangle, however, we do not wish it inherit all the properties of a rectangle such as needing both length and width instance variables) (Seidewitz, 1996). The implementation inherited from parent classes cannot be changed at run-time. Also, parent classes often define at least part of their subclasses' physical containment violating encapsulation (Lieberherr and Riel, 1989; Coad and Mayfield, 1997). There is a strong coupling between superclass and subclass so that through a change to the superclass it is possible to force changes to the subclass. Implementation dependencies between parent and child classes limit flexibility and ultimately reusability. This is sometimes referred to as the rigidity problem (Seidewitz, 1996) or fragile base class problem (Pooley and Stevens, 1999). Another problem is the weak accommodation of objects that change subclass over time ('transmute problem') (Coad and Mayfield, 1997), for example a part-time employee accepts a full-time post. And finally, there is the yo-yo problem (self-reference) caused by the up-and-down traversals in order to gain a full comprehension of any operation. At any point in this traversal, the implementation may self-referentially invoke another operation. But the implementation of this other operation will likely be found in a subclass back down the hierarchy (Seidewitz, 1996). One proposed solution to the yo-yo problem is to use a delegation hierarchy instead of inheritance (Aksit and Bergmans, 1992). Nevertheless, inheritance can easily be misused, resulting in poor class structures and architectures that are difficult to extend, maintain, reuse, and understand (Rumbaugh, 1993; Firesmith, 1995).

Object composition is an alternative to class inheritance. Composition and aggregation are two kinds of whole-part associations. They form object-based relationships needed to model complex hierarchies of objects. It is often necessary for a part class in a whole-part association to be the composite class in another. So whole-part associations can induce multi-level object composition hierarchies (part-of hierarchies) (Civelo, 1993). Here, new functionality is obtained by assembling or composing objects to get more complex functionality. The composed objects are required to have well-defined interfaces. This style of reuse is called black-box reuse. In composition the whole strongly owns its parts (e.g., an Engine is part of a Car), implying that the lifetime of the 'part' is controlled by the 'whole'. This control may be direct or transitive. In aggregation the coupling is looser (e.g., a Module is a part of a DegreeCourse).

Comparing inheritance with composition we find that inheritance is only useful in limited contexts, whilst, composition is useful in almost every context (Coad and Mayfield, 1997). Most authors favour composition over class inheritance stressing some of the following benefits.

– It encourages black-box reuse (since it produces black-box implementations which are much easier to maintain than the white box implementations commonly associated with misused inheritance (Lieberherr and Riel, 1989)).
– It forces objects to respect each other's interfaces, and because objects are only accessible via their interfaces, encapsulation is not broken.
– Classes implementing interfaces are kept small and focused on one task resulting in better design of class hierarchies (Gamma et al., 1995).
– They can be defined dynamically at run-time through objects acquiring references to other objects.
– It allows having multiple instances of the used class, which isn't possible with inheritance.
– Object composition is also applied extensively in design patterns (Gamma et al., 1995).

There are, however, disadvantages. Aggregation most frequently occurs in design problems where parts of the same type are logically interchangeable (e.g., a bibliography need not be assigned to a single document but may apply across multiple documents) (Blaha, 1993). Composition can be a transitive relationship. Therefore, the number of composition levels that can be reliably propagated is based on whether the kind of composition relationship is the same, otherwise the application of propagation at each level must be examined for validity. This is sometimes referred to as the transitivity problem (Odell, 1994).

Design patterns provide a means for capturing knowledge about problems and successful solutions in software development making it easier to reuse successful designs. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. They can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class or object interactions and their underlying intent. Put simply, design patterns can help a designer get a design "right" faster, thus reducing the effort required to produce systems that are more resilient, more effective and more flexible (Gamma et al., 1995). The required functionality of the software system is realised as patterns of interactions between objects (Capretz and Lee, 1993). The use of patterns is essentially a form of reuse of well-established good ideas.

It is claimed that design patterns (Gamma et al., 1995; Coad, 1992) provide a number of advantages. First, they provide a common vocabulary for designers to use to communicate, document, and explore design alternatives. This improves communication both among designers and from designers to maintainers. Second, they offer solutions, "best practices", to common problems. Third, they capture the experience of expert designers. Fourth, patterns help novices to learn by example to behave more like experts. Lastly, describing a system in terms of the design patterns that it uses may make it a lot easier to understand. On the other hand there are also possible disadvantages. For example, it is argued that the pattern is often more powerful and complicated than might be necessary making understanding and change potentially more difficult. Moreover, this complexity may be hard to anticipate at the time of making the decision to use a pattern. This leads us to conclude that whilst patterns are potentially very important there is still much that we do not understand about their use in practice. This could be a topic that empirical researchers wish to further address.

A more recent and promising reuse model aiming at reusing large components and high-level designs is that of *frameworks*. Based on OOT, they are defined as "semi-completed applications that can be specialized to produce custom applications" (Johnson and Foote, 1988). Usually, a framework is made of a hierarchy of several related classes. Framework based development involves mainly two activities: development of the framework itself and development of an application based on the framework. Developing a framework is a more demanding process than building an application. The framework designer should have a deep knowledge of the application domain and has to foresee future requirements and domain evolutions. Domain analysis techniques, focused on modelling the scope of the domain, and commonalities and variability of applications, have been developed to guide the definition of framework requirements (Coplien et al., 1998). It has been observed that successful frameworks have been extracted from legacy systems, by abstracting the knowledge of principal software designers. Currently, there are only few examples of quantitative evidence to support project managers in decisions about framework-based development (Mattsson, 1999; Morisio et al., 1999).

3. Past Experimental Work

We now turn to the experimental work related to OOT.

Table 1 summarises the 27 published experiments that we have identified. Nine are excluded from subsequent analysis and the remaining 18 are grouped into five categories:

Comparing OO with structured technology.
Exploration of OO design principles.
Exploration of class inheritance.
Exploration of design patterns.
Exploration of inspection techniques.

To aid comparison of the reviewed experiments, we present them using a framework suggested by Wohlin et al. (2000).

3.1. OO versus Structured Techniques

Agarwal et al.'s (1999) *motivation* was that the learning curve, associated with the OO methodology, as with any new technology, might be fairly steep. When compared with structured techniques, which have dominated software development for over two decades, the OO approach represents a fundamental shift in focus. For organisations with a significant staff of information systems analysts and designers, a potential hurdle in incorporating the OO methodology is the procedural or process-oriented (PO) mindset of the analysts and designers. It is claimed that systems professionals experienced in PO modelling can be characterised as having a 'procedural mindset'. The *objects* studied are OO and PO analysis and design methodologies for the *purpose* of *investigating* the effects of prior PO modelling experience *with respect to problem solving performance* in OO modelling from the *point of view* of the *researcher*. The *context* concerns an experiment run using subjects performing on two tasks (fictional systems): The PO task was an "Accounts Payable System", and the OO task that was an "Employee Benefits System".

Hypotheses. The performance is compared with experienced and inexperienced modellers at two levels of task granularity: (i) the *task* level (OO and PO) and (ii) the *subtask* level (structure[3] and behaviour). H_0: There is no difference in the quality of solutions generated by experienced PO modellers and inexperienced modellers. H_1: Experienced PO modellers generate higher quality solutions than inexperienced modellers for the H_{1a}: PO task, H_{1b}: *behaviour* subtask. H_2: There is no significant difference in the quality of solutions generated by experienced PO modellers and inexperienced modellers. H_{2a}: for the OO task. H_{2b}: for the *structure* subtask.

Variables. The *independent variables* were *type of experience* (PO, inexperienced), and *task type* (OO or PO). The *dependent variable* was *performance* of subjects for each task and subtask, measured by three variables: *structure*, *behaviour*, and *structure-behaviour*.

Participants. Twenty-two experienced systems analysts and designers, all with more than 2 years of experience in PO analysis and design, and 24 graduate and undergraduate business students with limited prior knowledge of PO modelling but no experience in on-the-job application of these concepts, were participating. Neither group had prior experience in OO analysis and design.

Experiment design. A 2×2 *factorial* design was used, with the two factors being the *experience level* of the subjects and the *nature of the task*. The subjects were divided into two groups. One group consisted of 22 experienced systems analysts and designers. The other group consisted of 24 students. Both groups were provided identical training sessions (two 3 h sessions) on OO analysis and design using the Coad and Yourdon methodology (Coad and Yourdon, 1991a). They were given two experimental tasks and were required to develop OO models for both tasks. Task 1 was an application inherently PO (behaviour) in nature, while task 2 was inherently OO (structure). The order of task presentation was *counterbalanced* in order to eliminate any confounding learning effects.

Results and interpretation. An ANOVA test was used to determine the interaction effects of experience and task characteristics on problem solving performance. Significance level was set at $\alpha = 0.01$. Results for *task-related* hypotheses indicate a non-significant trend toward supporting H_{1a} ($p = 0.057$). H_{2a} was not supported; the experienced group performed significantly better than the inexperienced on the OO task. Considering the *subtask related* hypotheses, where the analysis was broken down to look at the behaviour and structure subtasks, they found that the experienced group performed significantly better for behaviour but not for structure. Therefore, H_{1b} and H_{2b} are supported.

Critique. Our observations are focused first on the lack of training (6 h in total) which could be considered rather minimal. Specifically, in the first attempt the subjects had difficulty distinguishing between objects and attributes. Second, misuse of inheritance occurred where a few methods of the parent classes were hidden Gamma et al. (1995). This is also referred as the 'NOP problem' by Riel (1996), that is overriding an inherited method with an empty one in the child class. Third, concerns the authors' assertion that "although sequencing of processes is an integral construct of the PO modelling paradigm and is represented explicitly through directed data flows, there is no straightforward way of implementing such control in the OO modelling paradigm". We consider that it could be captured by a sequence or interaction diagram. Therefore, the experiment might be regarded as biased towards the PO paradigm.

Agarwal et al. (1996). The *objects* of this study are systems analysis and design using OO and PO methodologies, for the *purpose* of *exploring* whether the OO methodology offers better "cognitive fit" (the match of nature of the *task* and the way it is *represented*) over the PO, in the domain of OO and PO analysis and design, *with respect to* the *effectiveness* and *efficacy* in problem-solving performance, from the *point of view* of the *researcher*. The *context* concerns an experiment run using students as subjects performing on two types of systems analysis and design *tasks* —OO and PO, using two types of modeling methodologies—OO and PO. A task is classified as inherently *structural* (OO) if its description highlights data and structural relationships, and inherently *behavioral* (PO) if its emphasis is on processes and sequencing. The study examines the effects of the interrelationship between task and tool at two levels of granularity: (1) the *task* level (OO and PO) and (2) the *subtask* level (structure and behaviour). The methodology used for OO training was adapted from the one suggested by Coad and Yourdon (1991a, b), while that used for PO training was adapted from DeMacro (1978). Two experimental tasks—narrative descriptions of business information-processing problems—were utilized. Task 1, that was PO in nature, was an "Account Payable System" described in a little over 300 words, while Task 2, that was OO in nature, was an "Employee Benefits System" described in approximately 400 words.

