Reengineering Object-Oriented Applications

Stéphane Ducasse

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-03-008

September 11, 2003

Abstract

Reengineering object-oriented applications is becoming a vital activity in today industry where the developer turnover drains the system oral memory out of the systems themselves and where applications should constantly evolve to meet new requirements. This document summarizes the research effort led on reverse engineering and reengineering object-oriented legacy systems. It includes (1) the definition of a suitable meta-model for reengineering, FAMIX. This meta-model, even if flat, supports both reverse engineering and code refactoring analysis, (2) the presentation of a reengineering platform, MOOSE, (3) the evalution of software metrics for reengineer, (4) the definition of simple visual techniques to support large system understanding or finer grain code element, (5) the identification and cure support for duplicated code, (6) the use of dynamic information to support composable views and collaboration extraction, and (7) the identification of reengineer patterns. Keywords. Meta-Modeling, Language Independence, Reengineering, Reverse Engineering, Code Duplication, Reengineering Patterns, Program Traces, Dynamic Information, Program Visualization, Software Metrics, Refactorings, Interexchange Format, CODECRAWLER, FAMIX, MOOSE, FAMOOS, Smalltalk, Java, C++.

Classification: 68-02 Research Exposition, 68N30 Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc.) [New MSC2000 code] 68U35 Information systems (hypertext navigation, interfaces, decision support, etc.) [New MSC2000 code] ACM: D.2 Software Evolution D.2 Software Engineering, D.2.2 Tools and Techniques, D.2.7 Distribution and Maintenance

Rétro-Conception d'Application à Objets
Reengineering Object-Oriented Applications

Stéphane Ducasse

Rapport Scientifique présenté pour l'obtention de l'

Habilitation à diriger des recherches en Informatiques

Université Pierre et Marie Curie (Paris 6)

Soutenue le 21 Septembre 2001

Jury :

T. D'Hondt, Professeur, Vrije Universiteit Brussels, Belgique, (relecteur)
J. Estublier, Directeur de Recherches CNRS, Université de Grenoble, France, (relecteur)
R. Johnson, Professeur, University of Illinois, USA, (relecteur)
J.-P. Briot, Directeur de Recherches CNRS, Université de Paris 6, France
P. Cointe, Professeur, Ecole de Mines de Nantes, France
O. Nierstrasz, Professeur, Université de Berne, Suisse
J.-F. Perrot, Professeur, Université de Paris 6, France

Résumé / Summary

La ré-ingénièrie des applications est devenue une activité vitale pour l'industrie dans un contexte où les changements d'emplois appauvrissent les applications en détruisant la connaissance orale détenue par les développeurs et où les applications doivent constamment évoluer afin de satisfaire de nouveaux besoins. Ce document résume un effort de recherche sur la rétro-conception et la ré-ingénièrie des applications orientée objets. Il décrit (1) la définition d'un méta-modèle adapté à la ré-ingénièrie, FAMIX. Ce méta-modèle permet non seulement des opérations de rétro-conception mais aussi l'expression de transformation de code (refactorings), (2) la présentation d'une plate-forme de ré-ingénièrie, MOOSE, qui sert de base à un grand nombre de nos travaux, (3) l'évaluation de métriques logicielles pour la ré-ingénièrie, (4) la définition de techniques visuelles pour la compréhension de très grandes applications et de classes, (5) l'identification de code dupliqué de manière indépendante des langages et des solutions pour le traiter, (6) le mélange d'information dynamiques et statiques pour la génération de vues composables et d'information de collaboration et (7) l'identification de patterns de ré-ingénièrie.

Mots Clès. Méta-modèlisation, multiple langages, ré-ingénièrie, rétro-conception, duplication de code, traces dynamiques de programmes, information dynamique, visualisation de programmes, métriques, transformation de code, refactorings, format d'échange de modèles, CODECRAWLER, FAMIX, MOOSE, FAMOOS, Smalltalk, Java, C++.

Reengineering object-oriented applications is becoming a vital activity in today industry where the developer turnover drains the system oral memory out of the systems themselves and where applications should constantly evolve to meet new requirements. This document summarizes the research effort led on reverse engineering and reengineering object-oriented legacy systems. It includes (1) the definition of a suitable meta-model for reengineering, FAMIX. This meta-model, even if flat, supports both reverse engineering and code refactoring analysis, (2) the presentation of a reengineering platform, MOOSE, (3) the evalution of software metrics for reengineer, (4) the definition of simple visual techniques to support large system understanding or finer grain code element, (5) the identification and cure support for duplicated code, (6) the use of dynamic information to support composable views and collaboration extraction, and (7) the identification of reengineer patterns.

Keywords. Meta-Modeling, Language Independence, Reengineering, Reverse Engineering, Code Duplication, Reengineering Patterns, Program Traces, Dynamic Information, Program Visualization, Software Metrics, Refactorings, Interexchange Format, CODECRAWLER, FAMIX, MOOSE, FAMOOS, Smalltalk, Java, C++.

For Florence,
Quentin and Thibaut

Acknowledgements

This habilitation presents some of the work done by the PURE (Program Understanding and Re-Engineering) team, a part of the Software Composition Group from University of Berne. The presented habilitation could not have been possible without the SCG Team. Thanks for all the energy you gave me.

I would like to thank the following SCG members: Serge Demeyer, Matthias Rieger, Michele Lanza, Tamar Richner, Sander Tichelaar, Robb Nebbe, Georges Golomingi, Roel Wuyts, Nathanael Schärli, Gabriela Arevalo, Andreas Schlapbach, Juan-Carlos Cruz, Thomas Hofmann, Lukas Steiger, Daniele Talerico, Pietro Malorgio, Daniel Schweizer, Mickael Freidig, Daniel Frey, Franz Achermann, Manuel Günter, Isabelle Huber and Therese Schmid and for all the discussions, papers, experiments, jumps into the Aare, jokes we had and cookies we ate.

And last but not least Oscar Nierstrasz for having hired me, suggesting me to write such a habilitation, and most important for always giving me the freedom I need.

I would like to thank the habilitation jury members to have spent some of their summer time reading this document.

Finally, I would like to thank Florence for her understanding when I come back home so late, when I'm not listening but occupied with so many ideas floating around, and for having followed me to Switzerland, Quentin and Thibaut for not letting me work when it is time to have fun.

Contents

1 Introduction (version française) 13
1.1 Applications à objets léguées 13
1.2 Objectifs et contexte 14
1.3 Contributions 17

2 Introduction 19
2.1 Object-Oriented Legacy Applications 19
2.2 Goals And Context 20
2.3 Contributions 23

3 Object-Oriented Reengineering: Context and Problems 27
3.1 The Software Development Reality 27
3.2 Reengineering Life-Cycle 29
3.3 Why not Simply Throw it Away? 30
3.4 Object-Oriented Applications Reengineering 32
3.5 Conclusion 33

4 A Meta Model for Reengineering 35
4.1 Problem 35
4.2 State of the Art 36
4.3 Why FAMIX? 36
4.4 The FAMIX Meta Model in a Nutshell 39
4.5 FAMIX Validation: The MOOSE Refactoring Engine 40
4.6 Future Tracks 43
4.7 Contributions 43

5 MOOSE: A Reengineering Environment 45
5.1 Requirements for a Reengineering Environment 45
5.2 State of the Art 46
5.3 MOOSE 47
5.4 Tools Interoperability and Meta-Meta Description 51
5.5 Validation 55
5.6 Ongoing Work 56
5.7 Future Tracks 57
5.8 Contributions 57

6 Reverse Engineering Very Large Systems 59
6.1 Problem and Constraints 59
6.2 State of the Art 60
6.3 Combining Metrics and Graphs 61
6.4 Useful graphs 62
6.5 Experimental Validation 63
6.6 A Methodology 64
6.7 Tool support: CODECRAWLER 68
6.8 Ongoing Work 70
6.9 Future Tracks 70
6.10 Contributions 71

7 Supporting the Understanding of Fine Grained Code Elements 73
7.1 Problem: Code is Just Code 74
7.2 State of the Art 74
7.3 Class BluePrints 75
7.4 Ongoing Work 79
7.5 Future Tracks 80
7.6 Contributions 81

8 Iteratively Recovering Behavioral Views 83
8.1 Problem 83
8.2 State of the Art 84
8.3 Iteratively Recovering High-Level Views from Static and Dynamic Information . . . 85
8.4 Generating High-Level Views 86
8.5 Validation and Discussion 88
8.6 Implementation Issues 89
8.7 Ongoing Work 90
8.8 Future Tracks 90
8.9 Contributions 90

9 Metrics for Reengineering 93
9.1 Problems 93
9.2 State of the Art 94
9.3 Metrics For Problem Detection and Stability Assessment 94
9.4 Heuristics for Evolution Understanding 97
9.5 A Query Engine for Evolution Recovery 100
9.6 Future Tracks 100
9.7 Contributions 101

10 Language Independent Detection of Code Duplication 103
10.1 Duplicated Code 104
10.2 State of the Art 104
10.3 Our Approach: Language Independence 106
10.4 Some Empirical Experiments 108
10.5 Supporting Understanding and Reporting Duplication 110
10.6 Ongoing Work 116

10.7 Future Tracks 116
10.8 Contributions 116

11 Supporting Object-Oriented Duplicated Code Cure 119
11.1 State of the Art 119
11.2 Steering Refactorings 120
11.3 SUPREMO 121
11.4 Lessons Learnt 125
11.5 Future Tracks 126
11.6 Contributions 126

12 Patterns: Transferring Unit of Expertise 129
12.1 The Need for Reengineering Patterns 129
12.2 Reengineering Patterns. 130
12.3 State of the Art on Reengineering Patterns and Comparison 130
12.4 An Example: Transform Conditionals to Registration 132
12.5 Contributions 138

13 Conclusion 141
13.1 Evaluation 141
13.2 Potential Research Axes in Reengineering 145

Chapter 1

Introduction (version française)

“In 1508, Pope Julius II commissioned Michelangelo Buonarroti to paint the ceiling of the Cappella Sistina (Sistine Chapel) in Rome. Michelangelo labored five years to complete the task. In 1538, Pope Paul III called him back to add an enhancement, the Last Judgment, over the altar; this took him seven more years. Michelangelo’s work on the Sistine Chapel is a legacy – a gift from the past that would be impossible to recreate and warrants preserving. In 1965, the Holy See commissioned a team of scientists, historians, and artists to restore the paintings. Though the restoration decision stirred controversy, work proceeded over the next 30 years. Although no software compares to Michelangelo’s masterpieces, the restoration process offers striking parallels to software reengineering.” [RW98]

Cette habilitation constitue le résumé de la recherche que nous avons menée dans le contexte de la *ré-ingénièrie des applications à objets* durant une période de cinq ans à l’Université de Berne. Ce chapitre présente un survol de ce travail, résume nos contributions et propose un fil rouge pour la lecture de ce document.

1.1 Applications à objets léguées

Avant de continuer nous devons définir le terme *legué*, en anglais *legacy*, car ce terme possède une connotation très particulière dans la communauté génie logicielle et ré-ingénièrie en particulier.

legacy : –A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor. Source Oxford English Dictionary

Dans le contexte du développement, un système légué (*legacy system*) est une application dont (1) vous avez hérité et (2) qui a une grande *valeur*. De plus, les systèmes *légués* présentent les problèmes caractèristiques des systèmes prenant de l’age [Par94], [Cas98] : développeurs originaux partis, documentation obsolète, vieille méthode de développement, systèmes monolithiques, code dupliqué, mauvaise utilisation de la sémantique du langage....

Bien que le terme *legacy systems* originellement décrivent des applications développées en Assembler, Cobol, ou Fortran, il caractèrise admirablement bien des applications écrites en C++, Smalltalk et Java. En effet, même si la programmation orientée objet permet une meilleure encapsulation et flexibilité, développer des applications à objets nécessite un investissement constant pour controler l'entropie qui accompagne tout développement. De plus, le manque de formation adéquate, le changement de développeurs ou l'utilisation de langages hybrides produisent des systeèmes monolithiques qui sont extrêmement difficiles à maintenir et à faire évoluer [DD99b], [Cas98]. Avec l'apparition des frameworks [JF88], de nombreuses sociétés ayant des systèmes peu flexibles ont voulu les transformer en frameworks. Ainsi des sociétés comme Nokia, Daimler-Chrysler ou AEG qui ont adopté la programmation orientée très tôt et font face à de tels problèmes, ont participé au projet Esprit FAMOOS dont le but était de permettre l'évolution de tels systèmes. Mais cette demande d'évolution ne se limite pas aux applications développées avec des langages orientés objets, d'autres sociétés comme Dassault Systems sollicitent des collaborations avec des équipes de recherches comme l'équipe Adèle du LSR (Logiciels, Systèmes et Réseaux de l'IMAG) afin de les aider dans leur développement à base de composants. Le laboratoire LIFIA de l'Universidad Nacional de La Plata en Argentine débute un projet de recherche sur la rétro-conception des applications liées au Web.

