Using Restructuring Transformations to Reengineer Object-Oriented Systems

A Position Paper on the FAMOOS Project

Serge Demeyer, Stéphane Ducasse, Robb Nebbe, Oscar Nierstrasz, Tamar Richner[1]

Software Composition Group, University of Berne

Abstract. Applying object-oriented design methods and languages does not guarantee that the resulting software systems will be flexible and adaptable. The industrial partners in the FAMOOS project have learned this lesson the hard way: they are now faced with large and rigid software systems that hamper them in meeting a diverse and evolving set of customer requirements. Object-oriented frameworks are touted as a promising solution, but it is unclear how to transform object-oriented legacy systems into frameworks. This paper proposes an approach — i.e, a methodology and tools— for re-engineering object-oriented systems towards frameworks by means of *high-level* and *low-level restructuring transformations* that detect and resolve architectural and detailed design anomalies, and improve application flexibility and adaptability.

1. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4430. *Fax:* +41 (31) 631.3965. *E-mail:* famoos@iam.unibe.ch. *WWW:* http://iamwww.unibe.ch/~famoos/

1 Introduction

Surprising as it may seem, many of the early adopters of the object-oriented paradigm already face a number of problems typically encountered in large-scale legacy systems. In the FAMOOS project we are confronted with millions of lines of industrial source code, developed using object-oriented design methods and languages of the late 80s. These systems exhibit a range of problems, effectively preventing them from satisfying the evolving requirements imposed by their customers.

Although object-oriented design methods and programming languages are considered valuable for building flexible and adaptable systems, experience has shown that applying these techniques in isolation is not sufficient. This is now well-accepted in the object-oriented community and the state-of-the-art encourages the construction of *frameworks* [15] to cope with the wide variety of customer needs and evolving requirements. A framework is a semi-finished application architecture together with a component library, which supports the construction of families of systems for a specific problem domain. The software industry has accepted the principle of frameworks but now demands an approach — i.e, a methodology and tools— to transform their object-oriented legacy systems into object-oriented frameworks.

This paper reports on the FAMOOS approach for evolving object-oriented legacy systems into frameworks. This approach distinguishes between high-level problems in the application architecture, and low-level problems in the detailed object-oriented design. In both cases we are seeking to identify sets of useful restructuring transformations that resolve architectural and design problems, and can be used to gradually transform specific applications into more flexible, framework-based applications.

The following section presents the context of the FAMOOS project and the case studies on which we base our observations and against which we will validate our work. We then briefly introduce the readers to some terminology before describing the ideas behind the methodology and presenting the prototype tools we are developing to aid in re-engineering. Finally, we conclude with a discussion of related work and future directions.

2 The Case Studies

In the FAMOOS project, two industrial partners (Nokia, Finland and Daimler-Benz, Germany) provide real-world cases to be studied by two academic partners (University of Berne, Switzerland and Forschungs Zentrum Informatik, Germany) with the aid of tools supplied by the tool providers (SEMA Group, Spain and TakeFive, Austria).

The industrial partners have provided five legacy systems, as shown in Figure 1, written in C++ (four case studies) and Ada (one case study).