Hypotheses. H_0: There is no difference in quality solutions between users performing on PO or OO tasks, using PO or OO methodologies. H_{1a}: For the PO *task*, users of the PO methodology generate higher-quality solutions of the OO methodology. H_{1b}: For the *behavior subtask*, users of the PO methodology generate higher-quality solutions of the OO methodology. H_{2a}: For the OO *task*, users of the OO methodology generate higher-quality solutions of the PO methodology. H_{2b}: For the *structure subtask*, users of the OO methodology generate higher-quality solutions of the PO methodology.

Variables. Independent variables were the analysis and design methodology (OO and PO). *Dependent variable* was the performance of subjects measured as the overall quality and Jaccard's similarity coefficient (Anderberg, 1973; Everrit and Dunn, 1983).

Participants. Forty-three business students enrolled in an information systems course at a university were used. They had limited prior knowledge of both OO and PO modeling and no experience in on-the-job application of these concepts.

Experiment design. A 2×2 *factorial design* with problem-solving task and problem-solving methodology was used for the experiment. One group, consisted of 24 subjects, used the OO methodology, while the second group, consisted of 19 subjects, used the PO tool. The order of task presentation was *counterbalanced.* A maximum time limit of one and a half hours was allotted for each problem. The overall quality of a solution was assessed using two metrics: a subjective score between 0 and 10 assigned by two independent evaluators, and Jaccard's similarity coefficient, which provided a more objective assessment.

Results and interpretation. Two two-way analysis of variance (ANOVA) procedures—with task and methodologies as the two treatments—were run to examine the task level effects. If the overall ANOVA was significant, *t*-tests were used to test the specific propositions. Results indicate that the interaction effects between task and tool were significant for the overall quality ($p = 0.012$) and weakly significant for Jaccard's similarity coefficient ($p = 0.087$), while at the subtask level, interaction effects were significant for both the PO ($p = 0.018$) and the OO task ($p = 0.001$). The results of the follow up *t*-tests show that subjects who used the PO methodology performed significantly better on overall quality ($p = 0.000$), as well as on Jaccard's similarity coefficient ($p = 0.044$). However, there was no difference in performance using the two tools for the OO task for either dependent variable ($p = 0.270$ for overall quality; $p = 0.652$ for Jaccard's similarity coefficient). Thus, alternative hypothesis H_{1a} was supported by the data while H_{2a} was not supported. *T*-tests for the subtask-level ANOVA produced similar results. For the *behavior* subtask, subjects performed better using the PO methodology across both tasks ($p = 0.000$ for the PO task, and $p = 0.003$ for the OO task). For the *structure* subtask, however, there was no significance in performance ($p = 0.390$ for the PO task, and $p = 0.171$ for the OO task. Thus, H_{1b} was supported while H_{2b} was not supported.

Critique. Our critique is focused on three points. First, we consider the fact that OO techniques adopted do not graphically depict the control or sequencing of processes through a notation such as a sequence diagram, or do not implicitly enforce a sequence as the process model, as working against the OO methodology. Second, we have some concerns over the OO model presented by the authors as a correct representation, where the classes "Asst_Prof", "Assoc_Prof", and "Full_Prof" are subclasses of the "Faculty" class. This may be an example of misused inheritance since it apparently violates the *substitutability* principle, mentioned previously. Third, considering the difference in performance on the relationships for the OO subjects, it suggests a strong indication of the role of graphical representation.

Agarwal et al.'s (1999) *motivation* was two conflicting viewpoints: One suggested by OO proponents, that in addition to the previously described advantages of the OO paradigm, it also lends itself naturally to the way humans think. The other supported by evidence from research in cognitive psychology and human factors suggesting that human problem solving is innately procedural. The *objects* studied are the OO and PO *models*, for the *purpose* of *investigating* whether *problem representation* is a determinant of performance *with respect to comprehension* for the *point of view* of the *researcher*. The *context* concerns two experiments using students as subjects performing on two business application systems (a payroll system—'ABC', and a motor vehicle registration system—'Texas case'), represented both as an OO model (Object model and atomic and meta-models) (Coad and Yourdon, 1991a), and a PO model (DFD, and data dictionary) (DeMacro, 1978).

Hypotheses. H_0*:* There is no difference in understanding structure[4]-oriented aspects and PO aspects of an application represented using an OO model and a PO model. H_1: It is easier to understand structure-oriented aspects of an application represented using an OO model rather than a PO model. H_2*:* It is easier to understand PO aspects of an application represented using a PO model rather than an OO model. H_3: It is easier to understand both structure-oriented and PO aspects of an application together using an OO model or than a PO model.

Variables. The *independent* variables were the *type of models* (OO or PO) and *type of comprehension question* (OO or PO or hybrid). The *dependent* variable was the '*accuracy of comprehension*'.

Participants. Seventy-one undergraduate students, majoring in information systems with some prior experience with PO modelling but without OO experience, participated.

Experimental design. Two experiments were carried out using the two cases described above. The second was a replication of the first with a different set of subjects and a different task to ensure that the results were not biased by any task-specific characteristics. For both experiments, the subjects were randomly assigned to one of two groups: one group received the OO model while the other received the PO model. A total of eight questions were developed for each task. The comprehension questions were classified as structural, behavioural or a combination of both. The quality of comprehension was measured through subject's responses to questions designed along these dimensions. In the first experiment, 18 subjects received the OO model and 18 others received the PO model for the ABC case. In the second experiment, 18 subjects received the OO model and 17 received the PO model for the Texas case.

Results and interpretation. For the analysis of the collected data *t*-tests were used. Significance level was set at $\alpha = 0.05$. Overall, the results suggested little difference, in terms of comprehension accuracy, between the representations for structural or PO models, however, for the more complex combined comprehension tasks the students with the PO notation performed significantly better. The authors speculated that only the combined questions were sufficiently demanding to reveal any differences between procedural and OO notations. Hypotheses testing indicate that H_1, H_2, and H_3 were not supported. However, in H_3 PO model led to significantly better levels of comprehension.

Critique. Our observations concern the solution provided by the authors. They apply a class inheritance hierarchy ('Employee' and its subclasses) in a case where other authors argue that composition ('role modelling') would be better for this type of problem (Coad and Mayfield, 1997; Venners, 1998). Misuse of inheritance occurred in a subclass (GA) where a few methods of the parent classes were hidden (Gamma et al., 1995). This is described in the first experiment. We also note, as the authors did themselves, that other OO notations such as UML may better support comprehension tasks that have combined structural and behavioural aspects. Finally, we observe the existence of some related empirical evidence to authors' motivation of the ways in which humans "naturally" think is provided by Hatton's (1998) case study.

Briand et al.'s (1997) *motivation* in this study was to explore whether OO techniques offer significant advantages over structured techniques given the fact that much of the debate was based upon opinion and anecdote rather than empirical evidence. Additionally, the empirical research prior to this experiment provided scant support for OOT. The *objects* studied are *design techniques* (OO and structured) for the *purpose* of *investigating* their impact on developers' ability *with respect to understandability* and *modifiability* from the *point of view* of the *researcher*. The *context* concerns an experiment run using students as subjects performing on design documents.

Hypotheses. H_0: There is no difference between design documents, in terms of ease of *understandability* and *modifiability*, developed by the use of OO or *structured* techniques regardless of the application of various 'good' or 'bad' *design principles*. The alternative hypotheses were then stated as: It is easier to understand and modify H_1: 'good' OO design than 'good' structured design, H_2: 'good' OO design than 'bad' OO design, H_3: 'good' structured design than 'bad' OO design, H_4: 'good' structured design than 'bad' structured design, H_5: 'bad' structured design than 'bad' OO design.

Variables. The two *independent* variables were the *design technique* used (OO or structured) and the *design principles* applied ('good' or 'bad'). The two *dependent* variables were *understandability* (accuracy of comprehension), captured via means of asking questions about the components of the system designs, and *modifiability* (proportion of correct change locations identified and number of locations/time taken), captured by means of subjects performing impact analyses on the design documents (but not making the changes identified).

Participants. Thirteen student *subjects* with limited experience were used.

Experiment design. Four different design documents were provided to subjects. Two were OO and two were structured. The OO designs were designed using the OMT methodology Rumbaugh et al. (1991). For the structured designs, MIL/MDL based on DeRemer and Kron (1994) was used. The applicable design principles used included guidelines on coupling, cohesion, clarity of design, generalisation/specialisation, and keeping objects and classes simple, identified by Coad and Yourdon (1991a, b). For each paradigm one document was considered 'good' and one 'bad' according to the design principles listed above. Subjects were randomly assigned to one of four groups. A 2×2 *factorial design* in two blocks of size two was employed (*counterbalancing*). This has received some adverse comment since the meaning of good and bad design differs between the structured and OO paradigms. It is therefore arguable that the experimental design is *hierarchical* as opposed to factorial.

Results and interpretation. For the analysis of the collected data an ANOVA test was used. The significance level was set at $\alpha = 0.1$. Hypotheses testing indicate that: H_1 is not supported, H_2 is supported, H_3 is supported (modifiability, not understandability), H_4 is not supported, and H_5 is supported (understandability, not modifiability). Results from this experiment strongly suggest that the quality principles embodied in the "good" design collectively have a beneficial effect on the maintainability of OO design documents. However, there is no strong evidence regarding the alleged higher maintainability of OO over structured design documents. Furthermore, their results suggest that OO design documents are more sensitive to poor design practices than structured design documents. In addition, the authors draw three further conclusions. First, proper training and climbing the OOT learning curve should be a crucial activity if significant maintenance benefits are to be achieved. Two, adhering to quality OO design principles is important if the promised OO benefits are to be realised. Three, abuse of OO architectural guidelines add significantly to cognitive complexity. Consequently, it may be even more important to follow stringent quality standards when using OO design techniques.

Critique. Our observations regarding this study are that it examines very interesting aspects of software design, although the number of participants used was small with a possible effect on results. The fact that the design principles examined are not operationally defined and their application requires a certain degree of subjective interpretation indicating the need for further research in this direction.

