# Investigating Maintenance Processes in a Framework-Based Environment


Victor R. Basili† Filippo Lanubile‡ Forrest Shull†

†Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD, USA
{basili, fshull}@cs.umd.edu

‡Dipartimento di Informatica
University of Bari
Bari, Italy
lanubile@di.uniba.it



## Abstract[1]

The empirical study described in this paper focuses on the effectiveness of maintenance processes in an environment in which a repository of potential sources of reuse exists, e.g. a context in which applications are built using an object-oriented framework. Such a repository might contain current and previous releases of the system under maintenance, as well as other applications that are built on a similar structure or contain similar functionality. This paper presents an observational study of 15 student projects in a framework-based environment. We used a mix of qualitative and quantitative methods to identify and evaluate the effectiveness of the maintenance processes.


## 1. Introduction

An object-oriented framework is a class hierarchy augmented with a built-in model that defines how the objects derived from the hierarchy interact with one another to implement some functionality [16, 22]. A framework is tailored to solve a particular problem by customizing its abstract and concrete classes, allowing the framework architecture to be reused by all specific solutions within a problem domain. By providing both design and infrastructure for developing applications, the framework approach promises to develop applications faster [13]. Another benefit is that the use of frameworks results, over time, in a set of applications all based on the same underlying structure. The existence of this common infrastructure poses new questions for maintenance: Are different models of maintenance necessary in such an environment? Can a set of applications, all based on the same underlying framework, be exploited for reuse?

These questions become especially interesting and relevant if we think about the state of the practice in the software maintenance field. Consider the three maintenance process models in [2]: the quick-fix model, the iterative enhancement model, and the full-reuse model. With the quick-fix model, maintainers start by modifying the source code, then test the new version, and finally modify the existing documentation. Because of time pressure, this model is the most popular maintenance process model. With the iterative enhancement model, maintainers start with an analysis of the existing system's documents, and modify any relevant software artifact working downward from the requirements through the lower abstraction levels, including source code. This process model is applied in those more mature environments where maintenance activities are planned ahead and more effort is spent on preserving the integrity of the original design. The third model, the full-reuse model, differs from the iterative-enhancement model because it assumes that there exists a repository of software artifacts that can be reused. This repository might consist of the current and earlier versions of the application under maintenance, and/or a set of “similar” applications. This model is more “theory than practice” due to the high investment costs of repositories in which extensive reuse is feasible.

However, object-oriented frameworks and framework-based applications can be viewed as a repository of both design and code, a source of reusable components such as design patterns [7] and object-oriented classes. Domain-oriented frameworks, such as those developed in the financial and manufacturing domain, also implement a basic set of domain-specific requirements that can be considered reusable. In contrast to the quick-fix and


[1] This work has been supported by NSF grant CCR9706151 and UMIACS.

iterative enhancement models, which assume that there is only one source of reusable code (the existing system release), a full-reuse model, enabled by the usage of a framework, supports multiple sources of reuse. (For example, a set of applications that cover the functionality that needs to be added to the system under maintenance.) This model raises other questions: Is it feasible to expect the maintainer to find the right reuse sources for a given piece of functionality? Can we leverage this multiplicity for making changes faster and better?

This study examines some of the practical implications of this process of developing new applications by adapting functionality from a set of applications previously developed. A survey of the literature on frameworks shows that relatively little has been written on using frameworks (as opposed to building or designing them [6]). Most of the current work on using frameworks tends to ignore the possibilities provided by the previously developed applications and concentrates instead on strategies for documenting the framework design [4, 8, 11, 14, 23].

However, the effort required to learn enough about the framework to begin coding is very high, especially for novices [16, 22]. As in conventional maintenance, obtaining this understanding is a non-trivial task for a developer unfamiliar with the system. A new approach, which developers could use to minimize their learning curve, is necessary. Considering how quickly the use of commercial frameworks, such as Microsoft Foundation Classes and Java Development Kit, is increasing, framework-based applications will become the legacy systems of the near future.

The case study described in this paper is part of a broader empirical study addressing software reading for construction [3]: how application developers understand system artifacts in order to develop a new system or release. Reading for construction is important for comprehending what a system does, what capabilities exist and do not exist; it helps us abstract the important information in the system. It is useful for maintenance as well as for building new systems from reusable components and architectures.