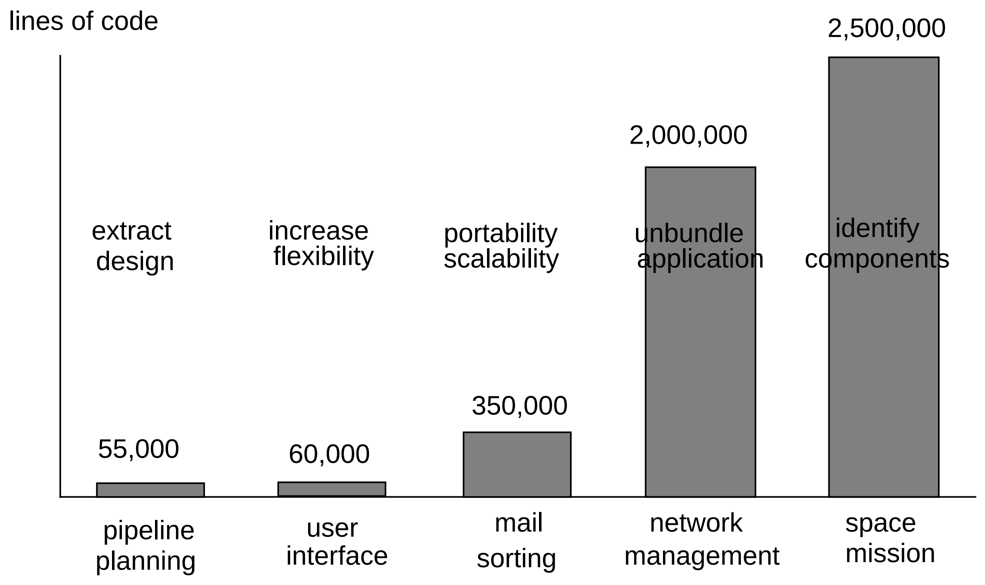

Figure 1 FAMOOS case studies

The five case studies are described briefly below:

- **pipeline planning**: This software system supports the planning of liquid flow in a pipeline between many stations. The industrial partner wants to *extract design* from source-code, in order to reduce the cost of implementing similar systems, probably in other languages.
- **user interface**: This software provides graphical representations of telecommunication networks to telecom operators. The industrial partner wants to improve portability, facilitate the addition of functionality and enhance tailorability to meet the needs of different customers, all to *increase the flexibility* of the software.
- **mail sorting**: This is software to control machines which sort surface mail. The software is highly configurable, to deal with the different ways the customers handle letters. The software itself is based on an internally developed distributed architecture but the industrial partner want to *improve the portability and scalability* of the system, and is consid-

ering adopting new technology (e.g. CORBA, Java, HTML) to achieve this.

- **cellular network management**: This is a network management system for digital networks. The main goal of the re-engineering project is to *unbundle the application*, i.e. split the system into subproducts that can be developed and sold separately.
- **space mission management**: A set of applications that —in different combinations— form systems to support the planning and execution of space missions. The industrial partner seeks to *identify components* in order to improve reliability and facilitate system maintenance.

During the first six months of the project, the industrial partners made a thorough analysis of each of the case studies and the kinds of problems they face, resulting in a 60-page report [28]. The document is an excellent source of material and we summarise here its main points as a starting point for developing our approach.

Goals and Motivations

A first observation is that the goals and motivations for re-engineering the software systems are quite diverse, yet some common themes emerge. Two case studies mention explicitly that they want to *unbundle* the software system into subsystems that can be tested, delivered and marketed separately. *Improving performance* is a goal of two case studies; two name it as a potential problem once the system is transformed into a framework. Two case studies state *porting* to other user-interface platforms an explicit target; one case study mentions it as a desirable aim but hard to achieve with the current architecture. One case study states *design extraction* as their primary goal, although all other case studies mention this as a required step in documenting the system. One case study specifies *exploitation of new technology* as a solution to the problem of scalability and portability and this theme recurs with two other case studies where the industrial partner would like to exploit new features of the programming language.

The industrial partners also specify the problems they see as obstacles to meeting these goals and to arriving at more flexible and maintainable code. Problems with the software system occur at different levels of granularity and we group these as high-level problems and low-level problems.

High-level Problems

These problems are perceived as the central issues in achieving the re-engineering goals and the industrial partners accept that solving these problems will require significant human intervention.

- *insufficient documentation*: All of the case studies face the problem of non-existent, unsatisfactory or inconsistent documentation. Tools are required which would help to generate module interfaces, maintain existing documentation and visualise the static structure and dynamic behaviour of their systems.
- *lack of modularity*: Four of the five case studies suffer from a high degree of coupling between classes / modules / subsystems that hampers further software development (compilation, maintenance, versioning, testing). A solution will involve metrics to help detect such dependencies and refactoring tools to help in resolving them.
- *duplicated functionality*: In three of the case studies several modules implement similar functionality in a slightly different way. This common functionality should be factored out in separate classes / components, but tools are missing which help in recognizing similarities and restructuring the source code.