Corritore and Wiedenbeck (2000): The *object* of this study is to analyze how *comprehension-related activities* evolve in successive maintenance episodes on the same program, and how the information gathering and comprehension of OO and procedural programmers differ, for the *purpose* of *investigation*, *with respect to* the *scope* and *direction* of *comprehension*, from the *point of view* of the *researcher*. The *scope of comprehension activities* refers to the breadth of familiarity with the program, gained by the programmer during comprehension activities, defined as the proportion of files accessed. The *direction of activities* concerns whether the strategic approach to program comprehension is top-down, bottom-up, or a mixture of the two. The higher-level abstraction refers to the *domain level* (documentation files) and the low-level abstraction refers to the *program model* (implementation files). Accessing the more abstract documentation and header files was interpreted as reflecting the use of *top-down* process and accessing the less abstract implementation files was seen as reflecting a *bottom-up* strategy. The *context* concerns an experiment run using professionals as subjects performing on the documentation and code of two functionally equivalent versions of a database program for a small airline, written in OO C++ and procedural C. The C++ version of the program made extensive use of the OO features of inheritance, composition, encapsulation, and polymorphism. Both programs were similar in length, C++ 822 lines versus C 783 lines. The program and all supplementary materials were presented on-line in a graphical Unix environment. The most notable difference was that, naturally, there were no inheritance hierarchy charts for the procedural paradigm.

Hypotheses. H_0: There is no difference between OO and procedural experts in the *direction* and the *scope of comprehension activities,* in program understanding during maintenance. H_1: OO experts show a more top-down direction of comprehension activities than procedural experts. H_2: OO experts have a narrower scope of comprehension activities than procedural experts.

Variables. *Independent* variables were *programming paradigm* (OO or procedural, represented by the C++ respectively C language), *file type* (documentation, header or implementation), and *activity* (program study, modification 1 and 2). The *dependent* variable was the mean proportion of files accessed.

Subjects. Thirty professional programmers participated. Fifteen were OO C++ and 15 were procedural C programmers. All but two had post-baccalaureate degrees. Twenty-seven held their highest degree in computer science or engineering. On average, they had been programming for 11.6 years with a range of 2.5–20 years.

Experiment design. The study was conducted as a *multi-test within object* study. Each participant was run individually in 2-h sessions that were held seven to 10 days apart. In the first session (program study) the participant studied the program for 30 min, followed by a short modification task. Data were not collected from it. Two modification tasks (modification 1, and 2) had to be performed during the second session. The order of presentation of the modifications was *counterbalanced. Blocking* and *balancing* were the design principles.

Results and interpretation. The statistical hypotheses were tested using Analysis of Variance. Follow-up analysis was carried out using ANOVA and Tukey's HSD. Significance level was at $\alpha = 0.10$. Considering H_1, during the *study phase*, the OO participants accessed significantly more documentation files than procedural participants, indicating a top-down direction of comprehension activities ($p < 0.05$). In *modification* 1, across both paradigms, participants accessed more implementation files than documentation files ($p < 0.05$). In *modification* 2, the direction of comprehension activities for both paradigms was bottom-up ($p < 0.05$). Therefore, H_1 was not supported. Considering H_2, the ANOVA indicated a significant main effect of paradigm. Namely, OO participants accessed significantly fewer files than procedural participants, indicating a narrower scope of comprehension activities for OO participants ($p < 0.05$). Therefore, H_2 was supported.

Critique. We consider the experiment to be a well-designed one. In particularly, the use of professionals is its major advantage. However, an important point is the lack of a specific design method and notation. Also, concerning the documentation of the programs, the authors statement that "the most notable difference between the two paradigms were that the procedural paradigm was not using inheritance hierarchy charts". We don't consider this reflects reality and comes in disagreement with their previous reference to the extensive use of OO features.

Henry and Humphrey's (1990): *motivation* was that due to a lack of scientific evidence there was only an intuitive feeling, between software engineers and managers, regarding the accuracy of claims that systems developed with OO languages were more maintainable than those programmed with procedural languages. The *objects* studied are OO and *procedural languages* for the *purpose* of *investigation with respect to maintainability* from the *point of view* of the *researcher*. The *context* concerns an experiment run using students as subjects performing on two systems developed by two languages representing two different paradigms.

Hypotheses. H_0: There is no difference in ease to maintain systems implemented with an OO language than those using a structured language. 'Ease to maintain', in this context, was measured by the time programmers took to perform a maintenance task, and by the number of changes to the code the task required. It also meant programmer perception of the ease of change. The alternative hypotheses were then stated as: It is easier to maintain OO programs than structured programs; H_1: OO programmers take less time to perform a maintenance task, H_2: OO maintenance requires fewer changes to the code, H_3: OO programmers perceive the changes as conceptually easier, and H_4: OO programmers make fewer errors during the maintenance task.

Variables. There were four *independent* variables: *subject* (the student identifier), *group* (A or B), *programming language*, and the *modification task*. The *dependent* variables were *maintenance times*, *error counts*, *change counts*, and *programmers' subjective impression*.

Participants. Twenty students participated.

Experiment design. The experiment was a *within subjects test*, using a *counterbalancing* procedure, where the subjects were requested to perform *enhancement maintenance* tasks. Two modification tasks were simulated requests from users to make functional changes to the system. The tasks were conducted over an 11 weeks period on two functionally identical programs, which were based on a 'laundry-list' handler system. Both systems used 15 files (modules) each, comprising a total of approximately 4000 lines of code for each system. The first one was designed with structured design techniques using a procedural language C, and the other designed with OO design techniques using an OO language (Objective C). Subjects actually performed each task twice. Finally, the subjects completed a post-experimental *questionnaire*.

Results and interpretation. This experiment supports the hypothesis that subjects produce more maintainable code with an OO language than with a PO language. For source code variables, Objective-C produces code that requires fewer modules and sections to be edited, fewer lines of code to be changed and added. The authors state that this leads to the conclusion that Objective-C produces fewer changes that they are more localised than procedural C. Additionally, C is never better than Objective-C for any variable in this study.

Critique. We consider that there is concern, however, that there may be learning effects due to the ordering of the two tasks. The Objective-C language is superset of the C language. There is also a question regarding the counting of maintenance time, since no constraints were imposed—other than the 11 weeks period—and so relies entirely upon the accuracy of subjects reporting minutes of thinking time and so forth. We also note that inexperienced subjects were used and design documentation was not provided.

Lee and O'Keefe (1996): The objects studied are the OO and the *structured rule-based* systems, for the *purpose* of *evaluating* and *comparing* the *knowledge representation* (KR) techniques *with respect to* the *maintainability* from the *point of view* of the *researcher*. The *context* concerns an experiment run in a PC lab, using students as subjects performing on two versions of a system dealing with the 'animal classification problem'. They were asked to modify the knowledge-based (KB) schema of the two versions based on new pieces of animal knowledge using the NASA development tool CLIPS version 5.0.

Hypotheses. H_0: There is no statistical difference in KR maintenance functions between modularized production systems and OO systems. H_1: It will be more effective to maintain an OO system than a modularized production system (representation effect $>$ structural effect). "Effective to maintain", in this context means that programmers take less time to perform a maintenance task with more confidence in their outcomes, and/or the implementation of the changes is more accurate.

Variables. The *independent variables* are the *knowledge representation* (KR) techniques and the *programming paradigm* (OO or structured rule-based). *Dependent* variables are: *time* taken to implement a change, the *accuracy* of the modification, and confidence in the *correctness* of the work.

Participants. Twenty graduate students were used. During a course they received 5 weeks (15 h in class and 10 h in laboratory) of KB systems development training (including KR, inferencing, development methodologies, modularization of rules, and CLIPS.

Experiment design. Subjects were randomly assigned to two groups. The *within-subjects* procedure was used with *counterbalancing*. The experiment lasted 90 min.

After the maintenance tasks were complete, a post-experimental questionnaire was administered to gather information about the following: (1) the maintainers' confidence in their maintenance functions, (2) perceptions of the relative complexity of the system tasks, and (3) the usefulness of documentation.

Results and interpretation. The paired comparisons *t*-test was used, since the experimental design is the "subject as his/her own control" design. Significance was set at $\alpha = 0.05$. Three subjects failed to finish the maintenance tasks and so withdrew from the experimental, since they could not understand the systems. Thus, the total sample size was reduced of 20–17. The results indicate: (1) that for maintenance time, OO subjects performed significantly faster than the rule-based ones ($p < 0.01$); (2) the implementation of new knowledge was on average better for the rule-based system than the OO system, but the difference was not statistically significant. One possible explanation, stated by the authors, is that default reasoning through property inheritance makes it more difficult to verify the OO KB than the rule-based KB, where the logical structure is more transparent. (3) The participants' reports on: (a) their confidence in the correctness of their modification, indicate no statistical significance, (b) their perceived complexity, indicates with statistical significance, that the rule-based system was perceived as more complex than the OO system ($p < 0.01$), (c) regarding the usefulness of documentation they ranked the OO system, due to its embedded structure, as significantly more useful than the rule-based one ($p < 0.01$).

Critique. First, we observe that the experiment deals with a small artefact. Also, due to counterbalancing, possible learning effects could have biased the results. However, regarding the OO paradigm, we would like to place emphasis on two problems both originating from the inheritance mechanism: (a) the misplacement of methods in the inheritance hierarchy, and (b) misclassification due to conceptual misunderstanding.

Lewis et al. (1991): The *objects* of this study are the OO and the *procedural paradigms* for the *purpose* of *exploring* whether the OO paradigm offers significant advantages over the structured *with respect to reuse* of previously developed components, for the *point of view* of the *researcher*. The *context* concerns an experiment run using students as subjects developing a specified target system implementing a 'fictional company'.

Hypotheses. H_0: There is no difference in *productivity* due to *reuse* of software components, between systems implemented using an OO language and those using a *procedural* language. H_1: The OO paradigm promotes higher productivity than the procedural one. H_2: Reuse promotes higher productivity than no reuse. H_3: The OO paradigm promotes higher productivity than the procedural paradigm when programmers do not reuse. H_4: The OO paradigm promotes higher productivity than the procedural paradigm when programmers reuse H_5: The OO paradigm provide incentives to reuse above those of the procedural one.

Variables. The *independent* variables were the *programming language* (C++ and Pascal) and *reuse*. The *dependent* variable was *productivity*.

Participants. Twenty-one senior-level software engineering students participated.

Experiment design. Two sets of potentially reusable components were designed and implemented by independent programmers. One set was implemented in a procedural based language, Pascal, and the other in an OO language C++. Each component was designed to have a specific level of applicability. The *levels of reuse* were described as: (1) completely reusable, (2) reusable with slight revision (< 25%), (3) reusable with major revision (>25%), and (4) not applicable for reuse. All completed projects were verified to meet a set of requirements concerning documentation, code quality, and functional correctness. The subjects were *randomly* divided, but were statistically *blocked* across their computer science grade point averages, into four groups. Half of them implemented the projects in Pascal, the other half in C++. Furthermore, part of each language group were not allowed to reuse at all, while the others were encouraged to reuse. The "no reuse" groups serves as control groups.