## 2. Description of the Study

We ran an exploratory study into framework usage as part of a software engineering course at the University of Maryland. This study took place in an environment much like the one described in the previous section: subjects were asked to develop an application by adapting a framework. Some of the functionality required for the new application could be reused from a set of existing applications that had been previously developed using the same framework.

Our class of upper-level undergraduates and graduate students was divided into 15 two- and three-person teams. Teams were chosen randomly and then examined to make certain that each team met certain minimum requirements (e.g., no more than one person on the team with low C++ experience). Each team was asked to develop an application during the course of the semester, going through all stages of the software lifecycle (interpreting customer requirements into object and dynamic models, then implementing the system based on these models). The application to be developed was one that would allow a user to edit OMT-notation diagrams [18]. That is, the user had to be able to:

- graphically represent the classes of a system and the different types of relations between them,
- perform some operations, e.g., moving or resizing, directly on these representations, and
- enter descriptive attributes (class operations, object names, multiplicity of relations, etc.) that would be displayed according to the notational standards.

The project was to be built on top of the ET++ framework [24], which assists the development of GUI-based applications. ET++ provides a hierarchy of over 200 classes that provide windowing functionality such as event handling, menu bars, dialog boxes, and the like. ET++ came with a set of 32 example applications that had been implemented using the framework. Although some of these were simple programs intended to demonstrate some aspect of the framework functionality, the majority were applications such as graphical file browsers, spreadsheets, and window-based text editors.

Since we approached this study from the viewpoint of software reading, our primary focus was on the processes developers would engage in, as they attempted to discover enough information about the framework and the set of existing applications to be able to use them to effectively construct a new application. We reasoned that the best approach would be to observe in practice a number of different techniques for reusing functionality from a set of applications, and the effects of these techniques in practice.

All of the subjects received training on using the ET++ framework and were encouraged to reuse functionality from the existing applications. Half of our subjects (8 of the 15 teams) were trained in a technique for reusing functionality from the application set, although they were not required to use it if they did not find it helpful for reuse. This training took place in classroom lectures, in which a small example was used to demonstrate the steps of the technique, which can be summarized as:

| Aspect of Interest | Measures | Form of Data | Unit of Analysis | Collection Methods |
|---|---|---|---|---|
| Adaptation Processes | techniques used | Qualitative | team | interviews, final reports |
| | tools used | Qualitative | team | Interviews |
| | team organization | Qualitative | team | interviews, self-assessments |
| | starting point for implementation | Qualitative | team | interviews, final reports |
| | difficulties encountered with technique | Qualitative | team | problem reports, self assessments, final reports |
| Products | degree of implementation for each functionality | Quantitative | team | implementation score, final reports |
| Other Factors | effort | Quantitative | team | progress reports, questionnaires |
| | level of understanding of technique taught | Quantitative | individual | exam grades |
| | previous experience | Quantitative | individual | questionnaires |

**Table 1. Types of measurements and means for collecting**

- looking through the example applications in order to find a potential match for the functionality sought;
- using the tools available to discover which classes in the example were responsible for implementing the functionality;
- adapting either whole classes or particular methods and attributes from the example application for use in the new system.

(The full techniques can be found in [20].)We call this a "strictly adaptive" technique because it was focused entirely on guiding the developer to find useful functionality in the existing applications, which could then be tailored to the current system.

The remaining 7 teams did not receive explicit training in the technique (for reasons pertaining to the larger study) but could adapt functionality from the applications in an ad hoc manner.

During implementation, our subjects did undertake significant amounts of reuse from the existing applications. We asked the students to provide records of the activities they undertook so that we could understand what methods they used and how effective these were. In this paper, we report our observations about their experiences from the point of view of a set of questions focused on a full-reuse model of maintenance:

- Can a framework-based environment effectively support a full-reuse model of maintenance?
- What are attributes of effective maintenance techniques in a full-reuse environment?
- Should maintenance in this environment be driven by the functionality provided by the framework and application set, or by the model of the application to be developed?
- Can we characterize the way in which functionality is reused from the existing applications? Are there attributes of the application set that make this process more or less difficult?