- *improper layering*: In two case studies the user-interface code is mixed in with the "basic" functionality, creating problems in porting to other user-interface platforms. A general lack of separation, or layering, is observed with regard to other aspects (distribution, database, operating system) in other case studies. In contrast to a lack of layering, one case study suffers from unnecessary layers. Overly layered modules resulted from each successive developer encapsulating the module with a new concept instead of revising it. This problem needs tool support for recognising and correcting such design flaws.

Low-level Problems

The code must also be "cleaned up" at a lower level, in order to improve its maintainability. The industrial partners perceive many of these problems as arising from the lack of familiarity of developers with the new object-oriented paradigm. Furthermore, several years of development with sometimes geographically dispersed programming teams that have changed over time have exacerbated these problems. Table 1 shows a summary of typical problems encountered in the case studies.

Table 1: Perceived problems with Object-Oriented Concepts

	pipeline planning	**mail sorting**	**user interface**	**cellular network management**	**space mission management**
misuse of inheritance		•			
missing inheritance	•		•		•
misplaced operations					•
violation of encapsulation	•	•		•	•
missing encapsulation	•	•	•	•	•

- *misuse of inheritance*: Inheritance is used as a way to add missing behaviour to one superclass. This is a result of having a method in a subclass being a modified clone of the method in the superclass.
- *missing inheritance*: In some cases, software engineers had duplicated code instead of creating a subclass. In other parts, long case statements that discriminate on the value of a variable are used instead of method dispatching on a type.
- *misplaced operations*: Operations on objects were defined outside the corresponding class. Sometimes this was necessary in order to patch "frozen" designs.
- *violation of encapsulation*: This was observed in extensive use of the C++ friend mechanism. Also, software engineers rely on the strong typing of the compiler to ensure certain constraints, leading in some cases to redundant type definitions which contaminate the name space.
- *missing encapsulation*: This problem has been named "C style C++", although it was observed in Ada as well. It refers to the usage of the C++ classes as a structuring mechanism for namespaces. Sometimes this was necessary to interface with external systems.

Summary: Low-Level and High-Level Restructuring

From the analysis of the FAMOOS case studies, we see that a re-engineering methodology for object-oriented legacy systems must support two conceptually separate activities. The low-level problems listed in Table 1 must be addressed by what we call *low-level restructuring*: the restructuring of the code to a new, functionally equivalent system, by repairing the system's implementation to facilitate subsequent maintenance. This kind of restructuring has to do with problems that can be identified quite mechanically. Many of these problems are also encountered when restructuring non-object-oriented systems to object-oriented systems [10]. The second activity is what we call *high-level restructuring*, and addresses the technical challenges and architectural problems associated with the high-level problems. This kind of restructuring should also preserve the overall system behaviour, but requires changes to enhance the system architecture to better support future extensions. High-level structuring is an activity which requires human intervention.

In the remainder of this document, we elaborate on each of these activities to define a initial approach for transforming object-oriented legacy systems into framework-base applications. But first, we provide some insight in framework related techniques.

3 From Legacy Systems to Frameworks

Whereas a specific application is designed to meet specific application requirements, a framework provides a generic solution that can be easily adapted to different needs. In order to reengineer applications into frameworks, one must detect where application specific and generic aspects are intertwined, and attempt to separate them. This kind of work is often required in other kinds of re-engineering efforts: factoring out genericity from application specific aspects makes for better adaptability and maintainability of the software.

Frameworks

A framework is a software structure which provides a skeletal software architecture and a library of components[15], [25]. The skeletal architecture factors out the commonalities of a family of applications and can be specialized for creating a specific application. The component library provides an extensible collection of software artefacts addressing a particular application domain. Successful frameworks are usually domain specific (as opposed to completely general), so that the framework architecture is in a sense an embodiment of requirement and design knowledge for a specific application domain.