Results and interpretation. An ANOVA test was used. The significance level was set at $\alpha = 0.05$. Productivity was defined as the inverse of the effort expended to produce a specific software product. It was captured by the following number of measures during system development and testing:

Runs—The number of runs made,
RTE—The number of run time system discovered,
Time—The time (in minutes) to fix all run time errors,
Edits—The number of edits performed, and
Syn—The number of syntax errors.

The Runs, RTE, and Time variables, given their significance to the development process, are considered the main variables of interest. The other two are gathered for completeness, but are given less emphasis. Hypotheses testing suggest the following. H_1, H_4, and H_5 are supported by the three main variables. H_2 is strongly supported by all variables. H_3 is not supported. Therefore, the results indicate that the OO paradigm is well suited to reuse.

Critique. This study investigates a factor of major significance—reuse—for software development. However, it would have been of interest to know to what extent there has been use of the central OO reuse mechanism, namely, inheritance.

Moynihan's (1996) *motivation* was the claim that one of the major 'selling-features' of the OO approach is its use of a uniform set of concepts across the development process, thus supporting verification, traceability, and reuse. The *objects* studied are OO and *functional-decomposition paradigms* for the *purpose* of *comparing* them for *communicating system functionality* to users *with respect to effectiveness* from the *point of view* of the *researcher*. The *context* concerns a simple experiment run using business managers as subjects performing analyses of a fictitious system.

Hypotheses. H_0: There is no difference between the functional-decomposition model and the OO model in effectiveness for communicating system functionality to users. The alternative hypotheses were then stated as: A functional-decomposition model provides more effectiveness than OO model for communicating system functionality to users. H_{1a}: In *Task-performance*: (i) identifying missing 'seeds' from the analysis, (ii) commenting on the apparent omissions in the analysis, (iii) making comments of a more strategic nature. H_{1b}: *Examining the style*: (i) understanding notation and concepts, (ii) helping in detecting incompleteness and internal inconsistency, (iii) provoking comments and questions, (iv) giving a holistic understanding of the application domain, (v) evaluating implementation benefits and priorities, (vi) visualizing an implementation of the system.

Variables. *Independent variables* were the *type of model* (OO or functional-decomposition). *Dependent variable* was the *effectiveness* for communicating system functionality to users.

Participants. Twenty (non-technical) executives attending a management development programme participated as subjects. None were information systems specialists.

Experiment design. For model presentation the Object Model Notation (Rumbaugh et al., 1991) was used for the OO model while Schema I (Martin, 1992) was used for the functional-decomposition model. The subjects were presented with the preliminary analysis of two versions (OO and functional-decomposition) of the same system. Discrepancies from a natural language document had been deliberately introduced. They were then asked to evaluate the models. The *effectiveness* was operationalized in terms of subjects' *task-performance* and in terms of subjects' *opinions* on the two 'styles' of analysis. The task-performance was measured by: (i) the number of 'seeds' the subject mentioned as being missing from the analysis, and (ii) the 'quality' of the subjects' critique on the content of the analysis. The subjects' opinions on the relative merits of the two *styles* of analysis were obtained by means of a post-experimental questionnaire.

Results and interpretation. *Fisher's Exact Test* at level $\alpha = 0.05$ was used. The results indicated that for *task performance* there appeared to be no evidence that the seed detection rate varied between the two notations. Likewise for eliciting detailed comments. However, the functional decomposition was significantly better for eliciting strategic level comments. Examining the *style* there was significant evidence on the first, third, fourth and fifth attributes. While no significance effect was found on the second and sixth attributes. Overall subjects stated that they found the OO model harder to understand, gave less of a holistic understanding and was less effective at helping them to evaluate likely implementation benefits and priorities.

Critique. We note the lack of technical expertise of the subjects and also the possibility of ordering effects arising from the presentation of both models to each subject although the ordering was reversed for half the subjects.

Wiedenbeck et al.'s (1999) motivation was the educational implications deriving from the switch from procedural to OO languages. Their particular emphasis was on program comprehension skills since this forms the underpinnings for many software engineering activities. The *objects* studied were the OO and *procedural* paradigm for the *purpose* of *investigating* the *mental representations* formed by novices in the two styles *with respect to program comprehension* from the *point of view* of the *researcher*. The *context* concerns two experiments run using students working with a program used in pedagogy. However, we exclude the first one from our review since it deals with a very small and trivial program (less than 10 LOC). The investigation was based on Pennington's model (Pennington, 1987a, b) which refers to the two levels of representation as the *program model* (textural representation) consisting of elementary operations and control flow information, and the *domain model* (situation or mental model) consisting of data flow and program function. Pennington argues that it is a key element of an expert programmer's plans, or "schemas", that represent their stereotypical knowledge of common programming tasks.

Hypotheses. H_0: There is no difference between styles of programming in affecting the comprehension of novice programmers, to be rejected by the alternative hypotheses. H_1: Different programming styles (OO or procedural) do affect the comprehension of novice programmers as regards the (a) *language* (OO or procedural), (b) *viewing* (described below), and (c) *question type* (operations, control flow, data flow, function).

Variables. The *independent* variables were the *type of paradigm*, *programming language* (C++ or Pascal), *study/question answering*, and *comprehension questions*. The *dependent* variable was *program comprehension*.

Participants. There were a total of 86 student *subjects*, with low but approximately similar programming experience.

Experiment design. The subjects were provided with either a program written in C++ (188 LOC) or Pascal (140 LOC). The programs used were a simplified implementation of a well-known example used in pedagogy, 'the game of life' simulation, where the original C++ program from which the OO version for this study was derived can be found in a textbook by Pohl (1993). The subjects were required to answer a set of 16 comprehension questions that targeted the four information categories forming part of the mental presentation: *elementary operations*, *control flow*, *data flow* and *function*. They participated in two study/query answering trials, referred to as *viewing 1* and *viewing 2*. In viewing 1 they first studied the program listing for 15 min, after which the program was removed and they answered questions. In viewing 2 they were given an additional 15 min to answer the same question set again, this time with the program available to them.

Results and interpretation. A *three-way mixed-model ANOVA* was used for the analysis of data collected. The *between-subjects* factor was *language* (OO or procedural). The *within-subjects* factors were *viewing* (1 or 2) and *question type*. Comprehension was measured by the percentage of correct answers. The ANOVA showed that there was a significant main effect of: (a) language, with the procedural group scoring higher than the OO group overall ($p < 0.0005$), (b) viewing, with viewing 2 higher than viewing 1 scores ($p < 0.0005$), and (c) question type ($p < 0.0005$). Therefore, considering comprehension the procedural subjects clearly performed better than the OO subjects. Possible explanations from the authors of the poor performance of OO subjects include the following. First, the information regarding function and control flow in an OO program was not highlighted by the notation. Second, the distributed nature of control flow and function in an OO program (i.e. a delocalised plan) made it more difficult for novices to form a mental representation. Third, the OO subjects faced a learning curve. Fourth, Pascal was designed as a teaching language, whilst C++ was not.

Critique. Our observations first concern the lack of explicitly stated hypotheses forcing us to extract them from the presentation of the results. Second, the program used, particularly the OO version, was a simplified implementation of a C++ program published in a text book, which might have been read by the subjects and could have led to bias towards to OO paradigm. Lastly, other documentation (e.g. data and control flow diagrams) was not provided for the subjects.

3.2. OO Design Principles

Briand et al. (2001): This research builds upon the results of a previous experiment (Briand et al., 1997) run by the same authors, which investigated the benefits of the quality design principles by Coad and Yourdon (1991a, b) *with respect to maintainability* of OO and structured design documents. This study concentrates solely on the investigation of the use of quality design principles and their influence on a developer's ability to understand and modify OO design documents. The context concerns two experiments run using students as subjects performing on design documents implementing two, but equivalent in complexity, systems.

Hypotheses. H_0: There is no significant difference between 'good' and 'bad' OO design documents in terms of ease of understanding and impact analysis. The alternative hypotheses were then stated as H_1: 'Good' OO design in significantly easier to understand than 'bad' OO design, H_2: It is easier to perform impact analysis (locate changes) on a 'good' OO design than on a 'bad' OO design.

Variables. The two *independent* variables were the *experimental run* (run 1 and 2) and the *design principles*. The *dependent* variables were the following six: (1) time spent on understanding the system in order to complete the questionnaire, (2) correctness of the understanding questionnaire, (3) time spent identifying places for modification, (4) completeness of the impact analysis, (5) correctness of the impact analysis, and (6) modification rate—the number of correct places found per time unit.

Participants. Thirty three students with little experience enrolled in a semester long class on Software Engineering participated.

Experiment design. Two OO design documents were used to test the hypotheses. Each design documentation was approximately 30 pages including a general system description and a requirements document. There were two versions of each document: one was compliant to the design principles and another violated them. To quantify the degradation applied to 'bad' design, counts of design artifacts and relationships as well as standard design measures were used. The degradation introduced, on average, more coupling and more (inappropriate) inheritance. The application domains used were (i) good system—a temperature controlling system and (ii) bad system—an automatic teller machine. For each system document there were two sets of tasks to be performed, (i) to complete a questionnaire concerning understanding and the structure of the design and (ii) to undertake two separate impact analysis (mark all the places to be changed and enhancement of system functionality). After completion of the tasks a debriefing questionnaire had to be completed. A standard *within-subjects* 2×2 *factorial design* was employed with *counterbalancing*. The subjects were randomly assigned to two groups.

Results and interpretation. The *Wilcoxon matched pairs test* was applied. Significance level was set at $\alpha = 0.05$. Results for H_1—ease of understanding—indicate no significance for the first variable while statistical significance was indicated for the second variable ($p = 0.00$). Considering H_2—ease of impact analysis—there is statistical significance for the second ($p = 0.00$) and fourth ($p = 0.02$) variables, practical significance for the third variable ($p = 0.07$), and no significance for the first variable.

Critique. We consider the fact that 13 subjects for the bad design did not follow the instructions, resulting in no separation between the time spending on understanding and on the impact analysis, could have biased the results. Also, the fact that a few questions examining the cohesion principle, due to lack of information at a high level of design, require a certain degree of subjective interpretation, indicates the need for further investigation in that direction. Lastly, we agree with the authors' suggestion for the necessity of investigation of each and every design principle independently, which could lead to better understanding of tradeoffs and impacts between various design aspects.