Les travaux présentés sont les résultats d'un effort de recherche conduit au sein du Software Composition Group (SCG) de l'Université de Berne depuis en Octobre 1996 jusqu'à présent. Le SCG participa au projet ESPRIT FAMOOS de Octobre 1996 à Octobre 1999. Ce travail a été aussi financé par le gouvernement suisse avec les projets de la fondation suisse de recherche scientifique suivants:

- *Meta-models and Tools for Evolution Towards Component Systems*, NSF Project No. 20-61655.00 from Oct 2000–Sep 2002.
- *A Framework Approach to Composing Heterogeneous Applications*, NSF Project No.20-53711-.98. Oct 1998–Sept 2000.
- *Framework-based Approach for Mastering Object-Oriented Software Evolution (FAMOOS)*, ESPRIT Project 21975 Swiss BBW No. 96.0015. Sep 1996–Dec 1999.
- *Infrastructure For Software Component Frameworks*, NSF Project No. 2000-46947.96. Oct 1996–Sep 1998.

1.2 Objectifs et contexte

Lors de l'écriture de de document nous nous sommes fixés les objectifs suivants:

1. Présenter un état de l'art condensé de chacun des aspects que nous avons abordé.
2. Décrire notre recherche.
3. Identifier de futures axes de recherche dans le contexte de la ré-ingénièrie.

Nous énumérons les contraintes auxquelles nous avons fait face et qui ont guidées notre recherche :

Code source comme seule matière. La seule source d'information avec laquelle nous pouvons travailler est le source des applications. En effet, les documents sont la plupart du temps absents ou obsolètes [Par94].

De multiple langages. Les applications sur lesquelles nous avons travaillées étaient écrites en différents langage principalement C++, Ada, Smalltalk et Java.

Taille et validation de nos outils. Les systèmes industriels sont en général énormes, plusieurs millions de lignes de code ou plusieurs milliers de classes. Ainsi la prise en compte de la taille des systèmes est un facteur important de notre recherche. En particulier, c'est pourquoi nous avons explicitement fait attention à valider nos outils sur du code industriel.

Simplicité. Comme le sujet en lui-même était nouveau (peu d'équipe de recherche avaient abordées une problèmatique aussi large) et comme aucun d'entre nous n'était familier avec ce domaine de recherche, nous avons essayé les approches les plus simples afin d'obtenir une compréhension de la problèmatique très tôt et garder les techniques plus lourdes comme l'analyse de flôt de données (data flow analysis), le découpage de programmes (slicing) lorsque nous aurons atteint les limites des approches simples.

1.2.1 Un fil rouge pour la ré-ingénièrie

Chikosky and Cross définissent la ré-ingénièrie comme *"the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*[CC90]. Comme montré par cette définition la ré-ingénièrie couvre un espace de recherche vaste qui inclut la représentation du code, la méta-modélisation, les formats d'échange de modèles, la compréhension de programme, l'analyse de code, l'utilisation d'information dynamique, la transformation de code, les métriques et bien d'autres aspects touchant à l'analyse de programmes.

La structure logique d'un effort de ré-ingénièrie débute par la représentation des constructeurs présents au niveau du langage, puis viennent des tâches de rétro-conception des applications et d'analyse suivies par celles de transformation. Plus précisèment nous avons quatre phases logiquement liées :

Source code représentation. Tout outil de ré-ingénièrie pour un langage non-réflexif ou pour plusieurs langages doit avoir une représentation des entités qui constituent les programmes analysés. Ceci implique de représenter les entités, de les extraire du code source, de les sauvegarder, de pouvoir les échanger entre plusieurs outils et de les manipuler lors d'analyses.

Rétro-conception (reverse engineering). *Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction.* [CC90]. La rétro-conception représente toutes les activités qui permettent de comprendre une application. Cela inclut la compréhension de programme, l'extraction de conception et d'architecture ainsi que la visualisation de programmes.

Analyse. Cette phase regroupe les approches qui permettent d'analyser les programmes. Elle inclut par exemple des activités liées aux métriques, à l'analyse de cluster, ou tout autre analyse permettant d'identifier des problèmes dans les applications.

Transformation de code. Cette phase inclut principalement la transformation des applications afin de palier les problèmes identifiés. Dans un contexte objet, les refactorings tiennent une place prépondérante.

1.2.2 Structure de notre recherche et de ce document

Ce document suit la structure logique d'un effort de ré-ingénièrie comme décrit plus avant à l'exception du travail sur les refactorings que nous présentons comme une validation du méta-modèle. Ceci est illustré par la figure 1.1.

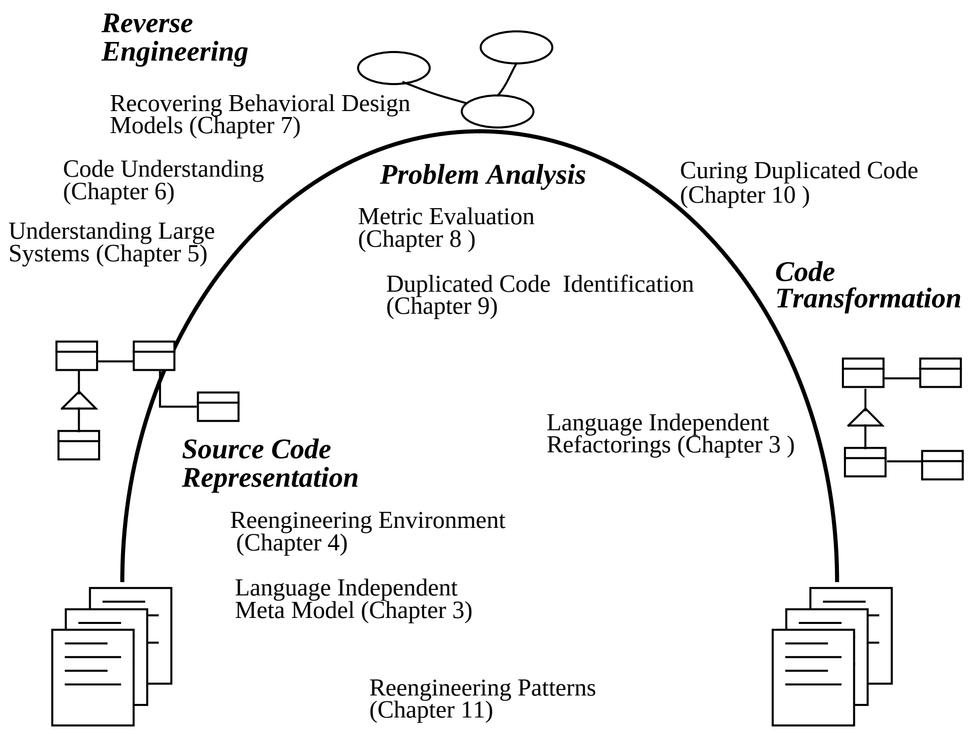

Reverse Engineering
Recovering Behavioral Design Models (Chapter 7)
Code Understanding (Chapter 6)
Understanding Large Systems (Chapter 5)
Problem Analysis
Metric Evaluation (Chapter 8)
Duplicated Code Identification (Chapter 9)
Curing Duplicated Code (Chapter 10)
Code Transformation
Language Independent Refactorings (Chapter 3)
Source Code Representation
Reengineering Environment (Chapter 4)
Language Independent Meta Model (Chapter 3)
Reengineering Patterns (Chapter 11)

Figure 1.1: le fil rouge de notre recherche présenté sous forme d'un cycle de ré-ingénièrie.

Dans le chapitre 3, nous présentons tout d'abord le contexte de ce travail. Ensuite, au chapitre 4 nous présentons le méta-modèle que nous avons conçu ainsi que sa validation au travers de l'expression de refactorings. Le chapitre 5 décrit l'environnement de ré-ingénièrie, MOOSE que nous avons construit et sur lequel un nombre de travaux ont été batis.

Nous présentons les différentes approches que nous avons développées pour la rétro-conception : la compréhension de très grands systèmes au chapitre 6, la compréhension de classes au chapitre 7, et l'utilisation d'information dynamique pour la génération de vues composables au chapitre 8.

Ensuite nous abordons les travaux liés à l'analyse des applications : au chapitre 9 nous abordons notre utilisation de métriques pour la qualification des applications ainsi que pour la compréhension de leur évolution, au chapitre 10 nous abordons la détection de code et son traitement au chapitre suivant 11

Nous finissions par un aspect plus méthodologique de notre travail et présentons au chapitre 12 les patterns de ré-ingénièrie que nous avons identifiés et décrits. Le dernier chapitre présente les axes de recherches que nous aimerions poursuivre.
Pour aider la lecture de ce document nous avons suivi une structure pour chaque chapitre à l'exception du premier. Chaque chapitre contient :

- une description du *problème* rencontré,
- un *état de l'art* condensé,
- notre *solution* au problème,
- nos travaux *en cours* et

- nous concluons par une liste de *futurs axes de recherches* et nos *contributions*.

1.3 Contributions

Les contributions présentées ci-après sont le résultat de ma recherche personnelle ou de recherches que j'ai supervisées dans le cadre de thèses ou DEAs.

- *Un méta-modèle indépendant des langages* (FAMIX) ([DDT99b], [TDDN00], [DTD01], [DT01]). Comme nous devions analyser différents langages Smalltalk, C++, Ada, nous avons défini un méta-modèle permettant de représenter les aspects centraux des langages à objets et d'étendre ce méta-modèle.

- *Implantation d'une plate-forme de rétro-conception* (MOOSE) ([TDD00], [DLT00], [DLT01]). Nous avons construit une plate-forme permettant d'extraire, charger, stocker et d'analyser plusieurs modèles simultanément, de calculer des métriques, de définir des analyseurs spécialisés. MOOSE est utilisé par Xavier Alvarez qui poursuit une thèse sur la compréhension de systèmes à objets à l'Ecole des Mines de Nantes dans l'équipe Objet dirigée par le professeur Cointe.

- *Evaluation de l'usage de métriques en rétro-conception* ([DD99a], [DDN00a], [DLS00]). Nous avons évalué comment de simples métriques pouvaient guider des efforts de rétro-conception ou de développement itératif de frameworks. Nous avons conclu que les métriques simples de taille n'étaient pas fiables pour la détection d'erreur de conception. Par contre, ces métriques pouvaient être utilisés comme indicateur de l'amélioration de la stabilité du système étudié et pour la compréhension des transformations apportées entre différentes versions.

- *Compréhension de grands systèmes* (CODECRAWLER) ([DDL99], [DL01]). Nous avons développé une approche pour permettre de comprendre (reverse engineer) de grands systèmes. L'idée est d'afficher les entités logicielles comme des nœuds des graphes triviaux et d'enrichir ces graphes en associant aux nœuds des tailles représentant des métriques des entités qu'ils représentent. L'outil développé, CODECRAWLER, est basé sur FAMIX et MOOSE. CODECRAWLER a été utilisé lors de l'analyse d'applications industrielles chez Nokia en Finlande (70000 LOC, 1.2 Million LOC), sur le code de la société MediaGenix en Belgique (3000 classes) et sur du code Cobol (13 MB) à la demande d'une société.

- *Compréhension de Classes* (CODECRAWLER) ([LD01]). Nous avons développé le concept de blueprint pour aider à la compréhension des classes. Nous avons ainsi construit un vocabulaire basé sur ces blueprints.

- *Détection de code dupliqué* (DUPLOC) ([DRD99], [RDG99]) [RDG99]). Nous avons développé une approche lexicale de détection de code dupliqué. Avec DUPLOC, le prototype qui valide notre approche, nous détectons ainsi la duplication de code écrit dans les langages : Smalltalk, C++, Java, Python, Cobol, C et APL. DUPLOC a été utilisé sur des applications industrielles chez Daimler-Chrysler en allemagne (80000 LOC), Nokia en Finlande (70000 LOC, 1.2 Million LOC) et sur du code Cobol (13 MB) à la demande d'une société. Pour valider empiriquement nos résultats nous avons appliqué DUPLOC sur plus d'une vingtaine d'applications industrielles ou académiques comme SWING, Mozilla ou GnuCC.

- *Utilisation d'information dynamique pour l'extraction de vues* ([RDW98][RD99], [RD01a], [Duc99], [Duc97]). Nous avons développé une approche itérative de constructions de vues des applications tirant partie d'une part de l'information statique et d'autre part, d'information dynamique. GAUDI, l'outil développé, est basé sur FAMIX.

