RECAST: Evolution of Object-Oriented Applications

Dr. Stéphane Ducasse
ducasse@iam.unibe.ch

Abstract

This research project is about reengineering object-oriented applications. Reengineering such applications inherits complex problems related to software maintenance, *i.e.,* program understanding, program analysis, and program transformation and adds to them (1) the complexity introduced by late binding, dynamic typing, and incremental definition specific to object-oriented programming, and (2) the complexity related to the new way of software development (multiple parallel versions, frameworks, and products lines).

Based on our research experience, this research project is structured in three non-orthogonal directions: (a) reengineering, (b) analysis of versions and (c) migration of object-oriented applications towards components.

Key words. Software Engineering, Object-Oriented Programming Reengineering, Reverse Engineering, Program Understanding, Architecture, Meta-Model, Code Analysis, Frameworks, Patterns.

1 Introduction

Most of the effort while developing and maintaining a system is spent supporting its evolution [Som96]. This document presents RECAST, a research project whose goal is to support the evolution of object-oriented applications.

The following two laws due to Manny Lehman illustrate the vision of RECAST. They stress the fact that software must continuously evolve to stay useful and that this evolution is accompanied by complexity.

Continuous Changes. *"an E-type program*[1] *that is used must be continually adapted else it becomes progressively less satisfactory"* [Leh96]

Increasing Complexity. *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*[Leh96]

RECAST is based on the vision that supporting evolution of applications will be *always* mandatory. This is independent of the language and the paradigm used to develop the applications. Tools and techniques are necessary to support the evolution of applications.

RECAST structures the research on evolution of object-oriented applications around three directions as shown in Figure 1.

[1] E-Type program: a software system that solves a problem or implements a computer application in the real world.

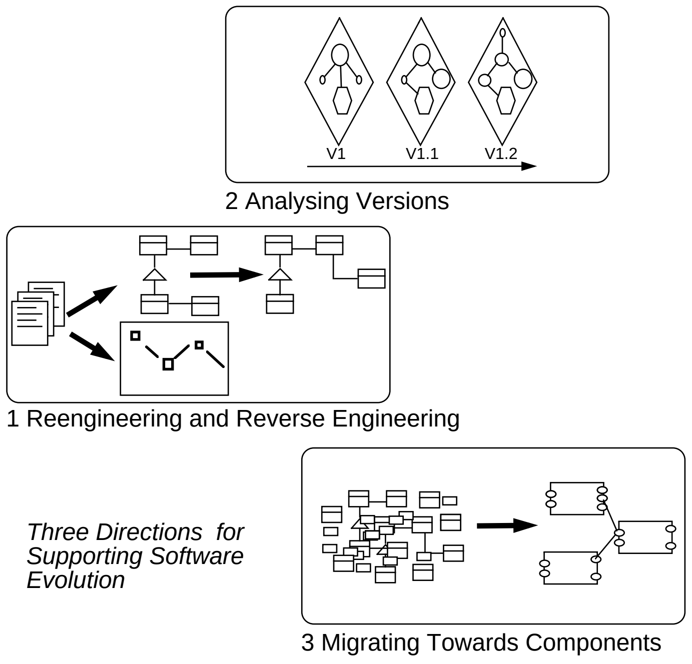
V1
V1.1
V1.2
2 Analysing Versions
1 Reengineering and Reverse Engineering
Three Directions for Supporting Software Evolution
3 Migrating Towards Components

Figure 1: The three research directions composing RECAST: reengineering, version analysis and component migration.

Reengineering. First it considers the *reengineering* aspect of evolution. It addresses questions such as: how can we understand a large industrial application, *i.e.*, reverse engineer it, how can we identify problems that hamper evolution, how can we change it, *i.e.*, reengineer the application to fix those problems.

Versions. Second, it considers multiple *versions analysis*, *i.e.*, systems are put best understood by taking into account their evolution over several versions. Hence a temporal dimension is added and used to understand applications and predict their future.