3.3. Inheritance Related Experiments

Cartwright's (1998) experiment is a small replication of that of Daly's et al. (1996), based on the same research question concerning the impact of class inheritance. However, she used only two versions of the bibliographic database program. Subjects were requested to make the same maintenance change but on a three level inheritance depth version and a 'flat' version.

Hypotheses. H_0: Three levels of inheritance depth compared with no inheritance have no impact upon the time to make a correct maintenance change, and respectively the size of a correct maintenance change, to be rejected in favour of the alternative hypotheses; Maintenance changes to a program with zero compared with three levels of inheritance are significantly different in amount of H_1: time to make a maintenance change, H_2: size of a maintenance change.

Variables. The *independent* variable was *inheritance depth* (zero and three levels). The *dependent* variables were the *time* to make a maintenance change and the *size* of maintenance change.

Participants. Ten students were used with a minimum of 6 months experience of C++.

Results and interpretation. To explore the hypotheses a *two-tailed unpaired t* test was used. Significance level was set at $\alpha = 0.05$. Results indicate that three levels of inheritance depth have a significant positive effect upon the time to make a change, thus supporting H_1 ($p = 0.0449$) and a significant negative effect upon the size of a change in Lines of Code supporting H_2 ($p = 0.0007$). Therefore, The responses from the debriefing of the subjects left the impression that the inheritance-based structure was harder to understand. Also, changes for the inheritance treatment were more compact but took longer to perform.

Critique. We consider that a small artefact was used and the number of subjects was very small. Additionally, the two least experienced subjects participated in the inheritance treatment, which could have skewed the results because the number of subjects was so low.

Daly et al.'s (1996) *motivation* arose from the results of their structured interviews and questionnaires of OO maintenance staff. This suggested that inheritance depth affected a programmer's ability to understand OO software, and that between four and six levels of inheritance depth was where difficulties began. The *object* studied is the *inheritance* mechanism in the OO paradigm for the *purpose* of *investigating* the impact of its *depth with respect to maintainability* for the *point of view* of the *researcher*. The *context* concerns two experiments and an internal replication run using students performing on C++ programs.

Hypotheses. H_0: The use of zero, three and five levels of inheritance depth does not affect the maintainability of OO programs, to be rejected in the favour of the alternative hypotheses H_1: The use of three levels of inheritance does affect the maintainability of OO programs (stated for the *internal replication* too). H_2: The use of five levels of inheritance does affect the maintainability of OO programs—subjects maintaining the inheritance program version will take longer than those maintaining the 'flat' version.

Variables. The *independent* variable was the *depth of the inheritance* (zero, three and five) hierarchy and the *dependent* variable was the *time* taken to complete the maintenance task.

Participants. Thirty-one students and recent graduates, without OO experience, participated.

Experiment design. To test the hypothesis a *within subjects randomised block design* between two groups with *counterbalancing* procedure was used. In the first experiment two programs were used, one implementing a university database system and another implementing a library database system. Two versions of each system of about 390 LOC in size for the 'flat' version and about 360 LOC for the inheritance version (including comments) of three levels inheritance were used. In the second experiment a larger version of the university system was used. Its inheritance depth was five levels for the inheritance version of approximately 800 LOC distributed in 12 classes, while the 'flat' version had three fewer classes but was around 300 LOC longer. The *tasks* were extensions with a new class for every model using C++.

Results and interpretation. Data collected was calculated using a *Wilcoxon signed ranks* test. Significance level was set at $\alpha = 0.05$. Results indicate that a third of the subjects failed to complete at least one of the tasks within the allotted time for the first experiment. The null hypothesis was rejected in favour of the alternative for the first experiment and the internal replication, and rejected for the second. The authors assert that subjects' relative performance deteriorated on the inheritance program version with a deeper hierarchy because of the increased search problem which was exacerbated by the fact that they were not provided with a conceptual model of the domain nor supplied with a strategy (Wohlin et al., 2000) on which to base a selection of superclass or copy template. While there was a considerable agreement on choice of superclass in the first experiment and its replication, in the second experiment there was a disagreement indicating a manifestation of conceptual entropy (Dvorak, 1994). They also believe that the inheritance mechanism, due to delocalisation, affects program understanding, and hence the deterioration of maintenance performance in deeper hierarchies.

Critique. Our opinion is that the authors failed to specify a purpose for their database making it difficult for the subjects to properly understand the system other than in a highly abstract fashion. The subjects may not be representative of software professionals as they lacked experience. The software systems used were simple both in size and structure. The tasks applied were simple and captured only a part of the maintenance process namely, extension. In order to perform a complete maintenance process other tasks like modifications and additions of data and/or operation members might be applied. The lack of a conceptual model may also affect the performance of developers seriously. The principle of *substitutability* was violated, where for example, 'Director' inherits from 'Technician'. Coad and Mayfield (1997) suggests a more flexible solution to this example by using composition. This means the inheritance structures are somewhat flawed which makes the results of the experiment more difficult to interpret. Yet another concern is the artificial flattening of the inheritance hierarchy. If designers were trying to avoid inheritance it is most unlikely that they would proceed in this fashion.

Harrison et al.'s (2000) experiment, which was another replication of Daly et al.'s (1996), attempted to determine whether the depth of inheritance has an effect on the modifiability and understandability of OO software.

Hypotheses. H_0: The depth of inheritance does not affect the modifiability and understandability of OO systems, to be rejected in favour of the alternative hypotheses. H_{1a}: Three levels of inheritance depth doe*s* affect the *modifiability* of OO systems. H_{1b}: Five levels of inheritance depth does affect the *modifiability* of OO systems. H_{2a}: Three levels of inheritance depth does affect the *understandability* of OO systems. H_{2b}: Five levels of inheritance depth does affect the *understandability* of OO systems.

Variables. The *independent* variable was *type of system* (zero, three and five levels of inheritance). The *dependent* variables were *understandability* (accuracy of comprehension) and *modifiability* (proportion of correct change locations identified).

Participants. The participants were 48 students from the second year of a BSc computer science course. All of them had an experience of at least 18 months developing C++ software although none on-the-job experience.

Experiment design. A 4×12 *between subjects design in four blocks of twelve* was employed. Two systems were used for the experiment. The 'flat' version of the first system contained around 360 LOC while the inheritance-based version contained around 390 LOC. The 'flat' version of the second system contained around 1200 LOC while the inheritance-based version contained around 900 LOC. Subjects were provided with paper copies of the questionnaire and source code. The tasks were to successfully identify the outputs, the affected classes, and the changes needed.

Results and interpretation. χ^2 testing was performed to analyse the data collected. Results indicated that for the 'flat' versus three levels of inheritance, subjective understanding was found to be significantly improved for the 'flat' version at the 0.05 level. Likewise for modifiability, the 'flat' version was easier to modify at the 0.10 level. For the 'flat' versus five levels of inheritance, subjective understanding was *not* found to be statistically significant. Both versions were therefore found to be equally difficult to understand. For modifiability, the 'flat' version was found to be significantly easier to modify than the five levels at the 0.10 level. The authors assert that their analysis indicates that 'flat' structures are easier to modify while the size and functionality of a system has a greater impact on its understandability than the amount of inheritance used.

Critique. Our criticism is similar to that of Daly et al.'s (1996) experiment. The tasks applied were simple and due to the lack of some design documentation a certain degree of caution is wise before drawing conclusions.

Unger et al.'s (1998) experiment was based on that of Daly et al. (1996), but changed certain parameters in order to increase external validity. They used three versions of the same program within one experiment, i.e., 'flat', three and five levels of inheritance depth. Students performed on a program that was an interactive application for displaying two different kinds of stock exchange data in various ways, written in Java instead of C++.

Hypotheses. H_0: Providing that sufficient information was made available about inheritance tree, there is no difference in maintainability of programs with no ('flat'), three and five levels of inheritance, as measured by effort and quality of delivered solutions, to be rejected in favour of the alternative hypotheses; H_1: Programs with more levels of inheritance will require *less time* for maintenance, H_2: programs with more levels of inheritance will result in a *higher quality* of maintenance.

Variables. The *independent* variable was the *inheritance depth*. The *dependent* variables were the *time* required for each assignment (measured in minutes) and the *quality of the delivered solutions* (measured subjectively by a graded scale of correctness).

Participants. Fifty-seven graduate and 58 undergraduate students with a considerable experience in different languages, and in OO programming in particular, were involved.

Experiment design. A *matched-between subjects design* was used. The subjects were randomly assigned to one of three groups ordered by expected performance. The experiment was performed twice with small changes. The first time there were 57 graduate student participating and the second time, the replication, 58 undergraduate student were participating. The maintenance tasks were: (a) addressing the Y2K problem and (b) adding a new class. Actual polymorphism was not used in any of the versions. The five-level version was the original version. The three-level version was derived from the five-level version according to the OO design principle of inheriting only interfaces, not implementations (Gamma et al., 1995 p. 18), consequently, subclasses were derived only from abstract classes, never from concrete classes. The 'flat' version was constructed by duplicating methods and instance variables from the five-level version. The authors claim that the three versions are not arbitrary variants of one program, but present three different, but sensible design styles. In our opinion this is not self evident for the 'flat' version. Documentation was provided using the OMT notation (Rumbaugh et al., 1991). Subjects decided for themselves when they considered their solution to be complete.

Results and interpretation. *One-side*, *paired-wise statistical tests* using *Bootstrap resampling percentiles* were performed to analyse the data collected. The significance level was set at $\alpha = 0.1$. Results indicate that deeper inheritance hierarchies did not generally speed up maintenance, nor did they result in superior quality. Therefore neither alternative hypothesis was accepted. The authors asserted that inheritance depth in itself was not an important factor for maintenance effort. Probably the real advantages of inheritance became visible only for tasks involving complicated functionality modifications where less inheritance meant a higher degree of code duplication and hence higher effort for changes and consistency checking.

Critique. We believe that the design documentation offered to the subjects, as well as the larger sized programs created closer conditions to those experienced by professionals. However our criticism is focused on two things. First, the quality of the solutions was only measured by correctness. We would expect that suggested design solutions should be examined too. The second problem is that the three-level inheritance program was obtained merely by collapsing the five-level version into three levels. However, one would not set out to design a system in this fashion and the redundancy due to duplication on methods is obvious. Conclusions drawn comparing the performance of the two experimental groups are also difficult because of lack of real match between the two parts of the experiment.