- *Refactorings indépendants des langages* ([TDDN00], [Tic01], [TD01]). Les *refactorings* sont des transformations de code dans le contexte des langages à objets conservant le comportement des applications. Jusqu'à présent les refactorings avaient été étudiés par langage. Basé sur notre méta-modèle, nous avons développé une analyse de refactorings indépendants des langages.

- *Reengineering Patterns* ([DDT99a], [DDN00c], [DDN00d], [DDN00b], [DRN99]). Malgré leur diversité les projets de rétro-conception se ressemblent. Nous avons défini une forme de schémas (patterns), dans le sens des Design Patterns[GHJV95], afin de rassembler et transférer la connaissance des experts en rétro-conception.

Outils. Nous considérons que les outils sont les tangibles résultats de notre recherche et qu'ils nous aident dans notre recherche à vérifier et à réaliser nos hypothèses. C'est pourquoi nous les présentons au même titre que les articles de recherches. Cependant, nous faisons une distinction entre un outil et un prototype de recherche. Pour nous, un outil peut être utilisé par une personne autre que son auteur. Trois prototypes de recherche ont le status d'outils et ont éte validés sur du code industriel lors de consulting : MOOSE, CODECRAWLER and DUPLOC. Ils sont disponibles à : http://www.iam.unibe.ch/∼pure/ et ont été téléchargés plus que nous nous y attendions.

J'ai développé MOOSE avec Serge Demeyer et plus récemment Sander Tichelaar and Michele Lanza [DLT00]. Je suis le principal développeur des nouvelles fonctionalités et ré-ingénieur de système. MOOSE est basé sur le méta-modèle FAMIX [DTD01]. MOOSE a servi de base à de nombreux autres prototypes comme mooseexplorer, MOOSE FINDER, GAUDI ou SUPREMO ou outils comme CODECRAWLER dans notre groupe de recherche. Ill a aussi été utilisé par Xavier Alvares de l'Ecole de Mines de Nantes durant sa thèse. Des consultants de Daedalos AG utilisent nos outils. Nous avons une proposition de Cincom (anciennement ObjectShare-ParcPlace) pour inclure nos outils dans leur CD, ce que nous allons accepter une fois MOOSE porté sur VisualWorks 5.i.
Sous notre supervision,

- DUPLOC a été développé sur sa thèse par Matthias Rieger pour identifier du code dupliqué.

- basé sur MOOSE, CODECRAWLER a été développé par Michele Lanza durant sa thèse de fin de diplôme et sa thèse pour permettre la compréhension de très large systèmes.

- basé sur MOOSE, SUPREMO a été développé par Georges Golomingi durant sa thèse de fin de diplôme. Il permet de comprendre la duplication de code dans le contexte de langage à objets.

- basé sur FAMIX et MOOSE, GAUDI a été développé par Tamar Richner durant sa thèse. Il permet de définir des vues sur les applications en utilisant de l'information dynamique générée lors d'executions.

- basé sur MOOSE, la machine de refactoring de MOOSE a été développée par Sander Tichelaar.

- basé sur MOOSE, MOOSE EXPLORER a été développé par Pietro Malorgio durant sa thèse de fin de diplôme. Il permet de naviguer et analyser du code.

- basé sur MOOSE, MOOSE FINDER a été développé par Lukas Steiger durant sa thèse de fin de diplôme. Il permet d'analyser différentes versions d'un système.

Chapter 2

Introduction

"In 1508, Pope Julius II commissioned Michelangelo Buonarroti to paint the ceiling of the Cappella Sistina (Sistine Chapel) in Rome. Michelangelo labored five years to complete the task. In 1538, Pope Paul III called him back to add an enhancement, the Last Judgment, over the altar; this took him seven more years. Michelangelo's work on the Sistine Chapel is a legacy – a gift from the past that would be impossible to recreate and warrants preserving. In 1965, the Holy See commissioned a team of scientists, historians, and artists to restore the paintings. Though the restoration decision stirred controversy, work proceeded over the next 30 years. Although no software compares to Michelangelo's masterpieces, the restoration process offers striking parallels to software reengineering." [RW98]

This habilitation is a summary of the research we made in the context of *object-oriented reengineering* over a period of five years at the University of Berne. This chapter presents the context of this work in a nutshell, summarizes our contributions and proposes a roadmap of the document for the reader.

2.1 Object-Oriented Legacy Applications

Before going any further, we have to give a definition of the term *legacy*.

legacy : –A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor. Source Oxford English Dictionary

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 refer to systems that present 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...

While the term *legacy systems* has been coined to refer to applications written in Assembler, Cobol, or Fortran, nowadays it can definitively describe applications written in C++, Smalltalk or Java. Indeed, even if object-oriented programming can support a 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, the developer turnover or the use of hybrid languages lead to monolithic systems that are extremely hard to maintain and to sustain their evolution [DD99b], [Cas98].

Moreover, as object-oriented technologies favor fast change iterations and incremental development, adopters of the technology want to get the promise and convert their monolithic applications into frameworks.

European companies like for example Nokia, Daimler-Chrysler or AEG, that were early adopters of the object-oriented technologies, are facing the challenge of migrating their object-oriented legacy systems into more flexible systems [DD99b], aiming in the long term to transform them into frameworks [JF88] [RJ96]. This is the reason why they participated in the FAMOOS Esprit project (A Framework based Approach for Mastering Object-Oriented Systems) as case study providers.

The work presented here is the result of a research effort lead in the Software Composition Group (SCG) of the University of Berne over a period of time that started in October 1996 until this habilitation was printed. The SCG participated in the FAMOOS project from October 1996 to October 1999. This work has been founded by the Swiss Government over the following Swiss Science Foundation projects :

- *Meta-models and Tools for Evolution Towards Component Systems*, NSF Project No. 20-61655.00 from Oct 2000–Sep 2002.
- *A Framework Approach to Composing Heterogeneous Applications*, NSF Project No.20-53711-.98. Oct 1998–Sept 2000.
- *Framework-based Approach for Mastering Object-Oriented Software Evolution (FAMOOS)*, ESPRIT Project 21975 Swiss BBW No. 96.0015. Sep 1996–Dec 1999.
- *Infrastructure For Software Component Frameworks*, NSF Project No. 2000-46947.96. Oct 1996–Sep 1998.

2.2 Goals And Context

While writing the present document we set ourselves three goals:

1. Present a condensed state of the art in the context of each of the research aspects involved in object-oriented reengineering.
2. Relate the research we have been leading in the context of the reengineering of object-oriented applications.
3. Identify future axes of research.

The constraints we faced and that guided this research were the following ones:

Source code as primary matter. The only source of information we could work with was the code. The documentation are most of the time inexistent [Par94], even sometimes comments are inexistent or obsolete[1].

[1]In one of the FAMOOS case study, we were analyzing a 600 line C++ method responsible for process management with no comment except for only one single variable. We asked the last developer before he left the company to explain the comment. After scrutinizing the code, the developer told us that the variable was simply obsolete.

Multiple languages. In the context of FAMOOS and with the other projects we have been involved we have to reengineer applications written in different object-oriented languages like C++, Smalltalk, Ada and Java. So we wanted to build tools that are able to deal with different languages.

Scalability and tool validation. Legacy systems tend to be huge, over million lines of code or couple thousand of classes, the scalability of the proposed approaches is then an important parameter. It is for this reason that we put a particular emphasis on validating our ideas by building scalable tools.

Simplicity. As the topic itself was rather new (few research teams have been working on the topic from a such broad view at that time) and since none of us was familiar with the subject, we applied the simplest thing that may give results and kept heavy techniques like data flow analysis, program slicing for the time when we would be facing the limits of simplicity.

2.2.1 Reengineering Roadmap

Chikosky and Cross define reengineering as *"the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*[CC90]. As shown by this definition reengineering covers a broad research area. It includes source representation, meta-modelling, interexchange formats, program understanding, code analysis, refactorings, metric computation, program visualization, use of dynamic information, program slicing, cluster analysis, data mining, and many others can be used to support reengineering research.

The logical structure of a reengineering effort starts by concerns regardings the representation of the code elements, then reverse engineering and analysis activities can be applied followed by restructuring ones. More precisely the four logically linked areas are:

Source code representation. Any reengineering tools for non-reflective languages or multiple languages has to have access to representation of code entities. Dealing with code of applications implies to support the representation of code entities, their extraction for source code, their storage, their exchange with other tools, and finally their manipulation.

Reverse engineering. *Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction.* [CC90]. It represents all activities that support the understanding of an application. This includes: program understanding, design/architecture extraction, or program visualization.

Problem analysis. It groups approaches that focus on identifying problems in legacy systems by using various techniques like metric computation, or clustering mechanims.

Code transformation. Code transformation includes mainly how code can be transformed to implement the solutions identified earlier. In an object-oriented programming context it means refactorings.

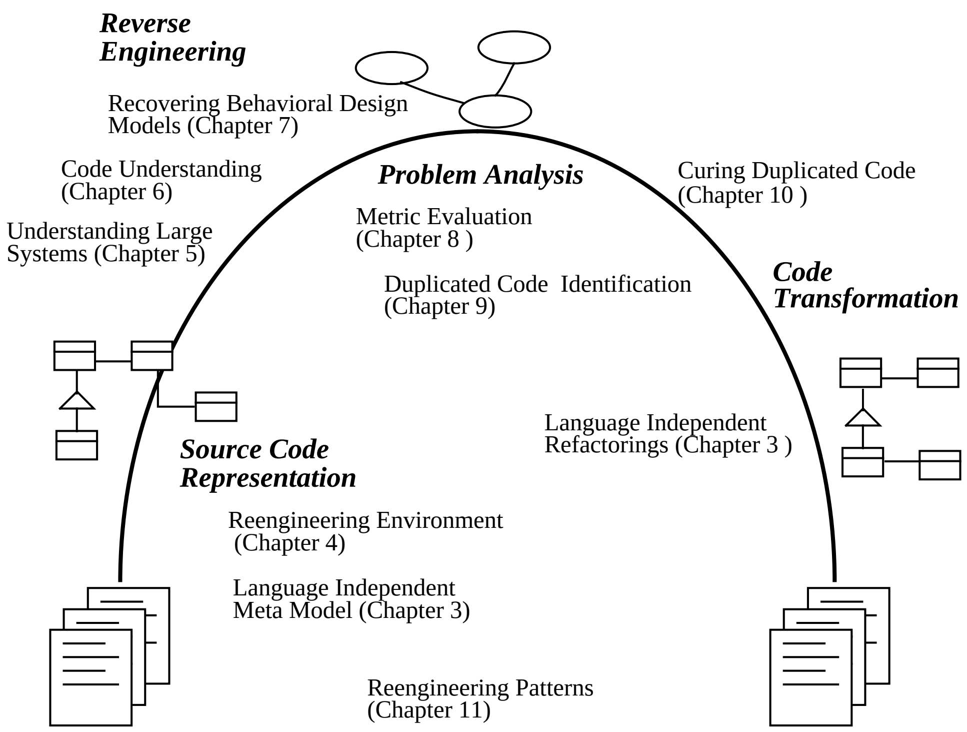

Reverse Engineering
Recovering Behavioral Design Models (Chapter 7)
Code Understanding (Chapter 6)
Understanding Large Systems (Chapter 5)
Problem Analysis
Metric Evaluation (Chapter 8)
Duplicated Code Identification (Chapter 9)
Curing Duplicated Code (Chapter 10)
Code Transformation
Language Independent Refactorings (Chapter 3)
Source Code Representation
Reengineering Environment (Chapter 4)
Language Independent Meta Model (Chapter 3)
Reengineering Patterns (Chapter 11)

Figure 2.1: Roadmap of the research presented in the context of a reengineering life-cycle.

2.2.2 Structure of our Research and this Document

Our research follows the logical structure of a reengineering effort that we presented above. This habilitation follows also this decomposition with the exception of the work on refactorings that we present as a validation of the meta-model as shown by the Figure 2.1. We first present the meta-model we designed to represent code and the environment we built to allow application analysis. We present several approaches we developed in the context of reverse engineering. Then we describe the different analysis we elaborated such as duplicated code identification. Finally we conclude with the code transformation research. In addition we report the work we made at a higher level of abstraction or “process” level in which we identified and recorded recurrent patterns occuring during reengineering efforts.

Problem and Context. First we present the problem and its relevance in the context of software engineering and object-oriented programming. In particular, we stress that *maintenance* which represents a major part of software development effort is a misleading term to denote an activity that mainly consists in enhancing systems (Chapter 3).

A Language Independent Meta-Model. Since we needed to be able to reason about source code and since we were dealing with different languages, we designed FAMIX, a language independent source code meta-model. To validate the expressivness of the meta-model we worked on *language independent refactorings.* Refactorings have been traditionally developed on a per language basis. We investigated if it is possible to reuse the analysis and code transformation by defining language independent refactorings (Chapter 4).