How does one acquire this application domain knowledge ? Framework development and application development are often parallel tasks. An initial framework can be used to create a new framework-based application. The specific application requirements may require an extension of the framework component library and a refinement of the skeletal software architecture. A good framework thus evolves iteratively with the acquisition of more domain knowledge and with experience in building new applications. Clearly, completed and running systems embody a great deal of domain-specific knowledge, which when recovered and coupled with experience and an understanding its problems can be used to build a framework. In FAMOOS we seek to re-engineer applications into framework-based applications, from which full-fledged frameworks can be derived.

Design Patterns

A design pattern provides a general solution to common software design problems[3], [11]. Design patterns provide design guidelines or rules of thumb. They are important in the context of re-engineering for two reasons. First the detection of design patterns [2], or near-design patterns in the code gives insight into the nature of the design problem faced. Second, design patterns provide flexible solutions to software design problems and can aid in improving the flexibility of the software [3][11].

3.1 FAMOOS Approach

In section 2 we saw that two kinds of restructuring are needed to support the re-engineering of the case studies: low-level restructuring and high-level restructuring. The former deals with repairing what is considered bad style (i.e., overuse of inheritance, missing inheritance, misplaced operations, misuse of encapsulation and missing encapsulation). The latter is concerned more with understanding and improving the architecture of a system (i.e., documentation, modularity, factoring out common functionality, proper layering). The methodological difference between the two kinds of activities relates to their automation. Low level problems can be detected mechanically, though their proper resolution may require some human intervention. High-level restructuring, on the other hand, requires a good deal of human expertise.

In the FAMOOS project, we defined a three-tiered model to support both kinds of restructuring activities. The three levels are defined as follows:

- *Source View*: A view of the system as expressed in terms of programming language mechanisms and represents a trivial interpretation of the source code.
- *Semantic View*: A language-independent view of the system representing the concepts present in the source code rather than the language mechanisms used to express it.
- *Pragmatic Views*: Semantic views are based on the semantic view augmented with knowledge not present of inferable from the source code such as coding conventions, architectural styles [29], design patterns and application or domain specific knowledge.

Having a layered model is not unusual in re-engineering. The model presented in [10] is similar to ours and reflects the need for both bottom-up (mapping from source to semantic concepts) and top-down (mapping application specific concepts to semantic concepts) recovery in program understanding.

We now discuss how our model may help to restructure object-oriented systems.

Low-Level Restructuring

In the FAMOOS project, we see low-level restructuring as a normalisation process, corresponding to the one of relational databases [9]. Normalizing a database schema improves the organization of the information but does not change the information content. Each normal form (in particular 2NF, 3NF and BCNF) formalizes certain desirable properties in terms of various kinds of dependencies. Moreover, after normalization the schema more clearly reflects the semantics of the problem domain.

Correspondingly, our semantic view focuses on different kinds of dependencies between “primitive” object-oriented concepts (classes, methods and state). Once we detected certain patterns of dependencies (by defining the appropriate recognizers [12]) in the semantic view, we can apply some restructuring operation (typically class refactoring [14]) to resolve the associated problem in the source, this way “normalising” the class hierarchy.

Partial dependencies between parts of inheritance tree and clients (i.e., different clients only use different parts of the inheritance tree) indicate misuse of inheritance. The solution is to analyse those dependencies to determine where the inheritance tree can be collapsed and where it must be refactored.

A common pattern of dependencies between unrelated classes or inside branches of controls statements (`if`, `case`, ...) indicate missing inheritance. The solution is to factor the pattern out into another superclass.

Circular dependencies between methods indicate at least one misplaced method [27]. The solution is to move the method to the correct class; however the dependencies of the clients on this method must also be updated.

Circular dependencies between methods and state indicate violation of encapsulation. The solution is to analyse those dependencies to determine where the breaking of encapsulation is actually needed and what should be made public to eliminate this need.