3.4. Design Patterns

Prechelt et al's (2001) *motivation* arose from the fact that due to given popularity of design patterns (Gamma et al., 1995) and their claimed advantages, one could expect that they will often be used in situations where their flexibility is not needed: The pattern solves the problem but is more powerful than required. The *objects* are to assess *designs using patterns versus alternative designs* for the *purpose* of *investigation with respect to differences in maintainability* for the *point of view* of the *researcher*. The *context* concerns an experiment run using professional software engineers as subjects, based on four different programs with different design patterns. Two different baseline program versions are used: Version PAT applies design patterns whereas version ALT employs a simpler solution that exhibits only the functionality and flexibility actually required. Design patterns used were: ABSTRACT FACTORY, COMPOSITE, DECORATOR, FAÇADE, OBSERVER, and VISITOR as described in the Gamma et al. (1995) book.

Hypotheses. Hypotheses that are spelled out informally as *expectations*, take the form: "a design pattern P does not improve performance of subjects doing a maintenance exercise X on program A (containing P) when compared to subjects doing the same exercise X on an alternative program A (not containing P)".

Variables. The *independent variables* are the *programs and change tasks*, the *program version*, and the *amount of pattern knowledge*. The *dependent variables* are *time* and *correctness*.

Participants. Twenty-nine professional software engineers were involved, having on average 4.1 years work experience and an average C++ experience of 2.4 years. Fifteen subjects already had some pattern knowledge.

Experiment design, results and interpretation. The subjects were assigned into four groups using *random block assignment*. A pretest (PRE) was run first, followed by a course teaching a number of design patterns. Then the posttest (POST) was run. The subjects received all documents printed on paper and their solutions were delivered in handwriting. For the data test *analysis of variance* (ANOVA) was used. For further analysis *distribution-free Bootstrap* methods was also used. *Correctness* was measured whether participant's solutions fulfilled the requirements or not. Following, each program's definition, expectations and results are described.

Stock Ticker is a program for directing a continuous stream of stock trades from a stock market to one or more displays that are also part of the program. Both versions of it consist of seven classes. The PAT version has 343 lines (including comments and blank lines) and contains an OBSERVER pattern. The ALT version has 279 lines. *Expectations*: (E1) When subjects lack knowledge of the OBSERVER pattern (in the pretest), they have to find out how the pattern works, thus PRE-PAT subjects should require more time than PRE-ALT ones. (E2) Given sufficient pattern knowledge, on the other hand, the PAT group may understand the program structure more quickly than the ALT group. Results support E1 ($p < 0.001$).

Boolean Formulae program contains a library for representing Boolean formulae (AND, OR, XOR, and variables) and for printing the formulae. The PAT version uses the COMPOSITE and the VISITOR patterns and consists of 11 classes spanning 470 lines. The ALT version consists of eight classes spanning 374 lines. *Expectations*: (E1) In principle it should be easier to create a single new class similar to another rather than adding a method to several classes. Since the VISITOR pattern is quite difficult to understand, we expect it will take more time for the PAT groups to find where to add the methods. (E2) Gaining pattern knowledge should help both groups because even in the ALT program a COMPOSITE is present. Results indicate that POST-ALT group is faster that the POST-PAT group, confirming a part of E1.

Communication Channels is a wrapper library program. The PAT version is designed with a DECORATOR pattern in six classes spanning 365 lines. The ALT version spanned 318 lines. *Expectations*: Two influences of the pattern on subjects' behavior are expected. First, due to its multiple decorator classes, delocalized behavior should be slower than the localized one. Second, since the new functionality is encapsulated in classes one need hardly care about mutual influences. Thus, it is expected the second influence to be stronger than the first and hence the PAT version to be preferable (E1), especially at higher levels of pattern knowledge (E2). The different states and its operations having different result codes that are spread over the different decorator classes. It is easier for the more localized ALT program with respect to both work time (E3) and correctness (E4). The PAT group will take longer (E5) and commit more errors (E6). Results support E1 ($p < 0.001$), the pattern-solution is also superior in terms of correctness, E3, E5 ($p < 0.001$) and E6.

Graphics Library contains a library for processing simple types of graphical objects. The PAT version contains the ABSTRACT FACTORY and COMPOSITE patterns and consists of 682 lines in 13 classes. The ALT version consists of 663 lines in 11 classes. *Expectations*: The main difference in time required for the maintenance task will be caused by program understanding. (E1) It is expected the simpler ALT program to be easier to understand. (E2) Pattern knowledge will help both groups because of the COMPOSITE structure in both programs. (E3) Since the structure of both programs is quite similar in the region of interest, it is not expected to observe significant differences between the ALT and the PAT groups. (E4) A difference between PRE and POST is expected due to pattern knowledge. Results support E1 ($p = 0.10$) and E3 ($p = 0.085$).

Critique. We consider this study suggests a number of interesting lessons learned. We agree with the authors' emphasis that, "unless there is a clear reason to prefer simpler solution, it is probably wise to choose the flexibility provided by the design pattern solution, because unexpected new requirements often occur". However, we consider there is a question of efficiency for subjects lacking knowledge of patterns (in the pretest PAT groups).

Prechelt et al's (2001) *motivation* is the claim that using design patterns programmer productivity and software quality are improved. The *object* is *design patterns comment lines* (PCL) for the *purpose* of *investigation with respect to maintainability* for the *point of view* of the *researcher*. The *context* concerns two experiments run using students from two university lab courses, the one referred to as UKA (University of Karlsruhe), and the second as WUSTL (Washingthon University St. Louis).

Hypotheses. H_0: By adding PCL, there is no difference in performance time and errors committed in pattern-relevant maintenance tasks. H_1: By adding PCL, pattern-relevant maintenance tasks are completed faster. H_2: By adding PCL, fewer errors are committed in pattern-relevant maintenance tasks.

Variables. The *independent* variable in both experiments was the presence or absence of *design pattern comment lines* (PCL). *Dependent* variables were *performance time*, and *number of errors* quantified by assigning points and counting correct solutions.

Participants. In both experiments computer science students were used. The 74 subjects of the UKA experiment were 64 graduate and 10 undergraduate students, while in the WUSTL experiment the 22 subjects were undergraduate students. On average, their previous programming experience was as following: the UKA subjects had 7.5 years experience, 69% of them having some experience in OO programming and 58% of them in GUIs. The WUSTL subjects had 5 years experience, 76% of them having some experience in OO programming and 50% of them in GUIs.

Experiment design. In both experiments subjects performed on two different programs. Both programs were written in Java and C++. Program 'And/Or-tree' was a library for handling And/Or-trees of strings and a simple application of it. The Java versions have 362 LOC (133 of these containing only comments) in seven classes. An additional 18 lines of PCL were added for the PCL version. Its C++ versions have 498 LOC (included 178 lines of comments) in six classes. An additional 22 lines of PCL were added for the PCL version. They use the COMPOSITE and the VISITOR design patterns (Gamma et al., 1995). Program 'Phonebook', is a GUI (Graphical User Interface) program processing tuples (name, first name, phone number). Its Java versions have 565 LOC (197 of these containing only comments) in 11 classes. An additional 14 lines of PCL were added for the PCL version. Its C++ versions have 448 LOC (145 of them containing comments) in 6 classes. An additional 10 lines of PCL were added for the PCL version. They use OBSERVER and TEMPLATE METHOD design patterns (Gamma et al., 1995). A *counterbalanced* experiment design was used. For 'And/Or-tree' each subject received four subtasks, while for 'Phonebook' five subtasks were required. For each task (but not for each subtask) performance time was measured for each subject. For each subtask, the subjects' answers were graded. The number of completely correct solutions was also recorded.

Results and interpretation. Significance level was set at $\alpha = 0.10$. The results for 'And/Or-tree' task indicate that the UKA subjects using PCL performed significantly better in correct solutions ($p = 0.077$) and time ($p = 0.094$). Also, the WUSTL subjects using PCL performed significantly better in time ($p = 0.046$). The results for 'Phonebook' task indicate that the UKA subjects performed significantly better only in time ($p = 0.055$). The WUSTL subjects for the same task had to be discarded, because they were not sufficiently qualified for this task. Therefore, H_1 was supported in both tasks, while H_2 was supported only in the UKA 'And/Or-tree' task.

Critique. We consider that the programs were rather small in size. They were also thoroughly commented, even without PCL. A learning effect is possible due to counterbalancing. Finally, the graded point scales are rather subjective in nature.

3.5. Inspection Techniques

Laitenberger et al.'s (2000) basic *motivation* is the claim that inspections can lead to the detection and correction of anywhere between 50 and 90% of the defects in a software document. The *objects* studied are the *Perspective-Based Reading* technique (PBR) and the *Checklist-Based Reading* (CBR) technique, for the *purpose* of *investigation with respect to* the subjects' *defect detection ability* from the *point of view* of the *designer*, *tester* and *implementor*. The *context* concerns an experiment run using professionals as subjects performing on *UML* design diagrams (Pooley and Stevens, 1999) such as *use-case*, *sequence*, class and *collaboration* diagrams.

Hypotheses. H_{01}: An inspection team is as effective, or more effective, using CBR than using PBR. H_{02}: An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for the defect detection phase of the inspection. H_{03}: An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for the meeting phase of the inspection. H_{04}: An inspection team using PBR finds defects at the same or higher cost per defect than a team using CBR for all phases of the inspection.

Variables. The *independent variables* are *Reading technique* and *Order of reading* (CBR $\rightarrow$ PBR versus PBR $\rightarrow$ CBR). The four *dependent variables* are *team defect detection effectiveness* and *cost per defect*, with three different definitions, *the defect detection phase*, the *meeting phase*, and the *overall inspection*.

Participants. Eighteen practitioners working in industry with various backgrounds were involved. They had already written design documents (median 3 on a 5 point scale). Moreover, they had some experience in developing design documents and between 1 and 16 years of programming experience with a median of 2 years. None of them had yet participated in an inspection.

Experiment design. Two different systems were used. A 'web-based system' that allows users to perform a quiz using the web consisting of 21 pages of design documentation, including six collaboration diagrams and four class diagrams. The second system is a 'point of sales' system with a design documentation of 18 pages including six collaboration diagrams and three class diagrams. 21 and 19 defects were inserted in each system respectively. Individual variability is controlled by assignment of subjects to teams. The *counterbalanced repeated measure design* was used. The subjects were randomly assigned to two groups. To provide more insight into an inspection meeting, they were randomly assigned to three-person inspection teams each person reading a design document from one of three perspectives (designer, tester or implementor). After the reading exercise a *debriefing questionnaire* was completed by each subject.