A Reengineering Platform. We implemented MOOSE— a reengineering platform that represents and support code analysis. Doing so we worked on interexchange format facilities (Chapter 5).

Understanding Large Systems. To support the first contact with large systems, we developed an approach based on the combination of metrics and simple graphs, and a methodology (Chapter 6).

Code Understanding. After helping in understanding the system as a whole, we present an approach we developed to support the understanding of finer code elements (Chapter 7).

Recovering Behavioral Design Models. We studied how dynamic information can enrich static information to support the understanding of application. In particular, we defined an iterative approach based on composable queries that generate specific behavioral views or role extraction (Chapter 8).

Metrics Evaluation. As metrics are scalable and legacy systems tend to be huge, we studied if metrics could be a good way to reduce the amount of information. In particular, we present the analysis we performed to evaluate if and how metrics can be useful to detect problems, to assess the evolution of an application or to detect the refactorings that occurred between different versions (Chapter 9).

Duplicated Code Identification. Duplicated code is a common problem. We work on its identification and support for understanding it in a language independent manner (Chapter 10).

Support for Duplicated Code Cure. In the specific context of object-oriented programming languages we present how a simple analysis supports the understanding of duplication and helps developers to steer the application of refactoring (Chapter 11).

Reengineering Patterns. Finally, we elaborate why reengineering patterns are important to record units of knowledge in the context of reengineering (Chapter 12).

Conclusions and Future Research Axes. We conclude by elaborating new research axes related to reengineering.

To ease the reading of this document we follow a similar structure during every chapter, expect the first ones. In every chapter we present

- *the problem* encountered,
- a condensed *state of the art*,
- Then we will present *our solution* to the problem,
- the *ongoing work* we are doing,
- and conclude with a list of *future research axes* and *contributions*.

2.3 Contributions

The following contributions are the result of my personal research or research that I supervised such as student projects, Master Theses or PhD Theses.
The contributions in the domain of reengineering object-oriented systems are:

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, 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 several 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 to detect design flaws, that metrics are a good indicator of systems stabilization and that they can be used to recover refactorings.

Reverse engineering large applications ([DDL99], [DL01]). We developed an approach 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 graphs with metric values of the represented entities.

Understanding of Fine Grained Code Elements ([LD01]) We developed a new approach, called class blueprint, to understand classes. It is based on calling relationships and a layered visualization. We developed a categorisation of blueprints.

Detection of code duplication ([DRD99], [RDG99]) We developed an approach to identify duplicated code. Our approach is based on textual analysis thus limiting the language dependence.

Use of dynamic information for extracting behavioral views ([RDW98][RD99], [RD01a], [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 to now, refactorings has been studied in a per language basis. We developed a framework based on the language independent meta-model we developed to evaluate language independent refactorings and the limit of the approach.

Reengineering Patterns ([DDT99a], [DDN00c], [DDN00d], [DDN00b], [DRN99]) Reengineering projects, despite their diversity, often encounter some typical problems and solution again and again. 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.

Artifacts. We consider that tools or software artifacts are the tangible representation of research ideas and that tools can support the elaboration of these ideas. That's why we list them as contribution of these research effort. However, we make a distinction between research prototype that only the developer of the prototype can use and tools that can be used by external persons. We consider that three prototypes we developed have the status of academic tools: MOOSE, CODECRAWLER and DUPLOC. These tools are free at: http://www.iam.unibe.ch/∼pure/ and have been downloaded more times than we expected.

I developed the MOOSE reengineering environment with Serge Demeyer, and recently Sander Tichelaar and Michele Lanza [DLT00]. It is based on the FAMIX language meta-model [DTD01]. MOOSE has been the foundation of several other tools such as CODECRAWLER, MOOSE EXPLORER, MOOSE FINDER in our research group but also it has been used as a platform by Xavier Alvares of the Ecole des Mines de Nantes in the context of his PhD Thesis and is currently used by the LoRE of the University of Antwerp. Some of the consultants of Daedalos AG are using our tools. We have a proposition of Cincom to include MOOSE into their release CD, for this we will probably convert MOOSE from VisualWorks 3.0 to VisualWorks 5.i.
Under our supervision,

- Duploc has been developed during his PhD Thesis by Matthias Rieger to identify duplication of code.
- based on MOOSE, CODECRAWLER has been developed by Michele Lanza during his Master Thesis and his PhD Thesis that allows one to reverse engineer large systems.
- based on MOOSE, SUPREMO has been developed by Georges Golomingi during his master Thesis that allows one to understand object oriented code duplication.
- based on FAMIX and MOOSE, GAUDI has been developed by Tamar Richner during her PhD Thesis to support program understanding using dynamic information.
- based on MOOSE, the MOOSE Refactoring Engine has been developed by Sander Tichelaar that supports language independent refactorings.
- based on MOOSE, MOOSE EXPLORER has been developed by Pietro Malorgio during his Master Thesis that allows one to navigate and analyze source code to find defects and possible improvements.
- based on MOOSE, MOOSE FINDER has been developed by Lukas Steiger during his Master Thesis that allows one to query the evolution of software systems.

Chapter 3

Object-Oriented Reengineering: Context and Problems

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

In this chapter we set up the context of this work by presenting some definitions, by explaining why reengineering is needed to rescue valuable software applications. We discuss that maintenance is a misleading name for a process that includes extensions and continuous improvements of applications. Then we show that reengineering is not limited to old legacy applications and that reengineering can also have its place in the context of iterative development.

3.1 The Software Development Reality

Software has become the key element in main areas of our industry, yet software development is a complex endeavor full of pitfalls and traps. Even successful projects, the over-budgeted ones that finally delivers their expectations, are facing problems of evolution or software aging [Par94]. The persisting character of the problems in software development led Pressman to coin the expression *chronic affliction* rather than *software crisis* [Pre94]. Several inherent facts to software development complicate it: the *complexity* of the domain and tasks modelled, the need of *continuous adaptation*, the *project management and human relationship* issues and the difficulty to find out what the customer *really* wants.

Software Development Facts. *Software maintenance* is the name given to the process of changing a system after it has been delivered. However, this term is misleading in the sense that it gives the impression that this process is just dealing with bug fixes. In fact, Sommerville referring to studies made in the eighties [LS80], [McK84] states that large organizations devoted at least 50% of their total development effort to maintaining existing systems [Som96]. [McK84] suggests that maintenance effort is between 65% and 75% of the total effort. Even if up-to-date figures are hard to find, maintenance remains the most expensive software activity.

In addition 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].

[1]E-Type program: software systems that solve a problem or implement a computer application in the real world.

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

We see that a majorityof the software development and maintenance is devoted to the evolution of software.

Reasons. There are simple reasons that explain such a situation. Sommerville points that (1) maintenance staff are often inexperienced and unfamiliar with the application domain, (2) the program being maintained may have been developed several years ago and have lost its original structure, if it got any, (3) changes made can produce errors hence produce further change requests, (4) as a system changes its structure tends to degrade, and (5) programs and documentation get desynchronized. Parnas in [Par94] that states that only the bad programs that nobody wants to use do not have to be changed, distinguishes two distinct reasons for software aging: *lack of movement* and the *ignorant surgery*. The first one results in failure to update to new requirements while the second is the result of changing the software without knowing it enough. The decay of code is so normal that Foote and Yoder identify patterns during software development that produce legacy applications [FY00].

One of the obvious reason is the categorization of activities into development versus maintenance is wrong. The data themselves tends to prove it. This wrong categorization is backlog of the waterfall model that was the one of the most used development model. The waterfall model decomposes development into a single directed flow of activities where the last one is maintenance. Such a sequential decomposition of activities prohibits the necessary interaction and feedback required by development and addressed by further development models such as the spiral model [Boe88] and to its extreme eXtreme Programming [Bec99].

There are deeper reasons for software decay which are linked with the dynamics of software itself. Lehman and Belady derived from empirical observations a set of software evolution Laws [LB85], [Leh96]. Two of the Lehman's Laws identify deep reasons for the need of software evolution [LB85], [Leh96].

> **Continuing Change.** "*An E-type program 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]

The first law states that system maintenance is inevitable. It stresses in particular that system requirements will always change so that a system must evolve if it is to remain useful. One to the reason 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 in addition to the ones concerned with the change, have to be done to preserve this degradation.

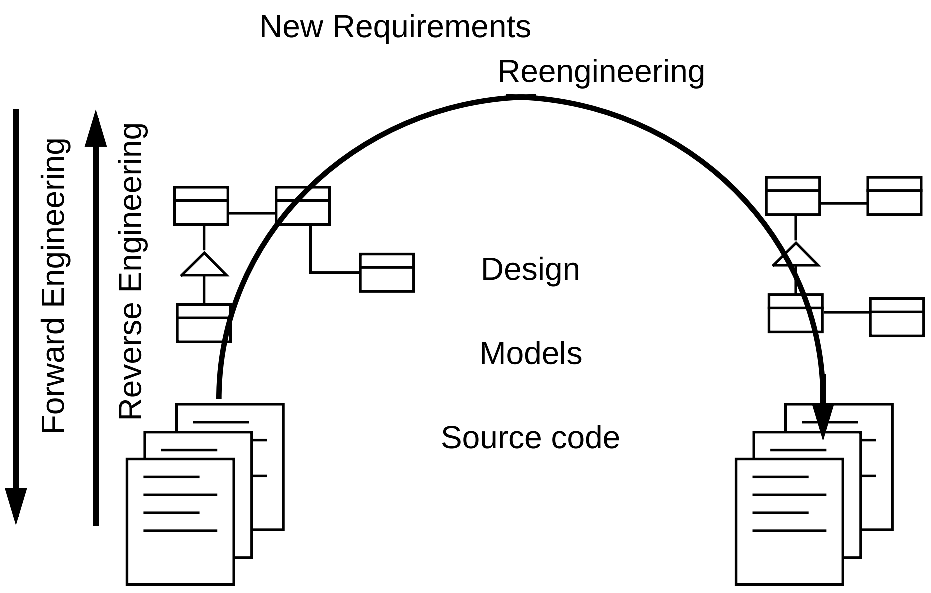


Figure 3.1: Reengineering, Reverse Engineering and Forward Engineering in Context.

3.2 Reengineering Life-Cycle

Up to now we have indifferently used the terms software evolution and reengineering. We now define more precisely some terms.

Reengineering and reverse engineering are often mentioned in the same context, and the terms are sometimes confused. Chikofsky and Cross define the two terms as follow:

> *"**Reverse engineering** is the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction."*[CC90]
>
> *"**Reengineering** is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."*[CC90]

The introduction of the term reverse engineering is clearly an invitation to define forward engineering:

> *"Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."*

If forward engineering is about moving from high-level views of requirements and models towards concrete realizations, then reverse engineering is about going backwards from some concrete realization to more abstract models, and reengineering is about transforming concrete implementations to other concrete implementations. In Figure 3.1 we see this illustrated.

In a typical legacy system, you will find that not only the source code, but the documentation and specifications are out of sync. Reverse engineering is therefore a prerequisite to reengineering since you cannot transform what you do not understand. You carry out reverse engineering whenever you are trying to understand how something really works. Normally you only need to reverse engineer a piece of software if you want to fix, extend or replace it. As a consequence, reverse engineering efforts typically focus on documenting software and identifying potential problems, in preparation for reengineering.

Casais in [Cas98] defines a reengineering life-cycle in 5 steps that can be mapped to the ones described in Figure 3.1; (1) Model capture (documenting and understanding the design of the legacy system), (2) Problem detection (identifying violations of flexibility and quality criteria), (3) Problem analysis (selecting a software structure that solves a design defect), (4) Reorganization (selecting the optimal transformation of the legacy system) and (5) Change propagation (ensuring the transition between different software versions).

Reengineering similarly entails a number of interrelated activities. Of course, one of the most important is to evaluate which parts of the system should be repaired and which should be replaced. According to Chikofsky and Cross *restructuring* generally refers to source code translation (such as the automatic conversion from unstructured spaghetti code to structured,or goto-less code), but it may also entail transformations at the design level.

> *"Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the system's external behavior.*"[CC90]

Refactoring is restructuring within an object-oriented context. Martin Fowler defines it this way:

> *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*"[FBB $^+$99]

It may be hard to tell the difference between software *reengineering* and software *maintenance*. IEEE has made several attempts to define software maintenance, including this one: *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment*. Most people would probably consider that *maintenance* is routine whereas *reengineering* is a drastic, major effort to recast a system, as suggested by Figure 3.1. However practitioners of eXtreme Programming [Bec99] would argue that reengineering is included in the way they develop software.

