TOWARDS AN APPROACH FOR ASPECT-ORIENTED SOFTWARE REENGINEERING

Vinicius Cardoso Garcia, Daniel Lucrédio, Antonio Francisco do Prado
Federal University of São Carlos, Department of Computer Science
P.O. Box 676 – São Carlos, Brazil

Eduardo Santana de Almeida, Alexandre Alvaro, Silvio Romero de Lemos Meira
C.E.S.A.R. – Recife Center for Advanced Studies and Systems
P.O. Box 7851 – Recife, Brazil

Keywords: Software Reengineering, Aspect Mining, Refactoring, AOP, MVCASE, Software Transformation.

Abstract: This paper presents a reengineering approach to help in migrating pure object-oriented codes to a mixture of objects and aspects. The approach focuses on aspect-mining to identify potential crosscutting concerns to be modeled and implemented as aspects, and on refactoring techniques to reorganize the code according to aspect-oriented paradigm by using code transformations it is possible to recover the aspect-oriented design using a transformational system. With the recovered design it is possible to add or modify the system requirements in a CASE tool, and to generate the codes in an executable language, in this case AspectJ.

1 INTRODUCTION

Software reengineering is being used to recover legacy systems and allow their evolution. Several enterprises are being forced to move their legacy systems to newer languages or are looking for new ways to improve their existing software systems. This is done mainly to reduce maintenance costs, improve development speed and improve systems readability.

Current uses of reengineering include existing software development techniques, such as component-based development and object-orientation, rebuilding legacy systems into more reusable and maintainable systems. However, some limitations that are inherent to object-oriented paradigm could lead to systems that are hard to maintain and reuse. Design patterns (Gamma et al., 1995) could be used to partially overcome these limitations, but this may not be enough.

In this way, even that the system design is recovered in a high abstraction level, giving the Software Engineer a readable vision of the system functionality, its maintenance, in many cases, is still an arduous and difficult task. This could interfere with the evolution of the system in order to keep up with new hardware and software technologies.

Aspect-Oriented Software Development (AOSD) (Kiczales et al., 1997) may help to reduce this dependency, by offering a new modular unit (*aspect*). Functionalities that are necessarily dispersed in object-oriented systems, such as exception handling and logging, for example, can be grouped into a single *aspect*, increasing the modularity and the reuse level of the retrieved assets.

This paper presents a approach to help in migrating from pure object-oriented codes to a mixture of objects and aspects using reengineering and AOSD techniques, such as aspect mining, refactoring and software transformation. The reengineering product has great reuse potential, due to the benefits of AOSD.

The paper is organized as follows: Section 2 presents the Phoenix Reengineering Approach. Section 3 presents the preliminary evaluation based on the Phoenix approach and revises it based on the identified requirements. Related works are presented in Section 4. Section 5 presents some conclusion and future works.

2 PHOENIX APPROACH

2.1 Overview

The Phoenix approach aims at migrating object-oriented systems to aspects. It is based on Aspect-Oriented reverse engineering techniques and is supported by two mechanisms: a transformational system (called Draco-PUC (Leite et al., 1994)) and a modeling tool (called MVCASE (Almeida et al., 2002)). The proposed approach combines differ-

Cardoso Garcia V., Lucrédio D., Francisco do Prado A., Santana de Almeida E., Alvaro A. and Romero de Lemos Meira S. (2005).
TOWARDS AN APPROACH FOR ASPECT-ORIENTED SOFTWARE REENGINEERING.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 274-279
DOI: 10.5220/0002520302740279
Copyright © SciTePress

ent techniques based on our experience in software reengineering (Alvaro et al., 2003),(Garcia et al., 2004b).

Figure 1 shows the approach, according to the SADT notation (Ross, 1977).

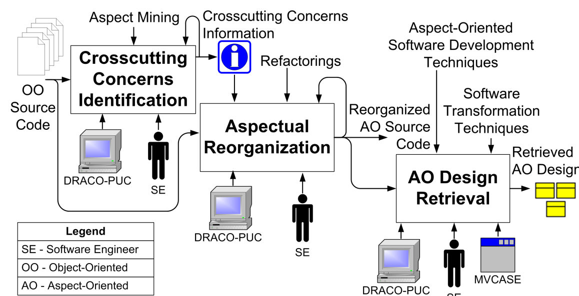

Figure 1: Reverse Engineering

In order to identify and extract crosscutting concerns in legacy systems, tree steps are performed, as follows.