## 3. Analysis

Since the analysis was carried out both for individuals and the teams of which they were part, we treat the study as an embedded case study [25]. In order to obtain an accurate picture of the work practices involved, we collected a wide variety of data over the course of the semester, using a number of different methods (see Table 1).

We analyzed this mix of qualitative and quantitative data to gain some insight into what was going on within each team. By comparing and contrasting teams, we searched for implications that addressed the study questions given in section 2. Since there has not yet been a large amount of work spent on understanding this area of framework use, our focus was on using this information to look for tentative but reasonable hypotheses and not on testing known hypotheses. The process of building theories from empirical research has been first proposed in the social science literature [9, 5] but it is also followed in the software engineering discipline [19].

### 3.1. Analysis of Maintenance Processes

Our first step was to get an overview of what adaptation processes teams had used. (By "adaptation processes" we mean how the team had been organized, what techniques they had used to understand the framework and implement the functionality, whether they based their implementation on an existing application or started from scratch, and what tools they had used to support their techniques.) To this end, we performed a qualitative analysis of the explanations given by members of the teams during the interviews and final reports, and on the self-assessments.

We first focused on understanding what went on within each of the teams during the implementation of the project. We identified important concepts by classifying the subjects' comments under progressively more abstract themes, then looked for themes that might be related to one another. Once we felt we had a good understanding of what teams did, we made comparisons across groups to begin to hypothesize what the relevant variables were in general. This allowed us to look for variations in team effectiveness that might be the result of differences in those key variables, as well as to rule out confounding factors.

We found that teams used adaptation processes for implementing the project that fell into 1 of 2 categories.

1. Starting from an existing application. 9 teams exploited the fact that they could treat the implementation as if it was a maintenance project in a full-reuse environment. That is, they selected one of the existing applications as a starting point and began their implementation by modifying the existing functionality of their base application. Additional functionality was reused from the other applications by adapting it to the base application. Since this approach begins by modifying an existing application, these teams are the basis for our observations about maintenance processes.
2. Starting from "scratch"[2]. The remaining 6 teams treated the implementation as more of a traditional development project. That is, they began their implementation from as little prior functionality as they could by implementing directly on top of the framework. Technically, this can still be considered a full-reuse approach since functionality from the existing applications was later extensively reused in the new system, by learning from the working examples and picking up the relevant pieces.

[2] In the context of framework usage, starting from scratch still implies that the design and code of the framework are reused.

All teams who started from an existing application chose the same one, a simple entity-relation diagram editor (known as "ER"). It was similar to the OMT editor to be developed, but much simpler: as specified for the OMT editor, the ER diagram editor allowed simple shapes to be added to an editable document, and allowed these shapes to be selected, moved, and associated with one another. However, more sophisticated functionality was lacking. (It should be noted that the ER application was the closest example in the set to the new requirements. The only other real candidate was "Draw", a small drawing editor. Subjects reported that Draw was not used as a basis for implementation because it was more complicated and harder to understand than ER, and contained too much extraneous functionality that would have to be removed.)

We measured the effectiveness of the process used by subjects by means of an "implementation score" that reflected how well the functionality specified in the requirements was implemented in the final system. To calculate this score for each system, we first evaluated the implementation of each required piece of functionality on a 6-point scale[3]. We then weighted the score for each functionality by a factor that represented how important we thought that functionality was to the entire system. The implementation score could then be obtained by summing the scores for each functionality. The weights were chosen in such a way that if each functionality worked well, an implementation score of 100 would be obtained. Scores less than 100 provided a rough indication of what percentage of system functionality had been implemented. (Because "extra credit" was awarded in rare instances that functionality beyond what was required was implemented, it was also possible for implementation scores to be slightly greater than 100.)