Results and interpretation. For the analysis of the collected data *match-paired t*-test were used. Significance level was set at $\alpha = 0.05$. The team results are considered the unit of analysis. Results indicate the following: Considering *defect detection effectiveness*, the PBR technique was significantly more effective than the CBR one (mean: CBR: 0.43; PBR: 0.58; $p = 0.025$), thus rejecting hypothesis H_{01}. For the *defect detection phase* the cost per defect was lower for the PBR technique than the

CBR (mean: CBR: 85 min/defect; PBR: 43 min/defect; $p = 0.001$), also rejecting hypothesis H_{02}. Considering the *meeting effort* the cost per defect was lower for the PBR technique than the CBR (mean: CBR:47 min/defect; PBR: 13 min/defect; $p = 0.002$), rejecting hypothesis H_{03}. Finally, considering the effort for the *overall inspection* the team with the PRB technique had a lower cost per defect ratio than the team using CBR technique (mean: CBR: 132 min/defect; PBR: 56 min/defect; $p = 0.001$), thus rejecting hypothesis H_{04}.

Critique. We consider there is a risk for the subjects to become confused performing with two different reading techniques, especially since they lack prior inspection experience. However, we want to stress the significance of such techniques that contribute to defect detection at an early stage of software lifecycle.

4. Discussion

Although no longer new, the OO paradigm is still of growing importance. For this reason it is unsurprising that the empirical research community has started to turn its attention to evaluating the technology and endeavouring to learn more about how it may be most effectively deployed. From this analysis, better considered as corpus-based, since the topic is continually evolving, emerge three sets of outcomes. First, there are some comments regarding the actual conduct of these experiments, as well as our observations regarding the conduct of experiments in general. Second, there are the types of findings and patterns that result from these experiments. Third, there are some suggestions for new directions and further work for the experimental community.

First, we discuss the conduct of experiments. From this review a number of issues emerge:

1. Students subjects are exclusively used, in all but four cases. It is likely that professionals are either reluctant or unavailable to participate in such experiments. However, the OO paradigm offers several mechanisms and tradeoffs where decisions on best alternatives are usually fuzzy and mostly based on expert judgement. In other words, cumulative knowledge is likely to play a very important role in the design phase. Thus, it is questionable whether novice designers performing, usually with limited time, are always the most appropriate subjects. However, Briand et al. (1997) argue that student based experiments can provide useful results for several reasons. First, they can provide confirmatory evidence for results from case or other studies (e.g. Daly et al. (1996)). Second, they can identify interesting hypotheses that are worthy of further investigation in more realistic settings.

2. The artefacts used are, in most cases, small and thus inadequate to capture the functionality and intricate concepts of real problem domains where decisions and tradeoffs are complex.

3. Design model documentation that could give better understanding of the problem domains was not made available except in a few experiments. This does not seem representative of real OO development or maintenance.

4. Some experimental materials are flawed. There tends to be a naive view of inheritance and architecture in general. In some cases it is unclear whether the use of inheritance is the most suitable design strategy for the particular problem at hand. Other materials are derived from textbooks or are generated in an unrepresentative fashion, for instance by flattening inheritance hierarchies.

In this review we have expressed some reservations about details of some of the experiments. Some are easier to address than others. A perennial concern is the use of student subjects. However, from a pragmatic point of view this is a hard problem to overcome. Access to professional staff is frequently limited or restricted on grounds of cost. Experiments using students obviously have some value. Moreover, students are not a homogeneous group, some have considerably more experience than others and it may be that this can be further exploited. Nevertheless the recent study by Kirsopp et al. (1999) does indicate significant differences in behaviour so some caution must be exercised. A more avoidable problem has been the tendency not to provide design documentation. This seems unnecessary. Another problem area has been the nature of some of the artefacts and tasks many of which seem trivial—again, perhaps unavoidable—and contrived less so. Examples include the maintenance tasks for several experiments that require highly artificial specialisations of extant class inheritance hierarchies. The problem here is that this may reveal little about maintenance of OO systems in practice. Therefore, concerning the "conduct of experiments" we believe emphasis should be placed on two points. First, experimental settings should be as realistic as possible applying to those in practice. Secondly, planning, operation, and analysis of experiments should be according to the principles and good practice proposed in the literature such as (Wohlin et al., 2000).

Second, we turn to the findings of these experiments. We have identified 18 published experiments that, individually and collectively, contribute to our understanding. They addressed five distinct types of hypothesis. The first group dealt with comparisons between procedural or traditional technology and OOT. Out of these experiments only two (Henry and Humphrey (1990) and Lewis et al. (1991)) found any benefit of OOT over and above procedural technology. Other studies were unable to discriminate between the competing technologies or found superior performance for the procedural technology (Agarwal et al. (1999), Moynihan (1996), and Wiedenbeck et al. (1999)). Briand et al. (1997) made the interesting observation that they believed OOT to be more sensitive to design quality than procedural technology. Many researchers have commented on the long learning curve for OOT. It may well be that given the predominance of inexperienced student subjects this pattern of results is unsurprising.

Some additional evidence for this lack of positive support for OOT can be found in an important case study performed by Hatton (1998). He studied, over a 4 year period, the change, configuration, and quality control history of two systems developed in OO C++ and procedural C. It was found that for C++ developers: (a) it was much harder to trace faults; (b) development time was 25% higher and corrections took significantly longer; (c) largely, due to the distorted and frequently non-local relationships between cause and effect, the manifestation of a failure could be a "long way away" from the fault that led to; (d) for all defects the average correction time being 260% longer. However, Hatton expressed reservations about how much the reported problems related specifically to OO and how much they relate to C++. We note, however, a recent survey pointing to the dominance of the Java language in the near future (Computing, 1999).

A fundamental issue, examined by a single experiment (Lewis et al., 1991), is that of reuse. The concept of reuse is nearly as old as software. Despite the claims that OOT provides more reuse results than other paradigms, there is an absence of supporting evidence in the literature. Rare exceptions are the Lewis et al. experiment we have examined above, and Stark (1993). Stark addressed the impact of OOT at the SEL[6] including reuse. He found some empirical support for OOT promoting reuse. Specifically, his findings from applying OOT on 11 projects in SEL indicated an increase in code reuse from the baseline 20–30% to 75–80%. Costs were reduced by a factor of 3, change and error rates were reduced by a factor of 10, and project cycle time was cut roughly in half. On the negative side, there were difficulties adapting the code and runtime performance. Despite the experimental and case study results, some authors provide warnings regarding reuse. We believe more empirical evidence could usefully contribute to this debate.

The second group of experiments, represented by a single study (Briand et al., 1997), dealt with OO design principles. Its findings, consistent with a previous experiment by the same authors (Briand et al., 2001), highlight the sensitivity of OO systems to violations of good design practice. This points to a need for further research and investment in the definition of design standards, heuristics and principles.

The third group of experimental hypotheses examined specific OO mechanisms and inheritance in particular. Here, whilst the results were not fully consistent, the pattern is that deeper inheritance hierarchies tend to be associated with negative factors such as hard to understand and harder to change. Such findings are in line with more recent findings from the OO community who no longer see class inheritance as a panacea and who now emphasise other mechanisms such as composition and aggregation except in a few specialised situations. We believe there is a danger that this move by OOT specialists is not being mirrored by empirical researchers.

The fourth group dealt with design patterns, represented by two experiments. These seem to offer some credence to the notion that patterns are of value. However, we believe it is difficult to draw too definite conclusions from just two experiments that were based on relatively small artefacts. We further believe that the benefits of design patterns could be more apparent in the long run, since the well-designed solutions they provide might repeatedly be exploited.

The fifth group examined inspection techniques. Although only a single experiment was examined, its findings seem of certain significance since they provide some evidence that such techniques could considerably contribute to software quality. Of interest to us could be e.g. whether they provide some means in efficiently examining design aspects, e.g. in assessing heuristics compliance.

Thirdly and finally, we consider areas for further research. From the foregoing analysis, we have observed that not all OO design issues appear to have been considered by empirical researchers. This review has also highlighted a preoccupation with a single OO mechanism, namely inheritance, which is considered of diminishing importance by the OO community. Clearly it is necessary that empirical research keep pace with technological developments. This finding is also supported by the fact that the other equally significant mechanism—composition—and its contribution to design, evolution and reuse, has not been investigated at all. Its importance can be extracted from some authors' positions. Gamma et al. (1995) state, "our experience is that designers overuse inheritance as a reuse technique, and designs are more reusable (and simpler) by depending more on object composition. You'll see object composition applied again and again in the design patterns". Additionally, some other authors suggest that inheritance might be used in limited contexts in favour of composition, in order to achieve more extensible and reusable designs (Kilian, 1991; Coad and Mayfield, 1997; Ennals, 1998; Venners, 1998), which we consider suggests a challenge for empirical research. Furthermore, other factors like existing heuristics, guidelines, principles, and design patterns proposed by various authors are likely to be highly significant (Rumbaugh, 1993; Firesmith, 1995; Reil, 1996; Coad and Mayfield, 1997). Clearly we need more empirical work to evaluate and refine the burgeoning number of heuristics. The work by Kirsopp et al. (1999) is a small step in this direction.

Acknowledgments

The authors wish to thank Tony Moynihan and Lutz Prechelt for their useful comments on an earlier draft. They also acknowledge the detailed and helpful comments of the anonymous referees.

Notes

1. Our search for published experiments included use of the ISI Scientific Citation Index, IEEE Digital Library, ACM Digital Library, and Computer Science Bibliography Collection Advanced Search, on 30 May 2001, using search terms 'object' and 'experiment'. This was augmented by technical reports and other sources that we had been made aware of at the time.
2. G. Booch (1994) speaks of the class and object structure as its *architecture* (p. 15).
3. This experiment compares procedural and OO mindsets (prior experience and performance on a specific technology) rather than procedural and OO artefacts, nevertheless, we choose to place it in the category of experiments comparing OO with procedural technology as it most closely fits in that category.
4. By 'structure' the authors refer to OO aspects.
5. By 'structure' or 'structural' the authors refer to OO aspects.
6. Software Engineering Laboratory.

References

Abreu, F., and Melo, W. 1996. Evaluating the Impact of Object-Oriented Design on Software Quality. Proceedings of the 3rd ISMS (Metrics'96).

Agarwal, R., and Sinha, A. 1996. The role of prior experience and task characteristics in object-oriented modeling: An empirical study. *International Journal of Human–Computer Studies* 45(6): 639–667.

Agarwal, R., De, P., and Sinha, A. 1999. Comprehending object and process models: An empirical study. *IEEE Transactions of Software Engineering* 25(4): 541–555.

Agarwal, A., Sinha, P., and Tanniru, M. 1996. Cognitive fit in requirements modelling: A study of object process methodologies. *Journal of Management Information Systems*, 13(2): 137–162.

Aksit, M., and Bergmans, L. (1992). Obstacles in Object-Oriented Software Development, in *OOPSLA' 92*.