Migration. Then finally it approaches paradigm shift by investigating how the *migration* of object-oriented systems into component systems can be supported. It introduces the problematic of supporting the identification, extraction and migration of code in terms of components.

The concrete results we want to obtain within a period of three to four years are the following.

Reengineering - Meta-meta-model. One of the key aspects of this project is the new infrastructure and meta-model we want to develop. Having an extensible meta-model will support a new range of work such as the architecture extraction, program understanding, and program analysis and will allow us to create a synergy between the other research directions. This is why this will be our first goal, as presented in Section 4.3.1.

Reengineering - Supporting Code Understanding and new IDEs. We want to investigate new ways of helping programmers to understand code. The idea is to add semantic information to code fragments and integrate it into new IDEs.

Version Analysis. We want to use the version information to support the understanding of application. As presented in Section 4.4, we want to be able to analyse versions of a system with our tool CODECRAWLER.

Component Migration - A Framework for Component Description. We want to work on the migration of object-oriented code towards components. For this purpose, we want to design a framework with which different component models can be represented and compared.

Component Migration - Component Identification. We want to gain experience in the identification of components within object-oriented code. We plan to apply concept analysis and cluster analysis techniques as presented in Section 4.5.

Structure of this document. First we start by arguing that evolution and reengineering are key activities in the life of a software application. We briefly present the specific problems encountered while reengineering object-oriented code. Second, as the present project is the logical next step in our research on reengineering, we first present our contributions so far. We explain the long term vision from which this project takes its roots and we present a precise list of goals for this project. We finish by stressing the strategic aspects of the project.

2 The Software Development Reality

Software has become the key element in many areas of the computer industry, yet software development is a complex endeavor full of pitfalls and traps. Even successful projects are facing problems of evolution or software aging [Par94]. The persistant character of the problems in software development led Pressman to coin the expression *chronic affliction* rather than *software crisis* [Pre94]. Several factors inherent to software development lead to this situation: the *complexity* of the domain and tasks modelled, the need for *continuous adaptation*, the *project management and human relationship* issues and the difficulty of finding out what the customer *really* wants.

2.1 Some Software Development Facts.

Software maintenance is the name given to the process of changing a system after it has been delivered. Sommerville, referring to studies conducted in the eighties [LS80, McK84], states that large organizations devoted at least 50% of their total development effort to maintaining existing systems [Som96]. McKee in [McK84] suggests that maintenance effort is between 65% and 75% of the total effort. So maintenance remains the most expensive software development activity. However, the term maintenance is misleading because it gives the impression that this process is just dealing with bug fixes.

A finer analysis of software maintenance shows that software maintenance is often equivalent to forward engineering and not only limited to corrective maintenance [LS80], [NP90]. Maintenance activities have been categorized in three different types as follows (the percentage shows the relative effort compared with the total maintenance effort) [Som96]:

- *Corrective maintenance* (17%) is concerned with fixing reported errors in the software,
- *Adaptive maintenance* (18%) is concerned with adapting the software to a new environment (*e.g.,* platform or OS), and

- *Perfective maintenance* (65%) is concerned with implementing new functional or non-functional requirements.

Clearly most software development “maintenance” is about supporting the evolution of software.

Among the reasons that lead to software decay, the most important ones are linked with the dynamics of software itself. Lehman and Belady derived from empirical observations a set of software evolution Laws. As presented in the introduction, Lehman’s Laws identify deep reasons for the necessity of software evolution [LB85, Leh96]. The first law states that system evolution is *inevitable* if the software is successful. It stresses in particular that system requirements will always change so that a system must evolve if it is to remain useful. One of the reasons for this is that the environment is changing. The second law states that, as a system is changed, its structure is degraded. So additional efforts, on top of the ones concerned with the change, have to be expended to prevent and reverse this degradation.

2.2 Legacy Applications

Before going any further, we have to give a definition of the term *legacy*. In the context of software development, a *legacy system* is a piece of software that (1) you have inherited and (2) is *valuable* for you. Moreover, legacy systems present all the problems of aging software [Par94, Cas98]: original developers no longer available, outdated development methods, monolithic systems, code bloat, lack of documentation, misuse of language constructs, and so on.