3.3 Why not Simply Throw it Away?

> *"Old systems that must still be maintained are sometimes called* legacy systems. *The amount of code in these legacy systems is immense. In 1990, it was estimated that there were 120* billion *lines of source code in existence.*"[Som96]

Often people believe that reengineering is not used in industry because industrials rewrite application from scratch. Although such a statement could be true in most of the cases this is often not the case. The quote above illustrates the relevance of the problem. In fact, developers would love not to to reengineer applications but they are forced to do so. Here are some simple reasons that force them to reengineer instead of rewriting systems:

- Legacy systems tend to be huge, so no one developer know a complete system. Requirements are forgotten since years by the developers and the users. Original developers left, so nobody knows the system anymore. Most of the time the documentation is hopeless or inexistent. It is why rewriting the system is nearly impossible.

- Legacy systems are providing revenue to companies, while every new project contains a risk of failure.
- Customers usually were satisfied with the previous release until they wanted this new feature, and they do not want to pay for a new complete development.
- Companies have neither the time nor the developers to afford the reconstruction of a system that is successful, even if its state may prevent any evolution.

3.3.1 Some War Stories

Even if anecdotal evidence does not prove anything it has the merit to be entertaining, which is somehow needed in a habilitation. Here we would like to relate some of the situations we are aware of.

A telecommunication company constructed a system (hardware and software) so that in case of emergencies like earthquakes or tornados, rescue teams could be reached by different means (telephone, pager, mobile phone...). This product was already sold to a lot of fire brigades in Switzerland. The application was composed by a user interface developed in Smalltalk and some C++ code for managing the communication. The system was not intrinsically complex. However, all the original developers left because the software has been sold to three different companies over the two last years. The code was in three different languages (french, english and german). The first page of the documentation stated that the computer had to be rebooted every three days (because of memory leaks). Even if the application was not complex, rewriting it was nearly impossible because the deadline was passed since months.

All the salaries of state employees are computed by a Cobol application. Every time a new law concerning state employees is passed, the software has to be updated without breaking the computation of previous laws. However, only one single developer knows the system and after some years of loyal services he changed company. Even if the clients are really satisfied by the application, nobody knows the exact computation rules for the salaries. There is no explicit knowledge of the computation performed by the system. Sometimes the clients call the programmer so that by reading the source code he can explain them specific rules of the state regulations. The problem is that the company selling this application would like (1) to be sure that they could continue to make money with it in case the programmer would stop working for them (SAP proposed to replace such an application for several hundred million french Francs), (2) not to have their hands tied by another company and (3) to sell their successful product to other administrations.

A successful company has 60% of the world market in surface mail sorting. The situation is the following (1) all the team members except one went away because of better salaries. (2) There is neither documentation of the system nor any comments in the code [2]. (3) The application is quite complex, because it was sold in different places of the world and distributed middleware were not available when they built the system, so all communication is home-made. This company is in trouble, because the American customers want to speed up the application by upgrading from 12 processors in parallel to 200 and they want to use PCs and not Unix systems. The code is not so bad but contains some reengineering gems such as a 5000 line method named create_Button() building the complete user interface of the system, or a parser in 3000 lines of C++ in one method. From the management point of view, the situation was not brillant. This company was asking a student to extract the design

[2]The few that I could found were obsolete. This was confirmed by a programmer before leaving the company.

as topic of a project. When the single developer left asked to hire somebody to clean code, he got as reply that there was no problem because they owned the code.

3.4 Object-Oriented Applications Reengineering

Maintenance or reengineering has been a research activity since decades for assembler or procedural programs like Fortran or Cobol. The term *legacy systems* has even been coined to refer to such systems. However, this is not because applications are developed with object-oriented languages that they do not need to be reengineered. Recently some research groups started to focus on the reengineering of OO applications. The SCG of the University of Berne participated in the FAMOOS Esprit Project (Frameworks based Approach for Mastering Object-Oriented Systems) whose goal was to study the reengineering object-oriented applications. More generally, reengineering will always be necessary, the Laws of Lehman tend to be true in any language. 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 for helping Dassault engineers to understand the application they are developing based on a component technology. The LIFIA (Laboratorio de Investigacion y Formacion en Informatica Avanzada) started a research project to support the reengineering of Web-based applications.

Robert Grass in [Gra98] quoting an empirical study led by Sasa Dekleva from 1992 states that (1) modern methods lead to more reliable software, (2) modern methods lead to less frequent software repair and (3) modern methods lead to more total maintenance time. These statements are not contradictory because modern methods make changes easier, so developed applications are changed more often.

Object-oriented programming by supporting the definition of frameworks promotes the development of complex yet customizable and evolving applications [JF88], [RJ96]. However, framework building and applications development based on them have to be constantly updated, improved and extended to provide more functionality and flexibility. They require care and redesign.

With the advent of eXtreme Programming [Bec99] or iterative development models, practices such as refactorings that were only required in the context of reengineering play now a central role. The fact that applications are continuously changed requires developers to have tools and techniques to understand, test and change the code to meet the new or next 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 relevant, so the need for approaches to understand and modify them is mandatory.

3.4.1 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 coarse-grained problems that include:

- Insufficient documentation: documentation either does not exist, or is inconsistent with reality.
- Improper layering: missing or improper layering hampers portability and adaptability.
- Lack of modularity: strong coupling between modules hampers evolution.

- Duplicated functionality: cut, paste and edit is quick and easy, but leads to maintenance nightmares.

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:

- Polymorphism and late binding make 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 their understanding more difficult. As the semantics of self is dynamic, understanding application is more difficult. This dynamicity 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, knowing which methods will be called necessitates to check if methods have been defined on the subclasses.
- Moreover, languages such as C++ with explicit pointer manipulations and complex syntax analysis difficult. For example the parser of the SNiFF+ product is not developed by Sniff developers because of “the too complex C++ syntax” sic [Bis98].
- On the one hand dynamically typed languages such as Smalltalk make the analysis of applications harder because they do not force the programmer to make types explicit and the tools have to infer them. On the other hand, statically typed languages such as C++ and Java force the programmer to explicitly cast objects thus leading to applications that are less maintainable and requiring more effort to be changed.

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 most common problems:

- Explicit Dispatch: methods are called by checking the type of the receiver explicitly instead of using late-binding.
- Misuse of inheritance: for composition, code reuse rather than polymorphism.
- Missing inheritance: duplicated code, and case statements to select behavior.
- Misplaced operations: unexploited cohesion, operations outside instead of inside classes.
- Violation of encapsulation: explicit type-casting, C++ friends...
- Class abuse: lack of cohesion, classes as namespaces.

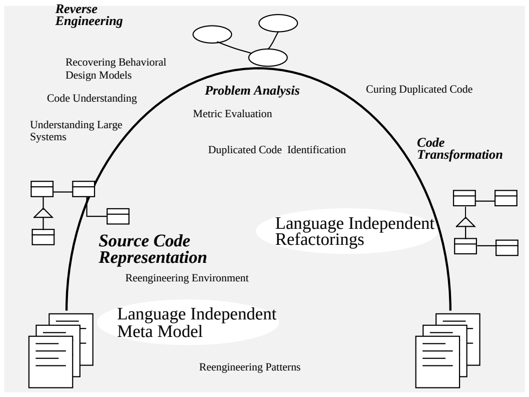
3.5 Conclusion

To summarize the context and the problems we presented, we can say that (1) reengineering is one of the key activity in software industry. (2) legacy systems exist in any programming languages and paradigm. (3) Reengineering Object-oriented applications is an important research field because legacy-systems developed using object-oriented programming languages already exist and because new developments are developed in this paradigm.

Chapter 4

A Meta Model for Reengineering

The work presented in this chapter logically belongs to the *source code representation* area that we identified in the first chapter. The issues there are to support the representation of different language for their analysis. In this chapter we explain why our need for language independence and reengineering analysis leads us to define FAMIX, the meta model we have developed to support reengineering. After briefly presenting FAMIX we show how FAMIX has been validated not only by supporting reverse engineering tasks but also refactoring analysis as shown by the following Figure.


Reverse Engineering
Recovering Behavioral Design Models
Code Understanding
Understanding Large Systems
Problem Analysis
Metric Evaluation
Duplicated Code Identification
Curing Duplicated Code
Code Transformation
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

4.1 Problem

In the context of the ESPRIT project FAMOOS [DD99b] we had the following constraints and needs:

Multiple-Languages. The case studies we have were written in Ada and C++. We decided afterwards that we wanted to be able to analysis Java and Smalltalk as well because we were in contact with legacy systems written in these languages. Moreover as Ada and C++ are not close languages the amount of work to support Java and Smalltalk was not a problem.

Size. The case studies range from 60000 lines of C++ to 2 Million lines of Ada, so the approach have to be scalable.

Parsing technologies. The parsing technology was a severe problem, because we did not want to work on writing or extending C++ parsers to deal with peculiarities of some dialects [1].

[1]Even the company developing SNiFF+ does not develop their C++ parser [Bis98].

Interexchange of Models. We wanted to be able to communicate results of analysis or model transformation with other tools developed by members of the projects.

We wanted to avoid building several analyzers, parsers for each specific language and we aimed at reusing as much as possible the representation and analysis made. In summary, we needed a source code representation that could support the analysis of code and its transformation, that would be scalable, and that we could exchange between several tools.

These considerations led us to design, FAMIX, a language independent meta model [DTD01] with language extensions [DT00], [Bae99], [Neb99], and [Tic99]. We implemented, MOOSE, a reengineering platform based on FAMIX and used SNiFF+ parsers to generate FAMIX models for all languages except Smalltalk. Such models could be loaded into MOOSE that visualizes, analyzes,... them (see Figure 5.2)

4.2 State of the Art

In the reengineering research community several models exist that represent the software itself. They are aimed at procedural languages (Bauhaus Resource Graph [CEK+00]), object-oriented/procedural hybrid languages (TA++ [Let98], Datrix [LLB+98]) and multiparadigm models [LS99]. Most models support multiple languages, either implicitly or explicitly.

Acacia implements a C++ meta-model aimed at reachability analysis and dead code detection. This meta-model represents software at the top-level declaration level (which is what we call, the program entity level). It is aimed to be complete, i.e., if an entity exists in the model then all entities and dependencies needed for compilation or execution of that entity are also included. The meta-model is C++ specific (although it has been used to analyze Java as well), hence the C++ specifics such as friend relationships, class and function templates, macros and C++ specific types. It also deals consistently with nested classes and related references. The work clearly identifies the requirements to a model to support reachability analysis and code detection [CGK98].

TA (Tuple-Attribute Language) [Hol98b] is a language to record information about certain types of graphs. It defines a simple format to describe graphs and a basic schema for describing program items such as procedures and variables and relationships such as call and reference relationships. TA++ [Let98] is an extension to TA that is aimed at providing a representation of high-level architectural information about very large software systems, in particular large legacy systems. Target languages range from Java and C++ to C, Pascal, COBOL, FORTRAN and even assembler.

Datrix is source code analysis tool developed at Bell Canada [LLB+98]. The model used to describe software is the Abstract Semantics Graph (ASG) [BC00]. An ASG is graph is representing an abstract syntax tree (AST) with additional semantic information such as identifiers' scope, variables' type, etc. The goal of the Datrix ASG model is completeness – any kind of reverse engineering analysis should be doable on an ASG without having to return to the source code – and language independence – the model should be the same for all common concepts of C++, Java and other languages.

4.3 Why FAMIX?

UML is currently embraced as "the" standard in object-oriented modeling languages [OMG99], the recent work of OMG on the Meta Object Facility (MOF) being the most noteworthy example. We welcome these standardization efforts, yet warn against the tendency to use UML as the panacea for all exchange standards or meta-models. In particular, we argue that UML is not sufficient to serve as a meta-model for reengineering applications because (1) one is forced to rely on UML's built-in extension mechanisms to adequately model the reality in source-code [DDT99b] and (2) since UML is specifically targeted towards OOAD, it lacks some concepts that are necessary in order to adequately model source-code, in particular the concept of a "*method invocation*" and an "*attribute access*". Of course it is possible to extend UML to incorporate these concepts, but then the protection of the standard is abandoned and with that the reliability necessary to achieve true interoperability between tools.

4.3.1 Mandatory Information

Because reengineering demands for a tight interaction between reverse and forward engineering, *the supporting tools need an adequate representation of source code.* At the minimum they should incorporate the core object-oriented implementation model depicted in Figure 4.1. Thus they should know about (i) classes, methods and attributes; (ii) the belongs-to relation between classes, methods and attributes; (iii) the invocation relation between methods;[2] (iv) the access relation between methods and attributes.