Partial dependencies between classes (i.e., a client will only use part of the interface defined for the class) indicates missing encapsulation. The solution is to factor out each part of that class into different classes.

High-Level Restructuring

In the FAMOOS project, we see high-level restructuring as a chain of design pattern transformations. That is, we want to detect and correct overly rigid architectural patterns that hinder flexibility.

Our approach is based on the idea of anti patterns [19] linked with corresponding resolutions in design pattern form [11]. Once we detected a certain anti-pattern (using pattern detection techniques like described in [2]) and apply the corresponding pattern we can tackle aspects of the high-level problems.

Design patterns are known to form a good basis for documenting frameworks [1][16], so once detected or applied a pattern we made a valuable contribution to the documentation problem.

Most design patterns in [11] decouple important aspects (creation, structure and behaviour) of object interaction. Thus, design pattern transformations can improve the modularity of a system and avoid duplicated functionality.

Design patterns like Observer, Bridge, Strategy [11], Layers, Pipes and Filters [3] are especially well suited for layered designs. Applying the corresponding design pattern operations incorporate a layered architecture into a software system.

Current status

At the time of writing, the FAMOOS project is still in its initial stages. The above ideas have been tested in smaller experiments [24][27] but not yet on the industrial case studies.

4 Tool Support

In order to understand a software system we regard it as essential to be able to recover multiple views of the software and to combine information obtained from different views [22][20][12]. Since none of the tools to which we currently have access allow a rich enough set of views to be generated, we are now developing a prototype tool called MOOSE as an environment to aid in the program understanding and problem detection.

The architecture of MOOSE corresponds to the basic re-engineering tool architecture [4]: an information base is generated using parsers and semantic analysers and this information base is used to extract new views of the code using queries, graph viewers, etc. The MOOSE information base explicitly represents the concepts that are present in the code, e.g. classes, methods, instances, method invocation, etc. This explicit representation allows for manipulating the code at a higher level than textual editing and allows to formulate hypotheses based on queries about the concepts present in the code.

MOOSE is an attempt to integrates third-party tools into a coherent whole. As an example, we import information from the symbol tables maintained in the Sniff+2.2.1[30] and Concerto/Audit-CC++[5] environments and export to public domain graph lay-out tools (XVCG). We are also experimenting with a wide range of analysis tools —perl-scripts, spreadsheets, query languages, prolog inference engines, graphical displayers— to test their applicability in re-engineering. Furthermore, we consider that a dynamic view of the software is also important for code understanding and plan to integrate dynamic analysis tools into MOOSE.

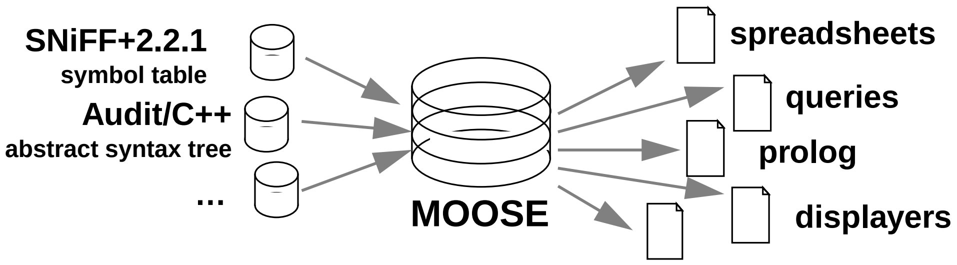

Figure 2 tool architecture

Mixing Static and Dynamic Information

A static view of software is based on the source code and consists of the classes, methods and instances which describe the software. A dynamic view is based on the software as it exists during execution and is viewed in terms of instances. The two views overlap at those instances which exist both statically and dynamically.

The mixing of static and dynamic information is invaluable for understanding program structure [22]. Dynamic information must evolve within the constraints expressed in the static information and static information can be understood at a coarser-grained level when coupled with dynamic information. Mixing both kinds of information can generate several new views of the application. For example, analysing the sequence of calls between objects can provide us with dependencies between classes and allows the detection of problems in the application [20].