Crosscutting Concerns Identification. Initially, the software engineer analyzes the legacy system aiming to identify possible crosscutting concerns that are present. These will serve as input to the aspect mining, which determines where these concerns are located inside the system. In our approach, aspect mining is performed using character sequence and regular expression analysis, and parser-based mining, implemented in the transformational system Draco-PUC. The idea is to find static join points, occurring in the context of the program, that refer to specific crosscutting concerns.

The parser-based aspect mining was performed through software transformations, implemented in Draco-PUC Transformational System, to identify the crosscutting concerns. The Figure 2 show the regular expressions to indicate the presence of exception handling (1) and database persistence (2) concerns.

In the second stage of the reverse engineering, the source code will be organized according to Aspect-Orientation principles, separating the non-functional requirements in aspects and the functional requirements in classes with their respective methods and attributes.

Aspectual Reorganization. After the crosscutting concerns are identified, the software engineer uses refactorings to extract and encapsulate these concerns into aspects. These refactorings (Garcia et al., 2004c) consider the interlaced nature of the legacy system code, and thus the transfer of individual members from classes to an aspect should not be isolated. In most cases, they are part of a set of transfers that comprise all the implementation elements of the concern that is being extracted. Such concerns typically include multiple code fragments scattered across multiple modular units (e.g. methods, classes, packages). Therefore, in many cases, more than one refactoring should be applied to extract a particular concern.

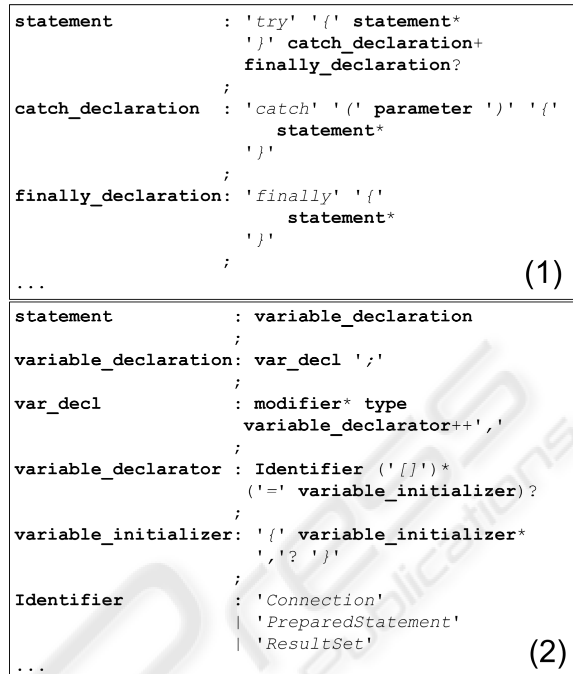

Figure 2: Exception handling and database persistence regular expressions

Aspect-Oriented Design Retrieval. Next, the software engineer, using software transformations in the Draco-PUC Transformational System, obtains the AO design, in UML descriptions. The transformations map descriptions in a programming language, corresponding to the reorganized AO code, into descriptions in a modeling language, which can be loaded into MVCASE tool. Then, the software engineering may edit the retrieved models in MVCASE, inserting minor corrections and refinements. We currently use an UML extension that is capable of representing AOSD concepts, and is implemented in MVCASE (Garcia et al., 2004a). More information on automatic design retrieval using transformations may be seen in (Alvaro et al., 2003).

Figure 3 shows an example of AO design retrieval using software transformations. The AO Source code (1) is analyzed by the transformations (2), which are responsible for mapping the code into descriptions in a modeling language (3). These descriptions are then loaded into MVCASE, becoming available for edition (4).