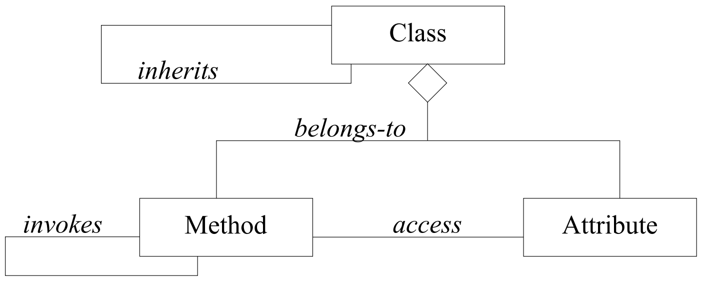


Figure 4.1: The Core Object-Oriented Implementation Model

4.3.2 Evaluating ways to extend UML

UML seems an interesting candidate for our purposes. However, we argue that in its current form it is not fit to model source adequately [DDT99b]. Basically the problem is that UML is specifically targeted towards object-oriented analysis and design (OOAD). The specification itself says [OMG99]:

> *"The UML, a visual modeling language, is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages. The UML is a language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system, but it does draw the line as you move toward code."*

The problems can be summarized as follows:

1. The UML metamodel defines a large number of concepts that are not relevant for an implementation model. "Aggregation" and "Constraint" are two examples but there are many more.

[2]Note that the (method) invocation association should take polymorphism into account. This implies that one invocation has several candidate target methods.

2. There is a substantial overlap between the requirements for a source code model representation and UML. With some flexibility it is possible to map “InheritanceDefinition” onto “Generalization” and “Class”, “Method” and “Attribute” on their respective counterparts with the same name. However, non-standard interpretation of concepts breaks the standard and its use for tools that expect the standard interpretation.

3. UML is specifically targeted towards OOAD and it lacks some concepts that are necessary in order to adequately model source-code. Especially the concept of a Invocation and Access. It is possible to extend UML to incorporate these concepts. These are the most plausable solutions (from which the last three are discussed in more detail in [DDT99b]):

   - Use the Usage dependency to model Invocation and Access. A Usage dependency represents “a relationship in which one element requires another element (or set of elements) for its full implementation or operation” [OMG99]. A Usage can be stereotyped to represent a call, which specifies that a source operation invokes a target operation. However, a UML Operation represents the specification of an operation rather than its implementation. The implementation of an operation is represented by a UML Method. The body of a method contains the invocations we want to represent and consequently we need a dependency between a Method and an Operation rather than between an Operation and an Operation to represent such an invocation. It is not clear from the UML specification if the intention of a call is the same as representing a FAMIX invocation. An additional dependency or stereotyped dependency would be needed between Methods and Attributes to represent an attribute access.

   - Use CallAction to model Invocation and Access. However, because a CallAction is defined in the context of instances, i.e., runtime entities, rather than classes a non-standard interpretation is needed for this solution to work. Moreover, to express a single method invocation we need to build a quite complicated construct (a chain of instances from the classes “Operation”, “Collaboration”, “Interaction”, “Message” and “Action”) which consumes a considerable amount of memory and is slow in processing as shown by the Figure 4.2.

   - Stereotype Association to let it represent an Invocation or Access. However, an Assocation normally associates two Classes, not two Methods or Methods and Attributes. Therefore, extra information needs to be stored in an Assocation to reference the actual contained entities involved.

   - Use the MOF, the OMG Meta Object Facility [OMG00], the metametamodel of UML to create to add new concepts to the language. However, this is a major UML extension which will not be supported by most UML-ware tools.

So to model Invocations and Accesses either minor extensions that need non-standard interpretations or major extensions that will break most tools are needed. Apart from the harder Invocation and Access, there are no straightforward ways to model entities such as GlobalVariables, Functions and implementation “details” such as LocalVariables either.

Our conclusion is that UML *in stricto sensu* is not sufficient as a tool-interoperability standard for reengineering tools.

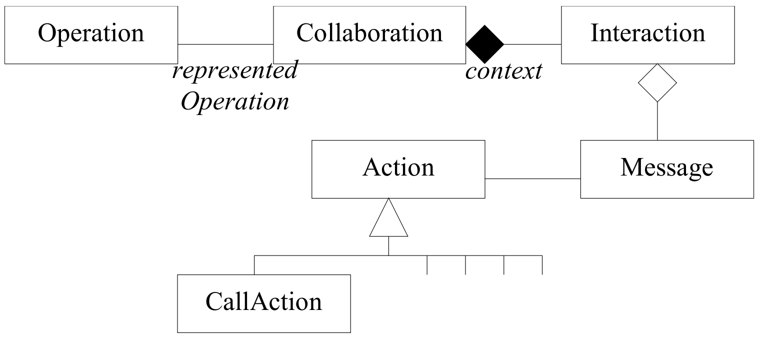


Figure 4.2: CallAction and how to navigate back to its origin.

4.4 The FAMIX Meta Model in a Nutshell

FAMIX supports multiple hybrid procedural, object-oriented languages. FAMIX started out as a model for information exchange between reengineering platforms. Careful consideration has been given how to model elements are stored and referenced. [DTD01] presents in details FAMIX and the design decision we have taken. [DT01] presents the lessons we learnt while designing FAMIX and building MOOSE, issues like the possibility to manipulate multiple models, groups of entities, incremental loading of information are discussed in detail.

FAMIX core represents all the core object-oriented elements such as classes, methods, attributes, inheritance relationships but also attribute access and method invocation (See Figure 4.3). Then language-specific plugins refine or extend this meta-model [DT00], [Bae99], [Neb99], and [Tic99].

The following information is worth mentioning in the context of the validation of the meta-model discussed in the next section.

Applications at entity level information. The model describes software systems at the so-called *program entity level* as opposed to the abstract syntax tree level. The most detailed information the model represents concerns local variables and method arguments and which method calls which method and accesses which attribute, but we do not represent the exact control flow within methods. This allows us to reason at a level which is sufficiently abstracted from language-specific details, but which is sufficiently detailed to support the analysis we need to perform.

Type Information. Type information is important information to store for languages that support it such as Java and C++. For dynamically typed languages such as Smalltalk type information can be inferred [Gar01], [Wuy01].

Multiple inheritance. Naturally this covers languages with multiple inheritance such as C++ and single inheritance in, for instance, Smalltalk. Java is covered by interpreting Java interfaces as abstract classes.

Language mappings. An important part of the usability of the model depends on how the actual programming languages are mapped to language-independent constructs. Language mappings will be discussed in further detail in section 4.5.

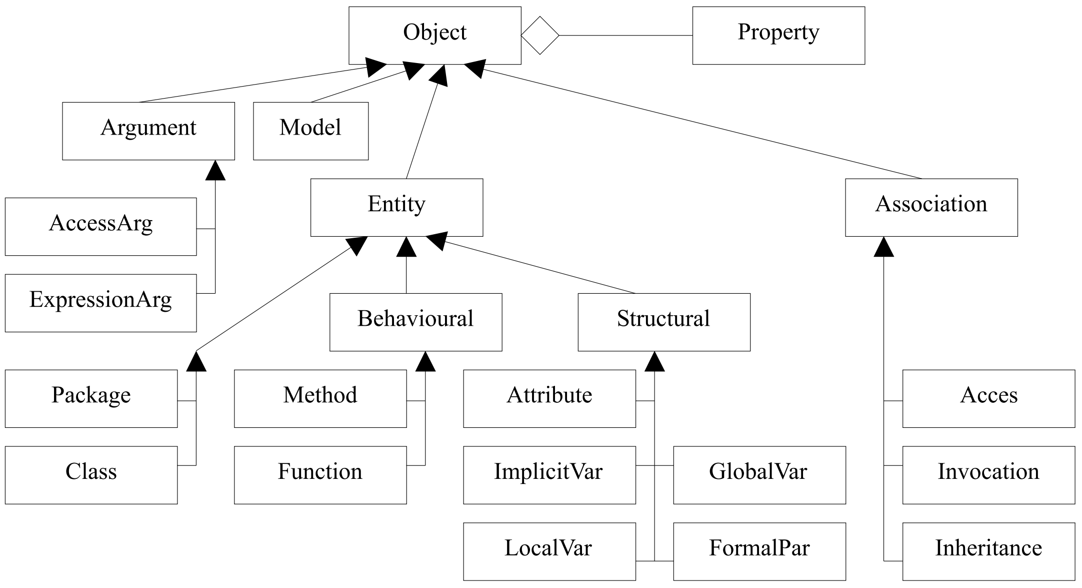


Figure 4.3: FAMIX language independent meta-model.

4.5 FAMIX **Validation: The** MOOSE **Refactoring Engine**

We validated FAMIX by been able to support (1) reverse engineering, (2) code analysis (mainly metrics) and (3) refactorings. These validations occured using, MOOSE, the environment that implements FAMIX. Here we present the expression of pre- and postconditions of common refactorings is possible in FAMIX [TDDN00] [TD01] [Tic01]. We present a short state of the art and present considerations that arise when dealing with a language independent representation.

4.5.1 State of the Art on Refactorings and Code Reorganizations.

Refactorings – behavior preserving code transformations – have become a key topic in the context of reengineering object-oriented applications [SGMZ98] [TB99a] [FBB$^+$99] [TDDN00] or new development process models such as eXtreme Programming [Bec99]. Research on refactorings originates from the seminal work of Opdyke [Opd92] in which he defined some refactorings for C++ [JO93], [OJ93]. Refactorings started to impact the way programmers develop or redesign their applications with the availability of the Refactoring Browser [RBJ97]. Today, the catalog of Fowler spreads the word to masses [FBB$^+$99].

In the context of research on refactorings, [TB99a] evaluates the impact of using a refactoring engine in C++. [Wer99] work on refactorings in the context of Java, [TB99b] in the context of C++. [FR98] reports a reengineering experience where refactoring on C++ was used and the dedicated tools was developed. [Rob99] specifies the refactorings available in the Refactoring Browser and focuses on the possibility to combine refactorings while [SGMZ98], [OCN99] work on the introduction of refactorings guided by Design Patterns applications.

Besides refactorings, research has addressed the reorganization of class hierarchies. [Cas91], [Cas92] propose algorithms for automatically reorganizing class hierarchies. These algorithms not only help in handling modifications to libraries of software components, but they also provide guidance for detecting and correcting improper class modelling. [DDHL96] proposes a new algorithm to insert class in a class hierarchy that take into account overridden and overloaded methods. [Moo96] proposes to decompose Self methods into anonymous methods and then reorganize class hierarchies by sharing as much as possible of the created methods. Note that this work while interesting from a scientific point of view could only be used to shrink applications for deployment as the symbolic meaning of method names is lost in the process.

4.5.2 Language Independent Refactorings

In [Tic01] [TDDN00] [TD01] refactoring analysis, i.e., precondition and postconditions checking, is performed for Smalltalk and Java programs based on FAMIX. [Tic01] presents the analysis based on FAMIX for all refactorings used in the Refactoring Browser except the ones required method body knowledge (Extract Method, Move To Component,...).

We validate language independent refactorings the following way: first the analysis is entirely performed in FAMIX then for Smalltalk we use the low-level code transformations of the Refactoring Browser to change the code, and for Java we currently use a text-based approach based on regular expressions. Although the text-based approach is more powerful than we initially expected, we plan to move to an abstract syntax tree based approach in the future.

When dealing with different languages, an interesting aspect is how to deal with the language differences and how design choices impact the expressive power of the resulting meta-model.

Language differences and FAMIX

The differences between Java and Smalltalk that are relevant for the language independence of the refactorings are discussed including the influence of how the mapping is done from the specific languages to FAMIX.

Java interfaces. In FAMIX, Java interfaces are modelled as classes. However, interfaces require some special rules to be observed. One cannot, for instance, pull up a non-abstract method to an interface.

Smalltalk metaclasses. FAMIX interprets Smalltalk classes and Smalltalk metaclasses as classes. Every class in Smalltalk has a metaclass associated with it. Such a metaclass does not have an explicit name and cannot be added or removed independently of the normal classes it belongs to. Another issue involves class methods and attributes. As a consequence of modelling metaclasses as classes class methods and class variables in Smalltalk are represented as instance methods and variables of the metaclass rather than as class members of the class like in Java.

Static versus dynamic typing. Due to the type-awareness of FAMIX, and the mapping to FAMIX of statically typed Java and dynamically typed Smalltalk, three phenomena can be observed:

- *Type related analysis.* In several refactorings there exists analysis for dealing with typed information. For Smalltalk much of that analysis is unnecessary. For instance, a query for all attributes with a certain type will return an empty set and this is known beforehand. This does not make the refactoring language-dependent. It just means that analysis is done that is unnecessary for Smalltalk: preconditions will not be violated and it will not result in any changes in the Smalltalk sources.

- Due to the lack of static type information in Smalltalk, invocations to a certain method name cannot be tracked to one class hierarchy with implementations.