In fact, developers would love not to reengineer applications, but they are forced to do so. Here are some reasons that force them to reengineer instead of rewriting systems.

- Legacy systems tend to be huge, so no one developer knows a complete system. After years requirements are forgotten by the developers and the users. Once the original developers leave, nobody knows the system anymore. Most of the time the documentation is hopeless or inexistent. This is why rewriting the system is nearly impossible.
- Legacy systems provide revenue to companies, while every new project represents a cost and contains a risk of failure.
- Customers were satisfied with the previous release until they wanted this new feature; they do not want to pay for a complete new development.
- Companies have neither the time nor the developers to expend the reconstruction of a system that is successful, even if its state may prevent any evolution.

2.3 Object-Oriented Applications Reengineering

While the term *legacy systems* has been coined to refer to applications written in Assembler, Cobol, or Fortran, nowadays it also describes applications written in C++, Smalltalk or Java. Indeed, even if object-oriented code can support better encapsulation and flexibility, developing applications with object-oriented technologies requires constant investment to control the intrinsic entropy that accompanies any software development. In addition to this, the lack of suitable training, developer turnover and the use of hybrid languages lead to monolithic systems that are extremely hard to maintain and sustain their evolution [DD99b, Cas98]. Moreover, as object-oriented technologies favor fast changes and incremental development, adopters of the technology want to reap these benefits and convert their monolithic applications into frameworks that are complex yet customizable and capable of evolving

[JF88]. However, building frameworks is a task that requires iteration, reengineering and careful redesign [RJ96].

With the advent of eXtreme Programming [Bec99] and other agile development models, practices such as refactorings that were previously used only in the context of reengineering now play a central role. The fact that applications are continuously changing requires developers to have tools and techniques to understand, test and change the code to meet the new requirements. This is also the case with iterative development where the design of an application may change to meet new requirements. That's why reengineering object-oriented applications is becoming more and more important.

2.3.1 Specific Problems of Object-Oriented Reengineering

Although the reasons for reengineering a system may vary, the actual technical problems are typically very similar. There is usually a mix of coarse-grained, architectural problems, and fine-grained, design problems. Object-oriented legacy systems suffer from the following traditional problems that include insufficient documentation, improper layering and adaptability, lack of modularity, and duplicated functionality.

Object-oriented programming promotes encapsulation and information hiding, which in a way should improve maintenance. However, in addition to the traditional problems enounced before like duplicated code, reengineering object-oriented languages has its own set of problems [WH92]. We list here some of the most preeminent.

- Late binding makes traditional tool analyzers like program slicers inadequate. Data-flow analyzers are more complex to build especially in presence of dynamically typed languages.

- Incremental class definition makes understanding a class more difficult. As the semantics of **self** is dynamic, understanding an application is more difficult. This dynamism of self produces yoyo effects when trying to follow which method will be executed. When an inherited method is called on an instance of a subclass, ascertaining which methods will be called necessitates checking if methods have been defined on the subclasses.

- Moreover, languages such as C++ with explicit pointer manipulations and complex syntax requires one to use complex tools and analysis. For example the parser of the SNiFF+ product is not developed by Sniff developers because of "the too complex C++ syntax" (Sic) [Bis98].

Besides these problems, the most common fine-grained problems occurring in object-oriented legacy systems are often due to misuse or overuse of object-oriented features. Here is a list of the most common problems: explicit dispatch, *i.e.,* methods are called by checking the class of the receiver explicitly instead of using polymorphism, missing inheritance, misplaced operations, *i.e.,* operations outside instead of inside classes, violation of encapsulation and class abuse, *e.g.,* classes used as namespaces.

2.4 Summary

To summarize the context and the problems of reengineering, we can say that:

1. reengineering is one of the key activities in software industry;

2. legacy systems exist in any programming languages and paradigm; and

3. reengineering object-oriented applications is an important research field because legacy systems developed using object-oriented programming languages already exist and because new software is being developed in this paradigm.