In the reading pattern called *LHS*, in (2), the transformer recognizes a aspect declaration, in the domain of AspectJ language. After identifying the aspect dec-

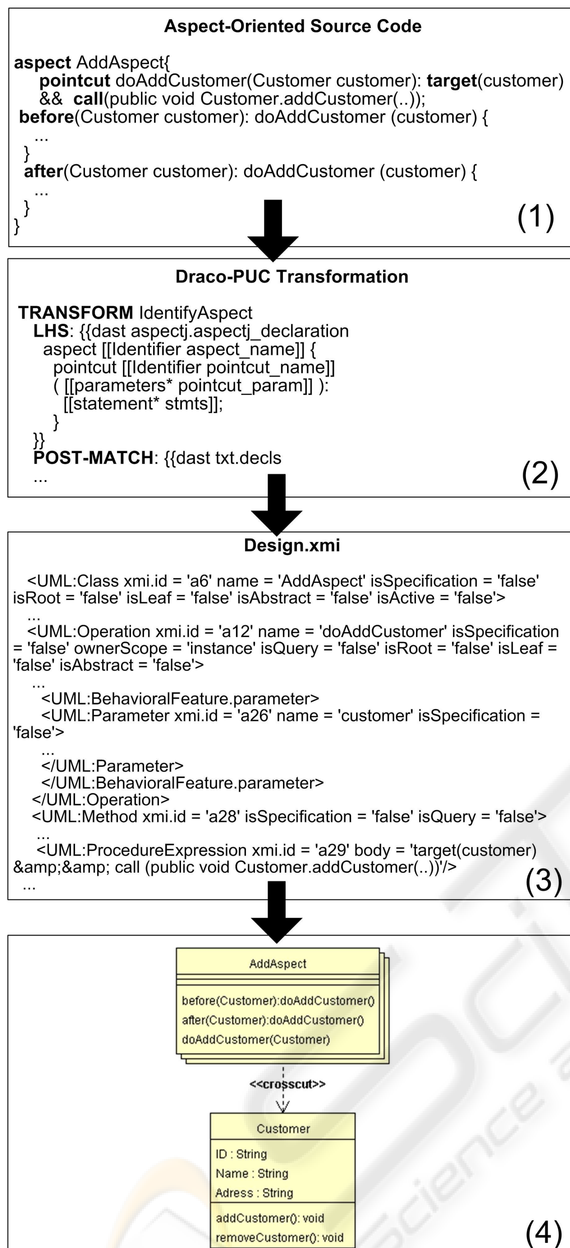

Figure 3: AO Design Retrieval

laration the control point *POST-MATCH* is executed. Following this, the written pattern, called *RHS*, is executed through a *TEMPLATE*, that persists the aspect in XMI language specifications. Later, the aspect members are also persisted in the XMI language specifications.

After that, the Software Engineer, using the CASE tool, imports the XMI descriptions containing the aspects definition to visualize the design of the legacy code. This design is represented using a class diagram with aspects-specific notation (Garcia et al., 2004a).

The retrieved design and the Aspect-Oriented source code constitute the knowledge of the legacy system. Next, this knowledge is encapsulated in highly modularized, reusable assets, with updated and consistent documentation. With the aspect-oriented design obtained from the code, the Software Engineer goes to the forward engineering, to obtain the new implemented aspect-oriented code.

Using the design obtained in the Reverse Engineering phase that was imported into MVCASE, the Software Engineer can apply modeling techniques to modify the existing functional and non-functional requirements, add new features or define the logical and physical architecture of the system components.

Once the new aspect-oriented design is finished, the Software Engineer may then implement it using an aspect-oriented language. This task is partially automated by MVCASE, through a plug-in that was developed to generate code in the target language, which in this case is AspectJ.

The generated code can then be tested and executed. If any problems are identified, the user can go back and correct them directly in the code, or in the design, since the code can be generated and tested again. This process continues until the system passes the tests. A set of unit tests and coverage tests may be created to help in maintaining the correctness of the re-constructed system.

3 PARTIAL EVALUATION

In order to obtain a partial evaluation of the Phoenix approach, a case study was performed.

The pilot project has involved the reverse engineering of a Bank Teller System, which was developed in Java. It is composed of 11 classes, where 3 contain business rules and 8 are related to Graphical User Interfaces, through the *java.swing* package. The system had approximately 2.5K lines of code.