We used a t-test to determine whether teams starting from the ER application tended to achieve significantly higher implementation scores than teams starting from scratch. One point, representing a team that experienced severe organizational difficulties that were primarily responsible for a very low implementation score (equal to 44), was removed from this analysis as an extreme outlier (according to the definition given in [15]). Due to the

[3] This scale is based upon the suggested scale for reporting run-time defects in the NASA Software Engineering Laboratory [21]: "required functionality missing", "program stops when functionality invoked", "functionality cannot be used", "functionality can only partly be used", "minor or cosmetic deviation", "functionality works well".

small sample size and the exploratory nature of this study, we used an $\alpha$-level of 0.20, which is higher than standard levels. Although not common, this $\alpha$-level has been used in similar hypothesis-building studies, e.g. [1]. We realize that statistical tests at this significance level do not provide strong evidence of a relationship, but instead see their contribution as helping detect patterns in the data that can be specifically tested in future studies.

The t-test yielded a p-value of 0.15 ($t = 1.538$), which is significant at the 0.20-level and provides some evidence that teams who started by modifying an application tended (mean implementation score = 89) to be more effective than those starting from scratch (mean score = 83).

Again following a qualitative method, we undertook an investigation into whether there were characteristic problems reported by the teams who adopted the full-reuse maintenance process. Student remarks from the problem reports, self assessments, and final reports were examined and provided the following indications:

- One-third (3/9) of the maintenance teams in our study had trouble adapting the example applications because they did not conform to a consistent organization or structure.
- Two-thirds (6/9) of the maintenance teams reported difficulties in finding the necessary functionality within the existing applications.
- Two-thirds (6/9) of the teams wasted time and effort during the course of the implementation phase by having to re-implement some functionality that they had implemented previously in a short-sighted way. Almost half (4/9) of the teams brought up the importance of their object models as guides to implementation. Half of these teams (2/4) reported that they had been able to stay fairly close to their original object model of the system during the course of implementation. Both of these teams ranked in the top half of the class with regards to implementation score. The remaining 2 teams were reporting problems; they had strayed from their original model during implementation. It seems that this inability to follow the model had some negative effects, as both were ranked in the bottom half of the class. (Since all but one of these teams had received the same grade on the original model, it seems unlikely that the variation in performance could have been caused by factors outside the implementation phase, such as the quality of the model itself.)

### 3.2. Analysis of Reuse Processes

In this section, we report our observations on reuse. Since all fifteen teams extensively reused functionality from the application set in implementing this system, our observations in this section come from all teams.

As in the previous section, we undertook a qualitative analysis to detect effective reuse techniques. The most relevant factor turned out to be the type of reuse process used by the team. There were two basic types of reuse process:

1. One-third of the teams used the strictly adaptive technique that we developed (discussed in section 2) and for which we provided training to some of the subjects.
2. Two-thirds of the teams used ad hoc reuse techniques. This category includes a number of different techniques which subjects developed on their own and found to be effective.

In order to understand how the type of technique used impacted the teams' effectiveness, we focus on certain *key functionalities*, that is, certain requirements for which there was a large degree of variation between teams in terms of their implementation score.

We then undertook a quantitative analysis of whether the teams' implementation of these key functionalities was effected by the type of technique used. We used a chi-square test of independence to test whether there was a correlation between the way teams tended to implement the functionality, and the type of reuse technique they used. Again, we use an $\alpha$-level of 0.20 due to the small size of our sample. We also present the product moment correlation coefficient, $r$, as a measure of the effect size [12]. (An $r$-value of 0 would show no correlation between the variables, whereas a value of 1 shows a perfect correlation.)

We identified 4 key functionalities: links, dialog boxes, deletion, and multiple views.

1. <u>Links</u>: The requirements for the OMT editor stated how the program should handle links between classes. The ER entity-relation editor provided simple functionality that linked objects with a line connecting their centers. Although useful as a starting point, this implementation was not sophisticated enough for the project, because the same two classes in an OMT diagram may be connected by multiple links, which should not overlap. There was, however, no application with functionality that explicitly addresses this concern. Almost all (4/5) of the teams who used the predefined technique implemented the less sophisticated version of the functionality found in the ER application. By comparison, less than half (4/10) of the teams who used an ad hoc technique turned in the less sophisticated implementation. The chi-square test was used to determine whether teams using the strictly adaptive technique had a different probability of turning in the more sophisticated

implementation than ad hoc teams. This test resulted in a p-value of 0.143 ($\chi^2$ = 2.143), which is statistically significant at the selected $\alpha$-level. An *r*-value of 0.38 confirms that this shows a moderate correlation [10] between level of sophistication and type of technique.