3 Building on our Previous Experience

RECAST, the project that we propose, is the logical follow-on of our research on reengineering performed in the context of the Esprit Project FAMOOS N○ 21975 and three Swiss National Science Foundation projects. We now present our accomplishments.

Contributions. Our contributions are the publication of refereed papers and theses, and the implementation of prototypes that we validated on large concrete industrial applications.

Definition of a language independent meta-model (FAMIX) [DDT99b, TDDN00, DTD01, DT01]. We needed to analyze several different object-oriented languages such as Smalltalk, Java, and C++. We designed a language independent meta-model representing the main elements of object-oriented programming languages. We put emphasis on it being extensible.

Implementation of a reengineering environment (MOOSE) [TDD00, DLT00, DLT01]. To support our research we developed a reengineering environment based on the meta-model we specified. It includes the possibility to analyze multiple models, to define dedicated program analyzers, and to load and save meta-models from different languages.

Evaluation of metric use in reengineering [DD99a, DDN00a, DLS00]. We evaluated how software metrics can support a reengineering effort. From our studies, we concluded that metrics are not reliable for detecting design flaws, that metrics are a good indicator of systems stabilization, and that they can be used to discover refactorings.

Reverse engineering large applications (CODECRAWLER) [DDL99, DL01]. We developed an approach and a tool to support the reverse engineering of large systems. The idea is to display software entities as nodes of simple graphs but to semantically enrich the obtained with metrics that describe the represented entities.

Understanding of fine grained code elements [LD01] We developed a new approach, called *the class blueprint*, to understand classes. It is based on calling relationships and a layered visualization. We developed a categorization of blueprints.

Detection of code duplication (DUPLOC) [DRD99, RDG99] We developed a technique for identifying duplicated code. Our approach is based on textual analysis but is nevertheless largely language independent. We applied it to applications written in Pascal, C++, C, Cobol, APL, Java and Smalltalk.

Use of dynamic information for extracting behavioral views [RDW98, RD99, RD01, Duc99, Duc97] We developed an iterative approach based on the mixing of dynamic information and static information specified by the meta-model we developed. Views of a system can be created and refined incrementally.

Language independent refactorings [TDDN00, Tic01, TD01] Refactorings are behavior preserving code transformation. Up until now, refactorings has been studied on a per language basis.

We developed a framework based on the Famix language independent meta-model to evaluate language independent refactorings and the limits of the approach.

Reengineering patterns [DDT99a, DDN00c, DDN00d, DDN00b, DRN99] Reengineering projects, despite their diversity, often repeatedly encounter some typical problems and solutions. We defined a pattern form to transfer reengineering expertise, and recorded *reengineering patterns*. Reengineering patterns codify and record knowledge about modifying legacy software: they help in diagnosing problems and identifying weaknesses which hinder further development of the system and aid in finding solutions which are more appropriate to the new requirements.

Overall we have published 13 papers in conference and journals on the topic, are finishing a book published by Morgan Kaufman Publisher on reengineering, made tutorials at OOPSLA and ECOOP, graduated 3 PhD students, and and act as consultant.

4 RECAST: Evolution of Object-Oriented Applications

4.1 A Vision

As we mentioned in the introduction, the vision of RECAST is that supporting the evolution of software applications will always be necessary and that software will always require energy for its maintenance and evolution. Tools and techniques are then necessary to understand, analyse and restructure applications. Such applications can be old systems as well as brand new ones for the reasons we exposed above.

Continuing the thrust of our current research, RECAST is structured around three main directions namely: reengineering, version analysis and migration towards components as shown by the figure 1. It reinforces our research on reengineering while opening it to new problems.

4.2 Concrete Goals

Within the directions we described in the introduction, we want to obtain the following concrete results within a period of three to four years.