Like most object-oriented systems, database persistence commands are dispersed through the system's classes since the separation of concerns is not always considered during the development process.

3.1 METHODOLOGY

The systems was obtained in the Internet[1]. No documentation was available other than source code comments. The system understanding was performed through its execution, which generated a document containing its main functionalities, such as:

[1] `http://www.portaljava.com.br`

i. The system allows the customer to register itself, as well as its accounts. The account information includes the initial balance, and the account type; and

ii. For the user to access his accounts, it is necessary to inform an user name and a previously stored password. After this identification, the customer informs the account number and the value that he wants to draw or deposit.

After understanding, the system was analyzed in order to identify the possible crosscutting concerns, that were interlaced and spread through the classes. In this pilot project, two concerns were identified: database persistence and exception handling.

Figure 4 shows an example of how parser-based mining may help in the identification of the exception handling crosscutting concern.

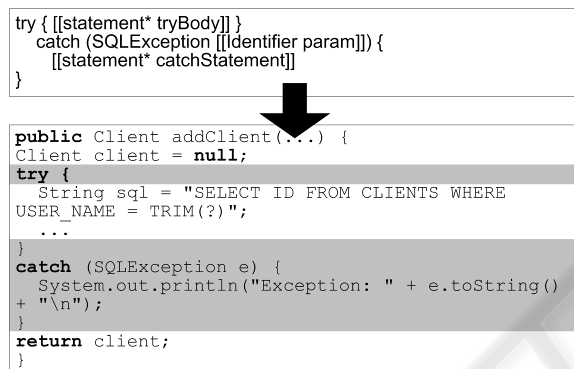

Figure 4: Exception Handling Crosscutting Concern Identification

The parser recognizes the *"try-catch"* syntactic structure in Java code and the occurrence of a *SQLException* exception type, that indicates the presence of a non-functional requirement (exception handling). This information is stored to be later consulted, in order to aid the software engineer to extract and encapsulate that crosscutting concern into aspects. However, it must be stressed that parser-based mining, as well as character sequence and regular expression analysis, is just an aid to perform the mining, which must be carried out manually by the software engineer.

Refactorings were applied to extract these concerns, and the AO Design was retrieved, both activities through transformations. In order to verify if the retrieved assets were still in conformance with the original system requirements, a new implementation of the system was performed in a forward engineering step. The new AO system was executed, and its observation verified that the functionalities of the OO system were maintained.

The Figure 5 shows an example of transformation to refactoring exception handling in source code. The *"try-catch"* syntactic structure (1) was recognized by an LHS reading pattern in the *"RefactoringExceptionByTryCatch"* transformer (2). After identifying the *"try-catch"* declaration, the control point *POST-MATCH* is executed. Following this, the written pattern (*RHS*) creates an aspect responsible to exception handling (3).