2. Dialog Boxes: There were existing applications which showed how to create dialog boxes containing graphical devices (e.g. text fields, radio buttons) and how to use them to display and store information. The difficulty was that this functionality was spread piecemeal over multiple applications and students had a hard time finding and integrating all of the functionality they needed. About half of the class (7/15) managed to get the dialog box functionality working correctly and interfaced with the rest of the system. Both the ad hoc and strictly adaptive techniques seemed equally likely to get this functionality working correctly. The chi-square test here yielded a p-value of 0.714 ($\chi^2$ = 0.134), for which the related *r*-value is 0.10. This confirms that response levels are effectively equal between the two categories.
3. Deletion with Undo/Redo: There was at least one existing application that clearly contained functionality to delete classes and links from a diagram. All teams were able to implement this functionality. Getting the functionality to support the ability to undo or redo a deletion was apparently more challenging, however, although the existing applications covered this as well. Partly, this may have been due to students simply forgetting to implement this part of the functionality since it was not explicitly mentioned in the requirements. Teams basically implemented deletion in one of three ways, of increasing sophistication. Again, both the ad hoc and strictly adaptive techniques seem equally distributed among these three categories. The chi-square test for this functionality yielded a p-value of 1 ($\chi^2$ = 0.000), with an associated *r*-value of 0, indicating that response rates are exactly equal regardless of the type of technique used.
4. Views: The requirements for the OMT editor stated that the program must provide multiple views of the currently opened document. Applications existed which satisfied the project's requirements about views. However, there were also existing applications that gave an even more sophisticated implementation that allowed views to be dynamically added and deleted. All but 3 teams achieved the more sophisticated implementation. 2 of these 3 teams turning in less functionality used the strictly adaptive technique. The chi-square test resulted in a p-value of 0.171 ($\chi^2$ =1.875), which is statistically significant at the selected $\alpha$-level. An *r*-value of 0.35 shows a moderate correlation between the variables.

# 4. Study Findings

The following hypotheses represent the study findings, along with the supporting evidence, based on the qualitative and quantitative analysis of the study results. They are organized by the research question to which they relate.

**Can a framework-based environment effectively support a full-reuse model of maintenance?**

This study shows that it was feasible to treat the framework and example applications as a full-reuse environment. All of our subjects did reuse functionality from multiple applications and exploited the underlying design and functionality of the framework, and completed the implementation of the system in the time allowed. (However, we cannot conclude that the full-reuse maintenance model was superior to other development models, since subjects in our study used only the full-reuse model.)

Moreover, subjects who took the most advantage of the full-reuse environment did better than those who started directly from the framework.

*Hypothesis 1: For implementing a set of requirements in a framework-based environment, if a suitable example application can be found, then adapting that application is a more effective strategy than starting from scratch.*

Obviously, more work needs to be done on how to recognize a "suitable" example. The current study indicates that the benefits of relying on an existing application as a starting point (which include being able to exploit an existing file structure and to model new classes on similar ones which already exist in the application) can outweigh the negatives (the extra work of identifying relevant functionality and removing irrelevant code), which leads us to our second question:

**What are attributes of effective maintenance techniques in a full-reuse environment?**

Having decided that this environment could be considered a full-reuse maintenance environment, we used our observations of the students to reason about useful attributes of full-reuse processes. In particular, we looked for practices that tended to be effective or difficult across some number of the student teams, in order to find attributes that applied to full-reuse maintenance in general and not just the approaches of particular groups.

Our beginning learners did report characteristic problems with their adaptation processes. For example, the majority of teams experienced difficulties finding desired functionality within the existing applications.

*Hypothesis 2: Techniques for adapting framework-based applications require guidance to help developers find even minor bits of functionality in the existing applications.*

Although the teams using this technique usually managed to get the functionality working in the end, the problems with finding it in the first place seemed especially acute when the functionality needed was a very small part of a much larger application (e.g., the dialog boxes from key functionality 2). This indicates that we did not provide enough guidance to developers to assist them in finding and extracting small pieces of functionality embedded within larger existing applications. Future studies need to be undertaken to determine if we can add this sort of guidance to make more useful techniques for developers who are not yet used to the framework or its derived applications.