- **Reengineering - Meta-meta-model.**
- **Reengineering - Supporting Code Understanding and new IDEs**.
- **Version Analysis.**
- **Component Migration - A Framework for Component Description.**
- **Component Migration - Component Identification.**

We now present each of these points in more details.

4.3 Reengineering and Reverse Engineering

Over the years we gained a better understanding of the needs required while reengineering applications. We want to extend our meta-model to allow the declarative definition of meta entities, and the generation of automatic functionality. Then we want to work on new ways to understand programs, to support the extraction of design and architecture.

4.3.1 Reengineering: A Declarative Meta-Model and Tools

Reasoning about the code entities is absolutely mandatory for reengineering a system. MOOSE, our reengineering environment, supports it well by allowing code visualization [DDL99] as well as code refactoring [TDDN00]. However, it is important to represent domain information that does not have a one-to-one mapping to the source code constructs. For example, Rigi, a reverse engineering tool, allows one to define a mapping from the code entities to domain entities [Mül86]. However, Rigi does not support code analysis function such as metrics computation and refactoring.

Year One and Two. Therefore we want to generalize our meta-model to allow the expression of any kind of entity. For this purpose we will define a meta-meta-model on top of which we will base our environment. We already experimented with an entity-relation meta-meta-model similar to the meta-meta-model used in CDIF [Com94]. This first experiment showed us the feasibility of the idea and its benefits, some tools in our environment were automatically able to update themselves when new entities were programmed. We plan to evaluate if using the MOF [OMG97] is more adapted to our needs. In addition we plan to have a declarative definition of the meta-entities that reengineers themselves can define by direct manipulation without having to program it. Based on this meta-meta-model we will adapt our tools like CODECRAWLER so that they automatically update themselves to new entities. This meta-meta-model will serve as a basis to compute generic metrics automatically.

Year Three and Four. Once this new meta-meta-model is built it will serve our other goals like extracting design information. First, in a top down approach, we plan to use a logic meta-programming system such as SOUL [Wuy01] to express rules describing design and architecture and develop an iterative approach like the one developed in the Reflection Model which allows the reengineer to adapt his hypothesis while extracting the information [MN97]. Second, in a bottom-up approach, we will work on the extension of CODECRAWLER [DDL99] to provide more abstract views based on the use of new entities such as applications, modules, or components and new relationships such as aggregations. For example, we want to support the extraction of design information such as UML-like relations that are not represented as constructs but represented by various idioms. Another example is that we want to be able to specify .Net components or CCM components to support higher-level view extraction.

4.3.2 Reengineering: Supporting Code Understanding

Some reengineering aspects like program understanding are close to the iterative software processus. Most of the time a developer has to understand the code he should enhance or fix. However, it is fascinating to realize that the majority of the current integrated development environments (IDEs) are *syntactically* oriented, *i.e.*, they merely highlight some constructs with special colors, advanced ones allow code navigation such as finding all the implementors or senders of a given method name.

We are convinced that simple *semantical* information can be inferred and presented to the developer to ease code understanding. One of the problems then is that not all the information is relevant and that there is no predefined information. So the developer is in a situation similar to a doctor who must propose hypotheses, choose the radiography to be done, and interpret the results.

Year One and Two. We want to work on the definition of simple heuristics that would work like code property revealers. The heuristics could be as simple as grouping all the methods of a class that access, directly or indirectly all the instance variables of that class. These heuristics should constitute an elementary set of views that can be used and combined by a developer to understand the code. This research is related to our wish to experiment with new IDEs as described below.

Year Three and Four. With rapid development life cycle and agile development methodologies, code understanding and code refactoring practices become increasingly parts of normal development and not limited to special reengineering tasks. However, for this to be effective, reengineering tools should not be disconnected from the act of developing software. This implies that the IDEs used by the developers should integrate reengineering tools as this is the case with the Refactoring Browser [RBJ97]. This is the reason why we want to conduct research on new IDEs integrating the results we obtained on code understanding and navigation and evaluate the impact on development practices. We started to work on how to manipulate and navigate software artifacts in a uniform way [WD01]. This research should bring us to define new ways of approaching coding and creating new tools for supporting the developer need.