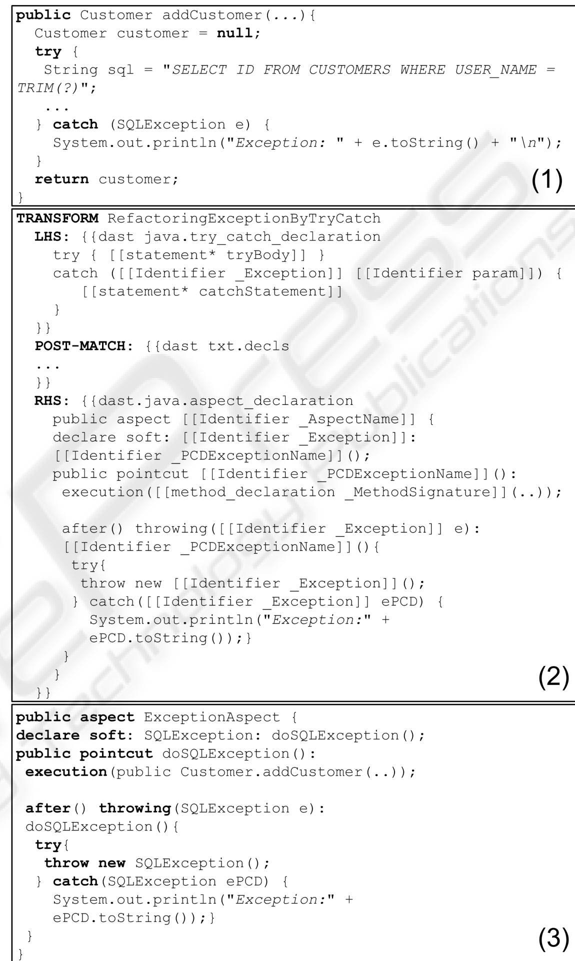

Figure 5: Exception Handling Aspectual Refactoring

Table 1 shows a brief comparison between the OO and the AO systems.

Table 1: Result Evaluation

	OO	AO
Classes	11	11
Aspects	-	6
Lines of Code (LOC)	2492	2324

The reduced number of lines of code and the increased number of modules (aspects) indicate that the retrieved assets are smaller and better divided. Since each aspect groups a single concern, the modules are also more cohesive. Therefore, we may deduct that

the retrieved assets are more reusable than the original ones.

3.2 DISCUSSION

This was only an initial study, to prove the viability of the proposed approach.

The consequences on identifying and separating crosscutting concerns are equivalent to those stated by AOSD:

i. Automation: The use of software transformations, through Draco-PUC, automates an important stage in the reengineering: the recovery of the system design. The Reverse Engineering activities, such as the crosscutting concerns identification and the aspectual reorganization, are also accelerated due to this automation;

ii. Requirements traceability: After the separation of concerns, it is easier to trace each module to a specific requirement;

iii. Easier Maintenance: Requirement changes, functionality improvements and code restructuring are also easier to perform;

iv. Readability: The new code is lighter and less polluted, because attributes and methods are not spread through the system; and

v. Reuse: The identified and extracted aspects can be implemented in such a way that they can be later reused. This may even give origin, with the accomplishment of different case studies, to a framework of aspects.

Also, some disadvantages could be observed:

Transformers construction: Since Phoenix uses a transformational system help the Software Engineer in some tasks of the process, there must exist transformers to identify crosscutting concerns, to refactoring source code and to recover design information directly from the code. However, these transformers must be first constructed, which requires great effort and knowledge about the involved languages. The time spent in the transformers construction is also a critical factor. However, it must be stressed that this effort is later reused in any legacy system written in the same language. The time reduction obtained when using these transformers in the design recovery, as discussed earlier, also justifies this effort.

4 RELATED WORK

The first relevant work involving the OO technology and retrieval of knowledge embedded in legacy system was presented by Jacobson and Lindstron (Jacobson and Lindstrom, 1991), who applied reengineering in legacy systems that were implemented in procedural languages. The authors state that reengineering should be accomplished in a gradual way, because it would be impracticable to substitute an old system for a completely new one.

Today, on the top of OO techniques, an additional layer of software development, based on components, is being established. The goals of *"componentware"* are very similar to those of OO: reuse of software is to be facilitated and thereby increased, software shall become more reliable and less expensive (Lee et al., 2003).

Among the first research works in this direction, Caldiera and Basili (Caldiera and Basili, 1991) have explored the automated extraction of reusable software components from existing systems. They propose a process that is divided in two phases. First, it chooses, from the existing system, some candidates and packages them for possible independent use. Next, an engineer with knowledge of the application domain analyzes each component to determine the services it can provide.

Investigations about AOSD in the literature has involved determining the extent that it can be used to improve software development and maintenance, along the lines discussed by Bayer in (Bayer, 2000). The AOSD can be used to reduce code complexity and tangling; it also increases modularity and reuse, which are the main problems that are currently faced by software reengineering. Thus, some works that use AOSD ideas in reengineering may be found in the recent literature.

In Kendall's case study (Kendall, 2000), existing object-oriented designs for role models are used as the starting point for reengineering with aspect-oriented techniques. In this work, it did not describe the reengineering process in full detail. They are just told the comparative results among the object-oriented code and the aspect-oriented code, target of the reengineering. The use of AOSD in this case study reduced the overall module (30 methods) and lines of code (146).