5 Related Work

Though not much work is yet reported on the re-engineering of large-scale object-oriented systems, there is a growing literature on the evolution of reusable object-oriented software using refactoring transformations [14] and on the use of design patterns in program understanding[2]. Also, since some of the typical problems of first generation object-oriented systems resulted from their lack of real ‘object-orientation’, some of the approaches and techniques for migrating procedural applications to object-oriented languages [10], [13] are also relevant to the FAMOOS case studies.

Many of the problems faced in re-engineering object-oriented systems are common to any re-engineering effort. In particular, software understanding and design recovery. The ManSART tool developed at MITRE offers sophisticated source-code queries, called recognizers, used for understanding software structure and detecting architectural features[12].

Several tools for understanding or analysing object-oriented applications exist. CIA++[7] and GraphLog [6] tools focus on static information. CIA++ builds a relational database of information extracted from C++ code and provides different views of this information. GraphLog is a visual tool for databases where queries are specified by drawing graph patterns with a graphical editor. IAPR

[17] provides architectural style recognizers based on constraint programming and design pattern matching. These tools extract and manipulate static information. Several tools handle dynamic information: GraphTrace [18] offers animated views of graphs method invocations. Object Visualizer [8] and HotWire [21] analyse dynamic behaviour of applications and provide visual effects to point out application anomalies or global behaviour such as memory allocation. Such tools are interesting as profiler tools and as tools for reverse engineering. Look [23] provides calling views of C++ applications. Scene [20] is a tool for the Oberon language that extracts interaction diagrams from dynamic trace information. In contrast to the tools described above that operate on either static or dynamic information, ProgramExplorer proposes an approach in which dynamic information is used to enhance static information for program understanding [22]. In Program Explorer static and dynamic information is represented as Prolog facts derived from parsing and debugging tools. These facts serve as a database on which queries can be made to extract new abstractions.

6 Conclusions and Future Work

The case studies provided by the industrial partners of the FAMOOS project, strongly suggest the need for two levels of restructuring: low-level transformations clean up the source code, repairing and refining the structures and dependencies; and high-level transformations resolve architectural problems.

We have identified some approaches to low-level and high-level transformations: low-level restructuring transformations are based upon the detection and resolution of dependencies; high-level restructuring transformations are based upon design pattern transformations. Now that we have identified these approaches, we can validate them against the case studies, to investigate their applicability on an industrial scale.

Open issues include understanding the scalability of our approach as well as the limits of tool support. The scalability of the approach is vital since two of the case studies are more than two million lines of code. The limit of tools in terms of automation is important since human intervention will ultimately limit what can be accomplished rather than computational power.

7 Acknowledgements

We gratefully acknowledge funding from ESPRIT IV project #21975 and from the Swiss National Science Foundation, grant MHV21-41671.94 (to T. Richner) and project grant 2000-46947.96. We would also like to thank the partners in FAMOOS project for the invaluable input that formed an essential ingredient of this paper. Finally, we thank Franz Achermann for his helpful comments on the manuscript.

All authors contributed equally to the paper, as reflected in the alphabetical order of the names.

8 References

[1] Beck, K., Johnson, R. “Patterns Generate Architectures”, In *Proceedings of ECOOP’94*, LNCS 821, 139-149, Springer Verlag, 1994.

[2] Brown, K. “Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk”, Master Thesis, North California State University, 1996.

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stad, M., *Pattern-Oriented Software Architecture —A System of Patterns*, Whiley, 1996.

[4] Chikofsky, E.J., Cross, J.H. II, “Reverse Engineering and Design Recovery: A Taxonomy”, In *IEEE Software Engineering*,13-17, January 1990.

[5] Concerto2/Audit-CC++, User Manual, Sema Group, 1996.

[6] Consens, M., Mendelzon, A., Ryman, A. “Visualizing and Querying Software Structures”. In P*roceedings of the 14th International Conference on Software Engineering,* 138–156, 1992.