4.4 Version Analysis

Up until now we only have considered one version of a system at a time. With this research direction we plan to bring the temporal aspect of the system evolution into consideration. Taking versions into account fits well with shorter development cycles promoted by iterative development process because they produce more information and also require tools and information to steer such iterative development.

Working on version analysis is difficult for two main reasons. One practical difficulty is that having access to several versions of industrial applications is difficult. Jazayeri et al in [JGR99] for example could only analyse changes because they only had access to a database change log and not the code itself. To circumvent this problem we started to collect versions of software such as the Swing Java framework, the VisualWorks Smalltalk framework, the Microsoft foundation class library and some open source systems like Squeak. The second reason is that analyzing multiple versions requires us to be able to analyze huge amounts of information.

Note that this direction does not exist in isolation and we expect to have a lot of synergy between the results obtained in the reengineering context.

Year One and Two. Due to the system size and complexity, and the proliferation of versions, developers often have only a partial view of a system. We want to apply the visual approach we developed [DDL99] to offer views that help the understanding of systems and their evolution. We foresee a lot of possible views due to the crossing of two aspects namely: the abstraction level and the temporal evolution.

We sketch some of the information we want to extract:

- *Qualification of artifacts.* Classes have different behaviors during their life, they can grow continuously, grow and shrink, disappear, be split or merge with others. Besides the identification of these behaviors, we want to develop a vocabulary to be able to describe them and support the understanding of the classes.
- *Movements between subsystems.* Changes between subsystems provide meaningful information about the stability of a subsystem. We expect to find a full range of behaviors.
- *Change Correlation.* Revealing the correlation between changes can bring to light hidden aspects of a system [GHJ98].

Year Three and Four. We have started to empirically evaluate how size metrics can be used to assess systems [DD99a]. We found that size metrics are not reliable for identifying problematic parts but can be used to assess the stabilization of a system. Our initial experiment was performed on three frameworks and we want to generalize this empirical validation to a representative sample and extend our experiment to other metrics such as coupling and cohesion. Frameworks are usually extracted from the design of three applications or major variations of a system [JF88, RJ96]. Frameworks are based on the definition of variation points called hotspots. We want to evaluate if a metrics analysis can support the identification of hotspots to help framework development.

4.5 Towards Components

Components are receiving a lot of attention as they provide another level of abstraction. In this context we want to provide analysis and techniques to support the migration of object-oriented applications towards component ones. This research direction is the one where we have the least experience so we plan to approach it using step-by-step experiments. We will also use the expertise of the other SCG team members with whom we will work.

To evaluate the risk of such a research direction we started some preliminary experiments to see how Envy applications (a white box way of packaging Smalltalk code) could be transformed into black box components [AD01]. This helped us to identify the following topics.

Year One and Two. Migrating towards components implies having an adapted component model that fits the purpose of the migration. Nowadays a profusion of component models exist such as COM, CCM, EJB or even proprietary models such as the one developed by Dassault Systems [FDE$^+$01]. We want to design a framework for expressing different models in terms of the composition of elementary aspects in a similar fashion to the wat that the Actalk model allows the expression of multiple concurrent object-oriented models [Bri89].

Year Three and Four. Once the target component are defined, analysis and tool support are necessary to support the identification of components [Kos00]. We started some initial experiments [AD01] and applied concept analysis to identify code patterns in object-oriented systems [SR97, LS97]. We learned that the scalability of the approach has to be addressed. We want to extend these preliminary results by reducing the scope of the search, by using other techniques such as cluster analysis [AL99] and by providing tool support.

5 Strategical Aspects

We want to stress some of the strategic aspects of this project: it is based on a community of research, it tackles really important and recurrent problems and it is concrete research with economic impact.