Anderberg, M.R. 1973. Cluster Analysis for Applications. New York: Academic Press.

Amstrong, J., and Mitchell, R. 1994. Uses and abuses of inheritance. *Software Engineering Journal* January: 19–26.

Basili, V., and Burgess, A. 1995. Finding and experimental basis for software engineering, *IEEE Software* 92–93.

Basili, V., Briand, L., and Melo, W. 1996. A Validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(October): 751–761.

Blaha, M. 1993. Aggregation of parts of parts of parts. *JOOP* September: 14–20.

Briand, L., Bunse, C., and Daly, J. 2001. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering* 27(6): 513–530.

Briand, L. et al. 1997. An experimental comparison of the maintainability of object-oriented and structural design documents. *Empirical Software Engineering* 2(3): 291–312.

Capretz, L., and Lee, P. 1993. Object-oriented design: guidelines and techniques. *Information and Software Technology* 35(April): 195–206.

Cartwright, M. 1998. An empirical view of inheritance. *Information and Software Technology* 40(14): 795–799.

Chatel, S., and Detienne, F. 1994. Expertise in Object-Oriented Programming, in ECCE 7, S. Augustin, Germany.

Civelo, F. 1993. Roles for Composite Objects in Object-Oriented Analysis and Design, in *OOPSLA'93*.

Coad, P. 1992. Object-oriented patterns. Com. of the ACM 35(9): 152–159.

Coad, P., and Mayfield, M. 1997. *Java-Inspired Design: Use Composition, Rather than Inheritance*, in American Programmer, 22–31.

Coad, P., and Yourdon, E. 1991. Object-Oriented Design. 1st Ed. Englewood Cliffs, NJ: Prentice-Hall.

Coad, P., and Yourdon, E. 1991. Object-Oriented Analysis. 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall.

Computing, Java on Course to dominate by 2002, in VNU business publications. 1999. p. 3.

Coplien, J., H. D., and W. D. 1998. Commonality and variability in software engineering. *IEEE Software* December: 37–45.

Corritore, C., and Wiedenbeck, S. 2000. Direction an Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study, in 8th International Workshop on Program Comprehension (IWPC'00).

Cunis, R. 1997. Improving Software Development the Object-Oriented Way, in SQE 97. Udine, Italy.

Daly, J. et al. 1996. Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software. *Empirical Software Engineering* 1(2): 109–132.

Daly, J. et al. 1996. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In: *Proceedings of the International Conference on Software Maintenance*. Washington: IEEE Computer Society Press.

DeMacro, T. 1978. Structured Analysis and System Specification. Englewood Cliffs, NJ: Prentice-Hall.

DeRemer, F., and Kron, H. 1976. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2)(June): 80–86.

Dvorak, J. 1994. Conceptual Entropy and Its Effect on Class Hierarchies. *IEEE Computer* 27(6)(June): 59–63.

Ennals, R. 1998. Inheritance considered harmful. EXE, May: 22–30.

Everrit, B. S., and Dunn, G. 1983. Advanced Methods of Data Exploration and Modeling. London Heinemann.

Fenton, N., and Pfleeger, S. L. 1997. Software Metrics, A rigorous & Practical Approach, 2nd Ed. London: International Thompson Computer Press.

Firesmith, D. 1995. Inheritance guidelines. *JOOP* May: 67–72.

Gamma, E. et al. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. New York: Addison-Wesley.

Harrison, R., Counsell, S., and Nithi, R. 2000. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems Software* 52(2–3): 173–179.

Hatton, L. 1998. Does OO Sync with How We Think? *IEEE Software* May/June: 46–54.

Henry, S., and Humphrey, M. 1990. A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software. *IEEE Conference on Software Maintenance*, IEEE Computer Society Press: 258–265.

Johnson, R., and Foote, B. 1988. Designing reusable classes. *JOOP* June/July: 22–30, 35.

Kilian, M. 1991. A Note of Type Composition Reusability. ACM SIGPLAN OOPS Messenger, 2(3): 24–32.

Kirsopp, C., Shepperd, M., and Webster, S. 1999. A empirical study into the use of measurement to support OO design evaluation. in IEEE 6th International. Metrics Symposium, November 5–6, 1999. Boca Raton, Fl.

Laitenberger, O. et al. 2000. An experimental comparison of reading techniques for defect detection in UML design documents. *Journal of Systems Software* 53: 183–204.

Lee, S., and O'Keefe, R. 1996. The effect of Knowledge Representation Schemes on Maintainability of Knowledge-Based Systems. *IEEE Transactions on Knowledge and Data Engineering* 8(1): 173–178.

Lewis, J. et al. 1991. An Empirical Study of the Object-Oriented Paradigm and Software Reuse, *OOPSLA '91* 184–196.

Lieberherr, K., and Riel, A. 1989. Contributions to Teaching Object-Oriented Design and Programming, in *OOPSLA*.

Lieberherr, K., Bergstein, P., and Silva-Lepe, I. 1991. From objects to classes: Algorithms for optimal object-oriented design. *Software Engineering Journal* July: 205–228.

Martin, J. 1992. Strategic Data Modelling. New Jersey: Prentice-Hall.

Mattsson, M. 1999. Effort distribution in a six year industrial application framework project. In: *IEEE International Conference On Software Maintenance* (*ICSM99*). Oxford, England.

Morisio, M. et al. 1999. Measuring functionality and productivity in web-based applications: A case study. In: Sixth *IEEE International Symposium on Software Metrics*.

Moynihan, T. 1996. An Experimental Comparison of Object-Orientation and Functional-Decomposition as Paradigms for Communicating System Functionality to Users. *Journal of Systems Software* 33(2): 163–169.

Odell, J. 1994. Six different kinds of composition. *JOOP* January: 10–15.

Pant, Y., Henderson-Sellers, B., and Verner, J. 1996. Generalization of object oriented components for reuse: Measurements of effort and size change. *JOOP* 9(2): 19–31.

Pennington, N. 1987a. Comprehension strategies in programming. In: S.S.G.M., E. Olson, (eds): Empirical Studies of Programmers: Second Workshop, Ablex, Norwood, NJ, Soloway. 100–113.

Pennington, N. 1987b. Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive Psychology* 19: 295–341.

Pohl, I. 1993. Object-Oriented Programming Using C++. Redwood City, CA: Benjamin/Cummings.

Prechelt, L. et al. 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27(12): 1134–1144.

Prechelt, L. et al. 2001. Two Controlled Experiments Assessing the Usefulness of Design Pattern Information in Program Maintenance. *IEEE Transactions on Software Engineering* (accepted for publication).

Ramakrishnan, S., and Menzies, T. 1996. An ongoing OO software engineering measurement experiment, in *International Conference Software Engineering Education and Practice*.

Riel, A. 1996. Object-Oriented Design Heuristics, New York, Addison-Wesley.

Rumbaugh, J. 1993. Disinherited! Examples of misuse of inheritance. *JOOP* Jan: 19–24.

Rumbaugh, J. et al. 1991. Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice-Hall.

Seidewitz, E. 1996. Controlling inheritance. *JOOP* 8(January): 36–42.

Shoval, P., and Frumermann, I. 1994. OO and EER Conceptual Schemas: A Comparison of User Comprehension. *Journal of Database Management* 5(4): 28–38.

Stark, M. 1993. Impacts of Object-Oriented Technologies: Seven Years of Software Engineering. *Journal Systems Software* 23: 163–169.

Stevens, P., and Pooley, R. 1999. Using UML: Software Engineering with Objects and Components, O.T. Series. Harlow, Essex, UK: Addison-Wesley.

Tichy, W. 1998. Should computer scientists experiment more? *IEEE Computer* May: 32–40.

Unger, B., and Prechelt, L. 1998. The impact of inheritance depth on maintenance tasks—Detailed description and evaluation of two experimental replications. Technical Report, Karlsruhe University: Karlsruhe, Germany.

Venners, B. 1998. Inheritance versus composition: Which one should you choose? in *Java World*.

Wiedenbeck, S., and Ramalingam, V. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human–Computer Studies* 51: 71–87.

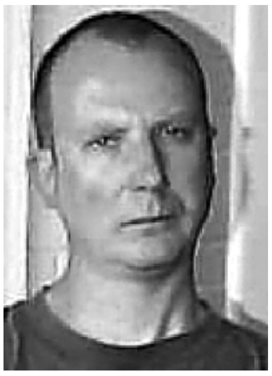
Wiedenbeck, S. et al. 1999. A comparison of the comprehension of object-oriented and procedural programms by novice programmers. *Interacting with Computers*, 11: 255–282.

Wohlin, C., and Runeson, P. et al. 2000. Experimentation in Software Engineering—An introduction. Norwell, MA. Kluwer Academic Publishers.

Yida, M., Sahraoui, H., and Lounis, H. 1998. Impact of complexity on reusability in OO systems, in *ECOOP'98*. Berlin, Germany.

Zelkowitz, M., and Wallace, D. 1998. Experimental Models for Validating Technology. *IEEE Computer* May: 23–31.

Ignatios Deligiannis is a Lecturer at Technological Education Institute of Thessaloniki, Greece, and PhD candidate at the University of Macedonia, Greece. He is also member of ESERG (Empirical Software Engineering Research Group at Bournemouth University, UK). His main interests are Object-Oriented software assessment, and in particular design heuristics and measurement. He received his BSc in computer science from the University of Lund, Sweden, in 1979, and then worked for several years in software development at Siemens Telecommunications industry.



Martin Shepperd received a PhD in computer science from the Open University, UK in 1991. He is professor of software engineering at Bournemouth University, UK. He has published more than 70 refereed papers and three books in the area of empirical software engineering. He is editor of the Journal Information and Software Technology and Associate Editor of IEEE Transactions on Software Engineering.

Steve Webster is currently a self-employed consultant working in the area of object technology and design processes. His current interest is in helping to make object and component processes work in large government organisations. He is a committee member for the BCS OOPS OT series of conferences held annually in the UK.

Manos Roumeliotis received the Diploma in electrical engineering from the Aristotle University of Thessaloniki, Greece in 1981, and the MS and PhD degrees in computer engineering from Virginia Polytechnic Institute and State University, Virginia, USA, in 1983 and 1986, respectively. At VPI & SU he taught as a visiting Assistant Professor in 1986. From 1986 through 1989 he was an Assistant Professor in the Department of Electrical and Computer Engineering at West Virginia University. Currently he is an Assistant Professor in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. His research interests include digital logic simulation and testing, computer architecture and parallel processing, and computer network optimization. He is a member of the IEEE Computer Society's Technical Committee on Computer Architecture.