- *Default types.* Several creational refactorings (Add Method, Add Attribute and Add Parameter) need to provide type information for Java. The solution we have chosen is to assign default types (Object for new attributes and parameters, void for method return types). Another solution would be to ask the user for a type and ignore this information in the Smalltalk case.

Roberts [Rob99] includes an extensive discussion about how dynamic analysis and dynamic refactoring could solve the lack of static type information in dynamically typed languages.

Java constructors. Java constructors are a special kind of method. Special rules apply, for example, that a constructor has to have the same name as its class, it does not have a return type, and the syntax to invoke it is different from a normal method invocation. In FAMIX Java constructors are represented as methods. However, in some cases the constructor-specific analysis needs to be performed anyway.

Global variables. Smalltalk has global variables where Java does not. All classes in Smalltalk are global variables as well, because every class is an object (an instance of the class Class). Because of this, attributes in Smalltalk cannot have the same name as a class (or any other global variable), because this might hide these globals in the scope of the new attribute. In Java types and attribute names do not interfere.

Evaluation of Language Independent Refactoring

We now present the benefits and drawbacks of having language independent refactorings.

Language independence brings useful reusability. Major parts of the refactorings are described and analyzed on a language-independent level. Similar concepts in the different languages are treated in a uniform way, resulting in reuse of analysis and reducing the language specifics to only the changes in the source code. However, in some cases the advantages of reuse come at a cost: Increased complexity of algorithms. To deal with multiple languages the underlying model needs to be general enough to cover the supported languages. For instance, the model supports multiple inheritance, which involves more complexity than would be needed, for instance, for single inheritance in Smalltalk alone.

- *Mapping back to the actual code.* The actual code changes are, naturally, language specific. Therefore, in some cases the concepts that are generalized at the language-independent level (e.g., Java constructors are methods, Java interfaces are classes) need to be mapped back to their language-specific kind, because at the code level they need to be dealt with differently than their normal counterparts.

- *Language-independent defaults.* To keep some refactorings as language independent as possible, some defaults are used. Typical examples are types: some refactorings use the most general type, i.e., Object for both Smalltalk and Java when the analysis cannot determine it. This works well for both languages, although it is clear that support for defining or changing types would be desirable for statically typed languages such as Java.

- Definitions of refactorings are tuned for compatibility over multiple languages. There is (only) one refactoring among the ones we treated, namely Push Down Method, which is defined the way it is, because of one of the implementation languages. It pushes down to all subclasses, although it could have been defined otherwise if in Smalltalk there would be enough information to push down to only one subclass.

- Not all checks are necessary for all supported languages. Depending on the implementation language some of the presented language-independent checks would not need to be carried out. An example is the analysis for attributes hiding each other, which cannot happen in Smalltalk.

Influence of meta-model design decisions. There is always a trade-off between reuse and complexity. Instead of mapping similar constructs to one representation, the two constructs can be both modelled explicitly. Naturally this decreases problems with differences between the constructs, but it also makes the model less general and opportunities for reuse could be missed. Another possibility is to not model a construct at all. This typically allows one to get rid of language specifics, but also makes the model less useful.

In the context of refactorings, the chosen mappings, most notably those of Java constructors to methods and Java interfaces and Smalltalk metaclasses, have worked out well. We especially found both Java mappings to easily fit and allow one to exploit the similarities with other constructs. For the metaclass mapping the advantages are less clear. Method and Attribute refactorings can be applied to (members of) metaclasses without any problems, but the class refactorings are not applicable at all. An alternative would be to not model metaclasses explicitly and model metaclass methods and attributes as class (in Java static) methods and attributes of the class the metaclass is representing. We have chosen not to do this, because, as said, some refactorings do work with this scheme and the alternative mapping results in problems with name clashes between class methods and instance methods and problems with the equal treatment of instance level class attributes and class level instance attributes which are different concepts in Smalltalk.

One of the modelling decisions that has worked out particularly well is the way candidate invocations are modelled. Every invocation lists the possibly invoked methods. In Smalltalk this list can be considerably larger than in Java, because no type information is available to restrict an invocation to a certain class or hierarchy of classes. The list of candidates abstracts from the differences in static and dynamic typing in the analysis of possible targets of invocations and so hides one of the main differences between Java and Smalltalk at the conceptual level of defining refactorings in terms of FAMIX.

4.6 Future Tracks

The Refactoring Browser defines the most common refactorings of object-oriented code. We would like to evaluate how the proposed refactorings can be combined to support the migration of white box components à la Envy into black box components.

In the next chapter, we list some of the ongoing work related to the meta-meta-model that we are building and which would let us express other metaa-model such as different meta-models for components [FDE+01].

4.7 Contributions

FAMIX is one of the few code representation meta-models whose intention was heavily influenced by language independence issues. FAMIX has proven to be successful and now its main ideas that drove its design are taken into account by the reengineering community effort to design a standard meta-model for program entity level in the context of the GXL exchange standard [HWS00] [Gxl01].

Based on the experience we have accumulated over years we designed a conceptual framework for designing and implementing reengineering platforms [DT01].

Additionally we clarify the semantics of certain refactorings [TD01].

Bibliography.

[DDT99b] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, pages 630-645, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[DTD01] S. Demeyer, S. Tichelaar and S. Ducasse, FAMIX 2.1 - The FAMOOS Information Exchange Model, University of Berne, 2001, Technical Report.

[DT01a] S. Ducasse and S. Tichelaar, Lessons Learned in Designing a Platform for Software Reengineering. 2001, Technical Report University of Berne.

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

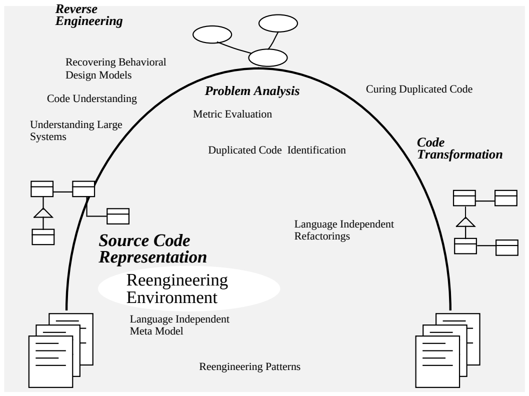
[TD01] S. Tichelaar and S. Ducasse. Pull Up/Push Down Method: an Analysis. Currently submitted to IEEE Transactions on Software Engineering.

[Tich01] S. Tichelaar. Meta-Models for Reengineering (temporary title) PhD Thesis of the University of Berne.

Chapter 5

MOOSE: A Reengineering Environment

Been able to represent, manipulate, and exchange information about code is the next logical step to support reengineering activities as shown by the following Figure.


Reverse Engineering
Recovering Behavioral Design Models
Code Understanding
Understanding Large Systems
Problem Analysis
Metric Evaluation
Duplicated Code Identification
Curing Duplicated Code
Code Transformation
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

We defined an environment based on the meta-model we presented in the previous chapter. Such an environment, called MOOSE serves as foundation for a number of experiments and tools such as CODECRAWLER (see Chapter 6 and Chapter 7) or Gaudi (see Chapter 8). In this chapter we briefly present MOOSE architecture, its main constituents. Then we present how we use a simple meta-meta-model, a description of FAMIX elements, to automatically create interexchange files representing the models. But first we start with an overview of the requirements for reengineering environments.

5.1 Requirements for a Reengineering Environment

Based on our experiences and on the requirements reported in the literature [MN97] [HEH+96] [Kaz96], these are our main requirements for a reengineering environment:

Support for reengineering tasks. An obvious requirement which determines the focus of the tool. It determines the information to store and which services the environment provides. Typical reengineering tasks are metrics computation, code analysis, visualization, grouping and refactoring.

Extensible. An environment for reverse engineering and reengineering should be extensible in many aspects:

- The meta-model should be able to represent and manipulate entities other than the ones directly extracted from the source code (e.g., measurements, annotations, associations, relationships, etc.).
- To support reengineering in the context of software evolution the environment should be able to handle several source code models simultaneously.
- It should be able to use and combine information from various sources, for instance the inclusion of tool-specific information such as run-time information, metric information, graph layout information, etc.
- The environment should be able to operate with external tools like graph drawing tools, diagram editors and parsers.

Exploratory. The exploratory nature of reverse engineering and reengineering demands that a reengineering environment does not impose rigid sequences of activities. The environment should be able to present the source code entities in many views, both textual and graphical, in little time. It should be possible to perform several types of actions on the views the tools provide such as zooming, switching between different abstraction levels, deleting entities from views, grouping entities into logical clusters, etc. The environment should as well provide a way to easily access and query the entities contained in a model. To minimize the distance between the representation of an entity and the actual entity in the source code, an environment should provide every entity with a direct linkage to its source code. A secondary requirement in this context is the possibility to maintain a history of all steps performed by the reengineer and preferably allow him to return to earlier states in the reengineering process.

Scalable. As legacy systems tend to be huge, an environment should be scalable in terms of the number of entities being represented, i.e., at any level of granularity the environment should provide meaningful information. An additional requirement in this context is the actual performance of such an environment. It should be possible to handle a legacy system of any size without long latency times.

Information Exchange and Tool Integration. A reengineering effort is typically a cooperation of a group of specialized tools [DDT99b]. Therefore, a reengineering environment needs to be able to integrate with external tools, either by exchanging information or ideally by providing a platform for tools for runtime integration.

In addition to these general requirements, the context of the FAMOOS project [DD99b] forced us to have an environment that is able to support multiple languages.

5.2 State of the Art

We cannot describe all the academic reengineering environments that exist. Even [BG97], [AT98], and [SS00] do not provide an exhaustive list of reengineering environments. We list only the ones that we know are still available and working as a starting point. Rigi [Mül86] supports reverse engineering by providing a scriptable tool with basic grouping and graph layouts. Rigi has been used a lot on the reverse engineering community, Shrimp [SBM01] provides navigation and new ways of grouping entities. Spool is an object-oriented reengineering environment that supports program understanding (hotspot, design patterns identification) [KSRP99] [RSK00]. Acacia is a tool that supports dead

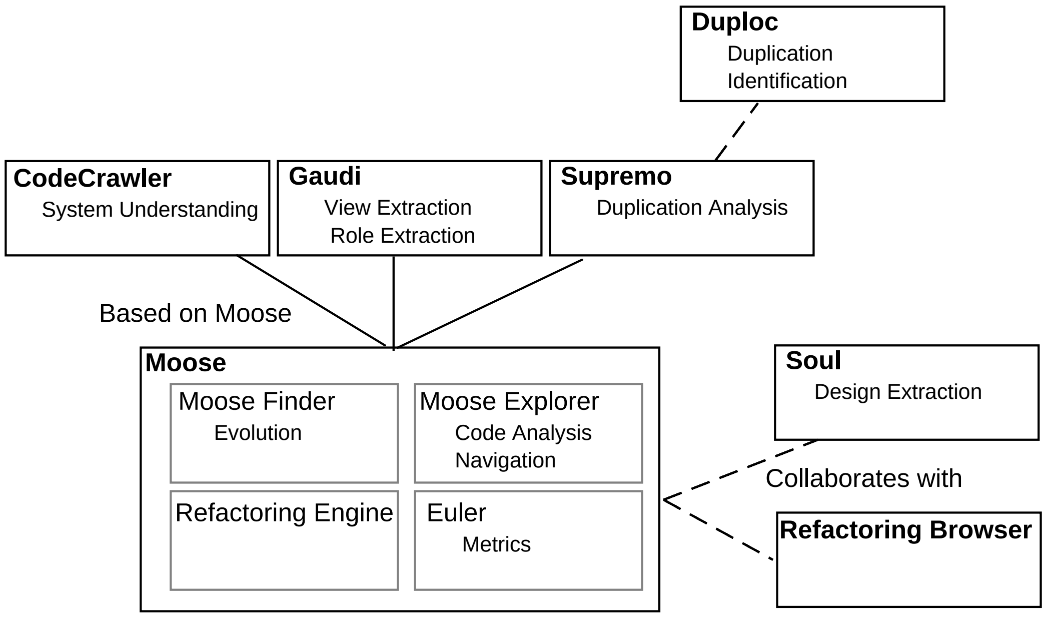


Figure 5.1: MOOSE serves as foundation for other tools.

code detection for C++ application [CGK98]. GRASP is a tool that visualizes software control structure [HIBM97]. Dali [KC99] supports the extraction of architectural element by applying database like queries to group or filter code entities. The MANSART tool [HYR96] queries abstract syntax tree (AST), and uses 'recognizers' to detect language-specific clichés associated with specific architectural styles. Multiple views of a same system can be generated and fusioned to create new ones [YHC97]. Portable Bookshelf supports architecture extraction and program visualization, it uses TA interexchange format [FHK$^+$97] [HP96]. We did not used these tools as basis for our work because we wanted to have the control of what we needed to model.