The Reengineering community. This project does not exist in isolation. We are currently participating to an international effort to create a European network of research groups working on reengineering. The current network is funded by the Belgium government and is called EVOLUTION (http://prog.vub.ac.be/poolresearch/FFSE/network.html). A European funded network is under construction. We are in contact with the LORE laboratory of the University of Antwerp and the LSR group of the University of Grenoble. We are participating in the standardisation of DMM, a meta-model for reengineering.

Recurrent is a recurring Task. Reengineering will always be necessary. The Laws of Lehman tend to be true in any language [Leh96]. The facts are already there, legacy applications in Java already exist. The research team Adele of the LSR laboratory of IMAG did a reengineering project with Dassault Systems to help Dassault engineers understand the application they are developing based on a component technology. LIFIA (Laboratorio de Investigacion y Formacion en Informatica Avanzada) started a research project to support the reengineering of Web-based applications.

Concrete industry related. This research is linked to concrete problems faced by industry. On the one hand it is really motivating to solve concrete problems and face real life constraints such as scalability issues. On the other hand we are conscious that we are the scientific experts, hence we have to take care and select the scientific problems we want to work on. Within this context, we are currently trying to have closer relationships with companies and evaluating how we could complement governmental research funding with funding from industry.

6 Conclusion

The RECAST research project attacks complex problems linked with the maintenance and evolution of industrial software in the particular context of object-oriented programming. The project is based on the experience we have accumulated over the years but does not limit itself to extending existing work but also opens new research avenues. It draws on a lot of different topics such as meta-modeling, code understanding, code visualization, or program analysis and transformation. Hence it provides a rich research framework but identifies clear short term goals that we will achieve. This project is relevant both from the point of view of the scientific community on which it is based but also the industrial context with which we interact.

References

[AD01] G. Arevalo and S. Ducasse. From white box to black box: an experience with envy application, 2001. Working Document.

[AL99] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE'99 (Sixth Working Conference on Reverse Engineering)*, pp. 235–256. IEEE, 1999.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. 201616416.

[Bis98] W. Bishoffberger. Private discussions concerning SNiFF+, 1998.

[Bri89] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In S. Cook, editor, *Proceedings ECOOP'89*, pp. 109–129, Nottingham, July 10-14 1989. Cambridge University Press.

[Cas98] E. Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, 1998.

[Com94] C. T. Commitee. Cdif framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, 1994. See http://www.cdif.org/.

[DD99a] S. Demeyer and S. Ducasse. Metrics, do they really help? In J. Malenfant, editor, *Proceedings LMO'99 (Languages et Modèles à Objets)*, pp. 69–82. HERMES Science Publications, Paris, 1999.

[DD99b] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Berne, 1999. See http://www.iam.unibe.ch/˜famoos/handbook.

[DDL99] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*, pp. 175–187. IEEE, 1999.

[DDN00a] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA'2000, ACM SIGPLAN Notices*, pp. 166–178, 2000.

[DDN00b] S. Demeyer, S. Ducasse, and O. Nierstrasz. A reverse engineering pattern language. In *Proceedings of Europlop'2000*, 2000.

[DDN00c] S. Demeyer, S. Ducasse, and O. Nierstrasz. Tie code and questions: a reengineering pattern. In *Proceedings of Europlop'2000*, 2000.

[DDN00d] S. Demeyer, S. Ducasse, and O. Nierstrasz. Transform conditional: a reengineering pattern language. In *Proceedings of Europlop'2000*, 2000.

[DDT99a] S. Demeyer, S. Ducasse, and S. Tichelaar. A pattern language for reverse engineering. In P. Dyson, editor, *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999*, Konstanz, Germany, July 1999. UVK Universitätsverlag Konstanz GmbH.

[DDT99b] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe and R. France, editors, *Proceedings of UML'99 (2nd International Conference on The Unified Modeling Language)*, LNCS 1723, pp. 630–645. Springer-Verlag, Oct. 1999.

[DL01] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[DLS00] S. Ducasse, M. Lanza, and L. Steiger. A query-based approach to support software evolution. In *ECOOP'2000 International Workshop of Architecture Evolution*, 2000.