This study has also shown the effectiveness of the adaptation technique relies on the set of applications:

*Hypothesis 3: The effectiveness of a technique for adapting framework-based applications depends on breadth of functionality and other characteristics of the existing applications.*

As others [17] have pointed out, learning how to implement functionality from existing applications is difficult because the rationales for design choices, which explain why the finished implementation looks the way it does, are usually not included in the documentation. When attempting to reuse functionality from existing applications, developers are implicitly asked to reconstruct the choices that led to the finished implementations they are studying. This situation can actually be made worse in a full-reuse environment, since effective reuse requires the developer to understand the rationales behind a number of applications, not just one.

**Should maintenance in this environment be driven by the functionality provided by the framework and application set, or by the model of the application to be developed?**

When using frameworks, there is the important question of whether to modify the object model of the system, so as to exploit a piece of functionality offered by the framework that might not exactly fit the original plan, or to keep the object model "as is", even if it makes implementing the application on top of the framework harder. From our subjects' comments about their object models, we hypothesize:

*Hypothesis 4: Techniques for adapting framework-based applications should follow the original object model of the system and resist the temptation to incorporate additional functionality provided by the framework, so that functionality is implemented correctly the first time.*

One reason for this may be that if teams were too willing to make modifications to their planned system in order to reuse functionality from other applications, undesired and unexpected changes to the system architecture will occur. As the underlying structure of the system continues to devolve, it becomes harder and harder to add changes effectively. (It is possible that, as developers get more experience with the framework, it may be possible to synchronize the design of the system more closely to the framework infrastructure from the beginning, thereby avoiding this danger. Our study did not address this possibility.)

**Can we characterize the way in which functionality was reused from the existing applications? Are there attributes of the application set that make this process more or less difficult?**

Through our observations, we identified two main categories of strategies for reusing functionality: strictly adaptive and ad hoc adaptive, which were defined in section 3.2, each with its own strengths and weaknesses. The relative effectiveness of each seems to be most strongly determined by how closely the object model of the system to be developed corresponds with the existing applications.

*Hypothesis 5: The appropriate technique for reuse depends on the distance between the functionality provided by the application set and that required by the object model of the system.*

Key functionalities 2 and 3 show that when the functionality called for by the object model is well-contained in the set of existing applications, just about any adaptation technique should be helpful. However, as illustrated by key functionality 1, a strictly adaptive technique can't take the developer far beyond what is provided by the existing applications themselves. In a situation in which the set of applications is sparse and does not contain the necessary functionality, an ad hoc technique may be more appropriate.

As key functionality 4 illustrates, if the set of applications is particularly large, then a strict adaptation technique may be most helpful. Despite its weaknesses, such a technique in procedural form was shown to guide the developer toward implementing the object model "as is" and away from "gold-plating," or spending time providing extra features that seem nice but are not necessary.

## 5. Conclusions

We described an empirical study of adapting framework-based applications in the context of parallel development projects undertaken by beginning users of the framework. The purpose of the study was exploratory, aiming to build some knowledge of reuse and maintenance of object-oriented framework applications.

It is remarkable to note that students successfully adapted functionality from existing framework-based applications to incorporate a new system, even if they had a strict deadline and they had no previous experience with the OO framework.

One limitation of our study might be that we used students as the subjects of the study but professional developers would have behaved differently. Certainly, this is always a danger in studies of this sort. However, in this case we feel that this difference would not be a strongly significant one. Although the level of industrial experience in the class was not high, all students had experience both programming in the language used (C++) as well as in object-oriented techniques. More importantly, even professional developers would almost certainly have been novices in terms of the use of the ET++ framework, so that the most immediately applicable experience would not have significantly varied in either case. A second limitation might be that our findings are tied to the framework we used, ET++. Although ET++ is a freeware framework and cannot compete in terms of usability with current visual-based frameworks running on Wintel platforms, it incorporates seventeen of the design patterns in [7]. Although generalization can be achieved only through replication in multiple studies, we believe that our findings are relevant for the class of sophisticated white-box frameworks.