Currently, with AOSD technologies being adopted and extended, new challenges and innovations start to appear. AOSD languages, such as *AspectJ* and *AspectS*, the contributions of several research groups and the recent integration with application servers, such as *JBOSS*, demonstrate the potential of AOSD in solving real problems, including those pursued in reengineering.

5 CONCLUSIONS AND FUTURE WORK

Many software reengineering approaches have been proposed to redesign and rebuild legacy systems. The goal is to develop a global picture on the subject sys-

tem, which is the first major step toward its understanding or transformation into a system that better reflects the quality needs of the application domain.

This paper presents a migrating proposal, that integrates different techniques and mechanisms to guide and help software engineers in redesign and rebuild of software systems, using transformations from object-oriented to aspect-oriented paradigm.

This integration has the goal of guiding the user in the task of retrieving the system design, according to the aspect-oriented paradigm, willing to get a better degree of reuse and develop a system that is easier to maintain. It also helps to improve development productivity and support for changes in the requirements.

Additionally, an evaluation was accomplished to show the reengineering proccess usefulness. By following the Phoenix, it could be verified that the AOSD brings several and important benefits to software development. The way as the aspects is combined with the system modules allows the inclusion of additional responsibilities without committing the clarity of the code, maintainability, reusability, and providing a greate reliability.

As a future work, the possibility of using the Object Management Group (OMG) Model Driven Development (MDD) in the Phoenix approach is being studied to enable rapid design, development, modification and deployment of the aspect-oriented application. The idea is that through MVCASE, the Software Engineer could generate complete, working applications directly from a visual model, in this case an UML+AO specifications, and using active synchronization to keep both model and code up to date during rapid application changes.

Graphical visualization of the possible crosscutting concerns source code is also being developed. In this way, the task of identifying different concerns in the legacy system should be facilitated.

REFERENCES

Almeida, E., Bianchini, C., Prado, A., and Trevelin, L. (2002). MVCASE: An integrating technologies tool for distributed component-based software development. In *Proceedings of the 6th Asia-Pacific Network Operations and Management Symposium. (APNOMS'2002) Poster Session*. IEEE Computer Society Press.

Alvaro, A., Lucrédio, D., Garcia, V. C., de Almeida, E. S., do Prado, A. F., and Trevelin, L. C. (2003). Orion-RE: A Component-Based Software Reengineering Environment. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 248–257. IEEE Computer Society Press.

Bayer, J. (2000). Towards engineering product lines using concerns. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*.

Caldiera, G. and Basili, V. R. (1991). Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–71.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley.

Garcia, V. C., Lucrédio, D., Frota, L., Alvaro, A., de Almeida, E. S., and do Prado, A. F. (2004a). A case tool for aspect-oriented software development (*in portuguese*). In *XI Tools Section - XVIII Brazilian Symposium on Software Engineering (SBES 2004)*. ISBN 85-7669-004-7.

Garcia, V. C., Lucrédio, D., do Prado, A. F., de Almeida, E. S., and Alvaro, A. (2004b). Using aspect mining and refactoring to recover knowlegde embedded in object-oriented legacy system. In *Proceedings of the IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004)*. IEEE Computer Society Press.

Garcia, V. C., Piveta, E. K., Lucrédio, D., Alvaro, A., de Almeida, E. S., do Prado, A. F., and Zancanella, L. C. (2004c). Manipulating Crosscutting Concerns. *4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop 2004)*.

Jacobson, I. and Lindstrom, F. (1991). Reengineering of old systems to an object-oriented architecture. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 340–350. ACM Press.

Kendall, E. A. (2000). Reengineering for separation of concerns. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

Lee, E., Lee, B., Shin, W., and Wu, C. (2003). A reengineering process for migrating from an object-oriented legacy system to a component-based system. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 336–341. IEEE Computer Society Press.

Leite, J. C., Sant'anna, M., and Freitas, F. G. (1994). Draco-PUC: A Technology Assembly for Domain Oriented Software Development. In *Proceedings of the 3rd International Conference on Software Reuse (ICSR'94)*, pages 94–100. IEEE Computer Society Press.

Ross, D. T. (1977). Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34. Special collection on Requirement Analysis.