[DLT00] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. 2000. COSET'2000 (International Symposium on Constructing Software Engineering Tools).

[DLT01] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. The Smalltalk Chronicles, 2001.

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pp. 109–119. IEEE, September 1999.

[DRN99] S. Ducasse, T. Richner, and R. Nebbe. Type-check elimination: Two object-oriented reengineering patterns. In F. Balmas, M. Blaha, and S. Rugaber, editors, *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*. IEEE, 1999.

[DT01] S. Ducasse and S. Tichelaar. Lessons learned in designing a platform for software reengineering. Technical Report, University of Berne, 2001.

[DTD01] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Berne, 2001. to appear.

[Duc97] S. Ducasse. Des techniques de contrôle de l'envoi de messages en Smalltalk. *L'Objet*, 3(4):355–377, 1997.

[Duc99] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, 1999.

[FDE+01] J.-M. Favre, F. Duclos, J. Estublier, , R. Sanlaville, and J.-J. Auffret. Reverse engineering a large component-based software product. In *Proceedings of CMSR'2001 (Conference on Software Maintenance and Reengineering)*, pp. 95–104, 2001.

[GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM'98)*, pp. 190–198, 1998.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JGR99] M. Jazayeri, H. Gall, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1999.

[Kos00] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universitat Stuttgart, 2000.

[LB85] M. M. Lehman and L. Belady. *Program Evolution - Processes of Software Change*. London Academic Press, 1985.

[LD01] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA'2001*, 2001.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology*, pp. 108–124, 1996.

[LS80] B. P. Lientz and E. B. Sawson. *Software Maintenance Management*. Addison-Wesley, 1980.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceegins of ICSE'97*. IEEE, 1997.

[McK84] J. R. McKee. Maintenance as a function of design. In *Proceedings of AFIPS National Computer Conference*, pp. 187–193, 1984.

[MN97] G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[Mül86] H. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[NP90] J. T. Nosek and P. Palvia. Software maintenance mamagement: changes in the last decade. *Software Maintenance: Research and Practice*, 2(3):157–174, 1990.

[OMG97] Object Management Group. Meta object facility (MOF) specification. Technical Report ad/97-08-14, Object Management Group, Sept. 1997.

[Par94] D. L. Parnas. Software aging. In *Proceedings of International Conference on Software Engineering*, 1994.

[Pre94] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[RBJ97] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RD99] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pp. 13–22. IEEE, September 1999.

[RD01] T. Richner and S. Ducasse. Iterative recovery of collaborations and roles in dynamically typed object-oriented languages. Technical report, University of Berne, 2001.

[RDG99] M. Rieger, S. Ducasse, and G. Golomingi. Tool support for refactoring duplicated oo code. In *Object-Oriented Technology (ECOOP'99 Workshop Reader)*, number 1743 in LNCS (Lecture Notes in Computer Science). Springer-Verlag, 1999.

[RDW98] T. Richner, S. Ducasse, and R. Wuyts. Understanding object-oriented programs with declarative event analysis. In *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, number 1543 in LNCS (Lecture Notes in Computer Science). Springer-Verlag, 1998.

[RJ96] D. Roberts and R. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of Pattern Languages of Programs (PLOP'96), Allerton Park, Illinois*, 1996.

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.

[SR97] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proceedings of International Conference on Software Maintenance*. IEEE, 1997.

[TD01] S. Tichelaar and S. Ducasse. Pull up/push down method: an analysis. Currently submitted to IEEE Transaction on Software Engineering, 2001.

[TDD00] S. Tichelaar, S. Ducasse, and S. Demeyer. Famix: Exchange experiences with cdif and xmi. 2000. Proceedings of WOSEF'2000.

[TDDN00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*. IEEE, 2000.

[Tic01] S. Tichelaar. *Meta-Model for Reengineering*. PhD thesis, University of Berne, 2001.

[WD01] R. Wuyts and S. Ducasse. Software classifications: a uniform way to support flexible IDEs, 2001. Working Paper.

[WH92] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.