[7] Grass, J.E. “Object-Oriented Design Archeology with CIA++”, In *Computing Systems*. vol. 5, (1), 5–67, 1992.

[8] De Pauw, W., Kimelman, D. and Vlissides, J., “Modelling Object-Oriented Program Execution”, In *Proceedings of ECOOP'94*, LNCS 821, 163–182, 1994.

[9] Elmasri, R., Navathe, S. B. “Fundamentals of Database Systems, Second Edition”. Benjamin/Cummings, 1994.

[10] Gall, H., Klösch, R., Mittermeir, R., “Object-Oriented Re-Architecturing”, In *Proceedings ESEC ‘95, LNCS 989*

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J. “Design Patterns”, Addison Wesley, Reading, MA, 1995.

[12] Harris, D.R., Yeh, A.S., Reubenstein, H.B. “Extracting Architectural Features from Source Code”, In *Automated Software Engineering*. vol. 3, (1-2), 109–139, 1996.

[13] Jacobson, I and Lindström, F., “Re-engineering of old systems to an object-oriented architecture”. In *Proceedings of OOPSLA ‘91*, 340-350, ACM Press, 1991.

[14] Johnson, R., Opdyke, W., “Refactoring and Aggregation”. In *Proceedings of ISOTAS ’93 LNCS 742,* Springer-Verlag, 264-278, 1993.

[15] Johnson, R., Foote, B., “Designing Reusable Classes”, Journal of Object-Oriented Programming, June/July, 1988

[16] Johnson, R.E., “Documenting Frameworks using Patterns”, In *Proceedings of OOPSLA’92, ACM Press*, 63-76, 1992.

[17] Kazman, R., Burth, M.,”Assessing Architectural Complexity”, University of Waterloo,1995. http://www.cgl.uwaterloo.ca/~rnkazman/assessing.ps

[18] Kleyn, M.F., Ginrich, P.C. “GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views”. In *Proceedings of OOPSLA’88,191--204, ACM Press, 1988.*

[19] Koenig, A. “Patterns and antipatterns”. Journal of Object-Oriented Programming, March-April 1995.

[20] Koskimies, K., Mossenbock, H., “Scene: Using Scenario Diagrams and Active Test for Illustrating Object-Oriented Programs”, 366-375, In *Proceedings of the 18th ICSE*, 1996.

[21] Laffra, C., Malhotra,A. “HotWire –A Visual Debugger for C++”. In *Proceedings of USENIX C++ Technical Conference.* 109–122,1994

[22] Lange, D.B., Nakamura, Y. “Interactive Visualization of Design Patterns can help in Framework Understanding”. In *Proceedings of OPSLA'95,* ACM Press 1995.

[23] LOOK, Objective Software Technology Ltd., 1 Michaelson Square, Kirkton Campus, Livingston, Scotland,1996. http://www.objectivesoft.com/

[24] Meijler, T.D., Demeyer, S., Engel, R. “Making Design Patterns Explicit in FACE”, to appear in ESEC FSE‘97 Proceedings

[25] Meijler, T.D., Nierstrasz, O., “Beyond Objects: Components”. In *Cooperative Information Systems,* M. Papazoglou (ed.), Academic Press, London, to appear.

[26] Murphy, G. C.,Notkin, D. “Lightweight Source Model Extraction”. In *SIGSOFT’95 Proceedings,* ACM Press 1995. Available on the world-wide web at "http://www.cs.ubc.ca/spider/murphy/".

[27] Nebbe R., Richner, T., “Understanding Dependencies”, submitted to the ECOOP ‘97 Re-engineering Workshop

[28] Riepula, M. et al. “Industrial Requirements”. FAMOOS Project Deliverable D1.1. Confidential.

[29] Shaw, M., Garlan, D., “Software Architecture: Perspective on an Emerging Discipline”, Prentice-Hall, 1996

[30] Sniff+2.2.1, TakeFive Software GmbH,1996.