Based on the evidence from qualitative and quantitative analysis, we formulated a set of hypotheses about the characteristics that contribute to the success of techniques for adapting functionality in a framework-based environment. These hypotheses should be further examined by confirmatory studies, including both case studies and controlled experiments.

## Acknowledgments

Our thanks to Gianluigi Caldiera for his help and assistance in designing and running this study. Thanks also to the students of CMSC 435 (Fall 1996) for their cooperation and hard work.

## References

[1] V. Basili, R. Reiter, Jr. A Controlled Experiment Quantitatively Comparing Software Development Approaches. IEEE Trans. Software Engineering, May 1981, pp. 299-320.

[2] V. Basili. Viewing Maintenance as Reuse-Oriented Software Development. *IEEE Software*, January 1990, pp. 19-25.

[3] V. Basili, G. Caldiera, F. Lanubile, and F. Shull. Studies on reading techniques. In *Proc. of the Twenty-First Annual Software Engineering Workshop*, SEL-96-002, pages.59-65, Greenbelt, MD, December 1996.

[4] K. Beck, R. Johnson. Patterns generate architectures. In Proc. *ECOOP'94*, Bologna, Italy, 1994.

[5] K. Eisenhardt. Building theories from case study research, *Academy of Management Review*, 14 (4), 532-550, (1989).

[6] M. E. Fayad, and D. C. Schmidt (guest editors). Object-Oriented Application Frameworks, *Comm. of the ACM*, October 1997, pp.32-87.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1995.

[8] D. Gangopadhyay, S. Mitra. Understanding frameworks by exploration of exemplars. In Proc. of the *7th International Workshop on CASE (CASE-95)*, pages 90-99, July 1995, IEEE Computer Society Press.

[9] H. G. Glaser, A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

[10] L. Hatcher, E. J. Stepanski. *A Step-by-Step Approach to Using the SAS® System for Univariate and Multivariate Statistics*. Cary, NC: SAS Institute Inc.[4], 1994.

[4] SAS® is the registered trademark of SAS Institute Inc.

[11] R. Johnson. Documenting frameworks with patterns. In *Proc. OOPSLA'92*, Vancouver BC, October 1992, SIGPLAN Notices, 27 (10), 63-76.

[12] C. M. Judd, E. R. Smith, and L. H. Kidder. *Research Methods in Social Relations*, sixth edition. Fort Worth: TX, Holt Rinehart and Winston, 1991.

[13] T. Lewis et al. *Object-Oriented Application Frameworks*. Mannings Publication Co., Greenwich, 1995.

[14] H. Mili, H. Sahraoui, I. Benyahia. Representing and querying reusable object frameworks. In Proc. *Symposium on Software Reusability*, Boston, May 1997.

[15] R. Ott. *An Introduction to Statistical Methods and Data Analysis*, Duxbury Press, Belmont, CA, 1993**.**

[16] W. Pree *Design Patterns for Object-Oriented Software Development*. ACM Press & Addison-Wesley Publishing Co., 1995.

[17] S. Rugaber, S. B. Ornburn, and R. J. LeBlanc, Jr. Recognizing design decisions in programs. *IEEE Software*, January 1990, pp.46-54.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[19] C. B. Seaman, V. R. Basili. An empirical study of communication in code inspection. In Proc. *ICSE'97*, Boston, MC, 1997.

[20] F. Shull, F. Lanubile, V. Basili. "Investigating Reading Techniques for Framework Learning." Technical Report CS-TR-3896, Dept. of Computer Science, University of Maryland, College Park. May 1998.

[21] Software Engineering Laboratory. "Recommended Approach to Software Development, Revision 3". SEL report SEL-81-305, June 1992.

[22] Taligent, Inc. *The Power of Frameworks*, Addison-Wesley, New York, 1995.

[23] J. Vlissides. *Unidraw Tutorial I: A Simple Drawing Editor*, Stanford University, 1991.

[24] A. Weinand, E. Gamma, R. Marty, Design and implementation of ET++, a seamless object-oriented application framework, *Structured Programming*, 10 (2), 1989.

[25] R. Yin, *Case Study Research: Design and Methods*, Sage Publications, London, 1994.