5.3 MOOSE

MOOSE is a language independent tool environment to reverse engineer and reengineer software systems. MOOSE consists of a repository to store models of software systems, and provides facilities to analyze, query and navigate them. Models consist of entities representing software artifacts such as classes, methods, etc. MOOSE serves as foundation for reengineering tools such as CODECRAWLER [DDL99] or SUPREMO [GKN01] and collaborates with other tools like SOUL [Wuy01] (Figure 5.1).

5.3.1 MOOSE architecture

MOOSE has a layered architecture (see Figure 5.2). We describe the architecture starting from the bottom.

Import/Export Framework. There are several ways to import information about software systems into MOOSE. In the case of VisualWorks Smalltalk, the language in which MOOSE is implemented, sources can be directly extracted via the meta-model of the Smalltalk language or via the

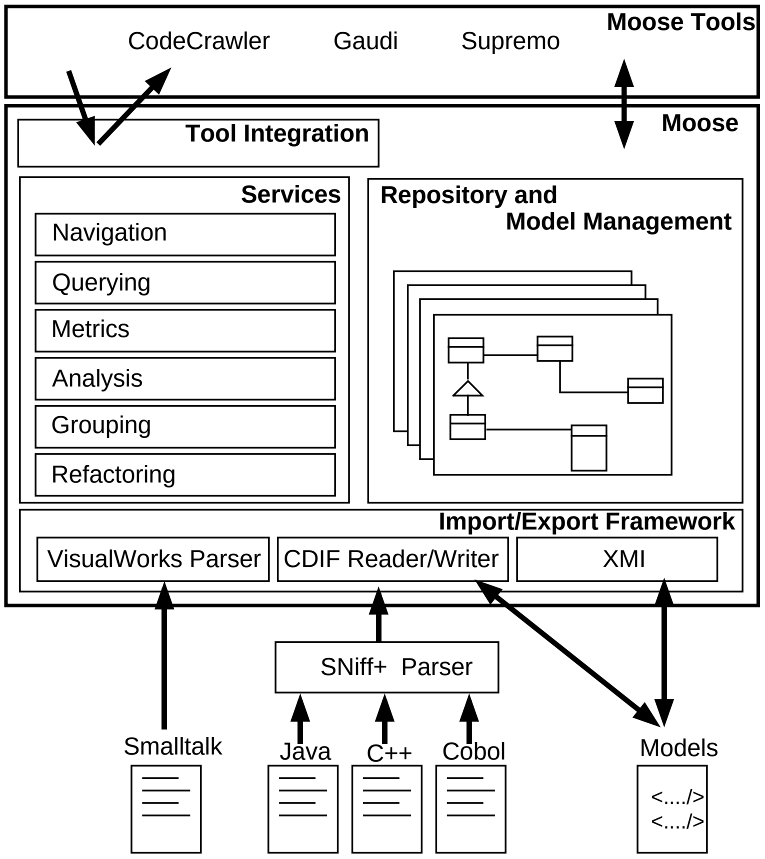


Figure 5.2: Architecture of MOOSE.

built-in parser. For other source languages MOOSE provides an import interface for CDIF or XMI (XML-based Metadata Interchange) files based on the FAMIX meta-model. CDIF [Com94] and XMI [OMG98] are industry-standard interchange formats for exchanging models via files or streams. Over this interface MOOSE uses external parsers for source languages other than VisualWorks Smalltalk. Currently C++, Java, Ada and some other Smalltalk dialects are supported. Information exchange is discussed in more detail in section 5.4.

Repository and Model Management. Information is transformed from source code into a source code model. MOOSE can maintain and access several models in memory at the same time. The models are based on the FAMIX meta-model. Every model contains elements representing the software artifacts of the target system. The information in this model can then be analyzed, manipulated and used to trigger code transformations by means of refactorings.

Services. MOOSE provides several services that make the life of a reengineer easier.

- Querying and Navigation. Every element in a model is represented by an object, which allows direct interaction of elements, and consequently an easy way to query and navigate a model.

  MOOSE FINDER is a tool that allows one to compose queries based on different criteria like entity type, properties or relationships, etc. A simple query finds entities that meet certain conditions. Such a query can in turn be combined with other queries to express more complex ones. The MOOSE FINDER supports multiple models in the context of software evolution (See figure 5.3).

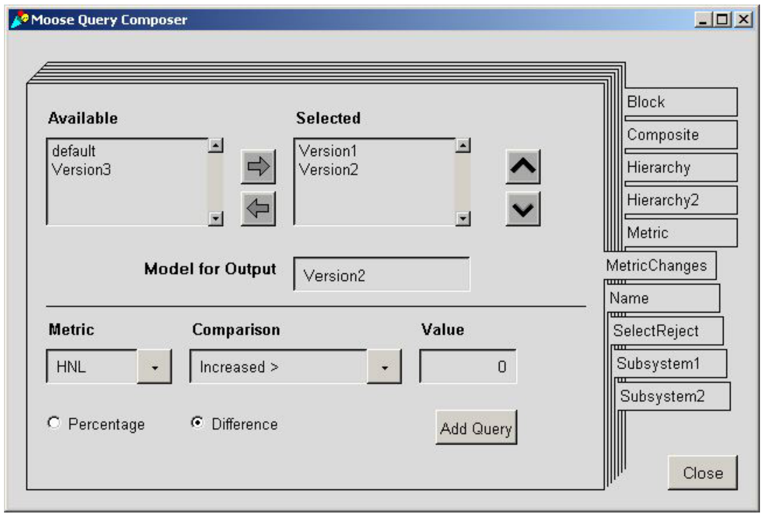
  MOOSE EXPLORER proposes a uniform way to represent model information (see figure 5.4). All entities, relationships and newly added entities can be browsed in the same way.

- Metrics and other Analysis Support. MOOSE's analysis services are mostly implemented as operators that can be run over a model to compute additional information regarding the software elements. For example, metrics can be computed and associated with the software entities, entities can be annotated with additional information such as inferred type information, analysis of the polymorphic invocations, etc.

- Grouping. MOOSE has grouping mechanisms, with which it is easy to group several entities into one group entity, which is treated from then on as an entity itself. This is useful when a reengineer wants to reduce the amount of information by looking at the subject system from a higher level of abstraction, i.e., instead of looking at several hundreds of classes, he can group them into a few dozen applications and thus obtain a better view of the whole system.

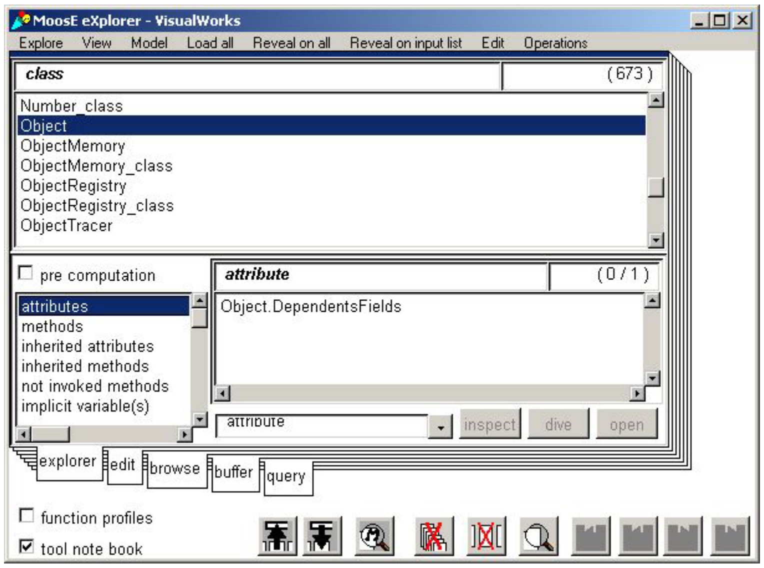
- Refactoring. The MOOSE Refactoring Engine implements language-independent refactorings as defined in [Tic01].

Tools Layer and Tools Integration Framework. The functionality which is provided by MOOSE is to be used by tools. This is represented by the top layer of Figure 5.2. Tools can use the repository and services of MOOSE and use the Tools Integration Framework to find each other and integrate.



Available: default, Version3
Selected: Version1, Version2
Model for Output: Version2
Metric: HNL
Comparison: Increased >
Value: 0
( ) Percentage [✓ Difference]

Figure 5.3: MOOSE FINDER.



class (673)
Number_class
Object
ObjectMemory
ObjectMemory_class
ObjectRegistry
ObjectRegistry_class
ObjectTracer

☐ pre computation
attributes
methods
inherited attributes
inherited methods
not invoked methods
implicit variable(s)

attribute (0 / 1)
Object.DependentsFields

attribute

☐ function profiles
☑ tool note book

Figure 5.4: MOOSE EXPLORER: From top to bottom, the first pane represents a current set of selected entities (here all the classes of the current model). The bottom left pane represents all the possible ways to access other entities from the currently selected ones. The resulting entities are displayed in the right bottom pane and can then be further browsed. Diving into the resulting entities puts them as the current selection in the top pane again, which allows for further navigation through the model.

5.4 Tools Interoperability and Meta-Meta Description

In MOOSE, interoperability between reengineering tools is supported in two ways. First, there is the possibility to stream in/out models in text files using industry standard exchange formats CDIF and XMI. Second, tools written in VisualWorks Smalltalk can interoperate with the MOOSE repository, its services and each other at runtime.

We present briefly the two formats and describe how we use a meta-meta-model, a meta-description of the FAMIX model, to support the automatic creation of interexchange files and load of models. [SDSK00] presents a discussion on the properties that an interexchange format should have and states why the Spool environment uses XMI.

5.4.1 Information Exchange with CDIF and XMI.

To exchange FAMIX-based information between different tools, MOOSE provides two textual formats. One is CDIF [Com94], an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF were that it is an industry standard and has a standard plain text encoding which is for convenient querying and human readability.

The following code shows an example of CDIF model. In [DT01] we discuss the different possibilities of formats and how they support incremental loading of entities. The format used in MOOSE is what we call a *chunk* format, i.e., the entities are not nested into their scoping entity and each entity contains information representing the relations it have with other modelled entities.

```
(Class FM16
        (name "Controller")
        (isAbstract -FALSE-)
)

(InheritanceDefinition FM17
        (subclass "Controller")
        (superclass "Object")
)

(Attribute FM18
        (name "model")
        (uniqueName "Controller.model")
        (belongsToClass "Controller")
        (accessControlQualifier "protected")
        (hasClassScope -FALSE-)
)
```

Recently, we have adopted XMI (XML Metadata Interchange [OMG98]) as a second storage and exchange format [Sch01] [Fre00]. XMI is an OMG standard for exchanging models based on the MOF (Meta-Object Facility [OMG98]) and uses XML (Extensible Markup Language [BPSM98]) as the underlying technology to save this information.

The main reason to support a second standard is that CDIF did not succeed in becoming a widely used standard. XMI seems to stand a better chance, especially because it is based on XML, which is likely to become the *de facto* standard for transferring information between applications and allows the use of XML-based technologies such as XSL. Secondly, XMI is based on the MOF, which is likely to become the *de facto* standard to describe meta-models and offers excellent integration to MOF-based meta-models such as UML. As shown in Figure 5.2 we use CDIF to import FAMIX-based information about systems written in Java, C++ and other languages. The information is produced by external parsers such as SNiFF+ [Tak96] [TD99]. Next to parsers we also have integrations with external environments such as the Nokia Reengineering Environment [DD99b].

```
<Famix.Class xmi.id="_2">
  <Famix.Entity.name>Controller</Famix.Entity.name>
</Famix.Class>

<Famix.InheritanceDefinition xmi.id="_3">
  <Famix.InheritanceDefinition.subclass>Controller
      </Famix.InheritanceDefinition.subclass>
  <Famix.InheritanceDefinition.superclass>Object
      </Famix.InheritanceDefinition.superclass>
</Famix.Class>

<Famix.Attribute xmi.id="_4">
  <Famix.Entity.name>model</Famix.Entity.name>
  <Famix.Entity.uniqueName>Controller.model</Famix.Entity.uniqueName>
  <Famix.Attribute.accessControlQualifier>protected
     </Famix.Attribute.accessControlQualifier>
  <Famix.Attribute.belongsToClass>Controller</Famix.Attribute.belongsToClass>
  <Famix.Attribute.hasClassScope xmi.value="false"/>
</Famix.Attribute>
```

The format itself is more verbose than CDIF. While normally verbosity implies human readability, in the case of XMI we feel it is “over-verbose”. Consequently, it is less “simple to process”, less “convenient for querying” and less “human readable” than CDIF. This verbosity of XML may be a problem for large systems. For example, we experimented with Swing and the same model was 8.3 Mb in CDIF versus 21.8 Mb in XMI.

5.4.2 FAMIX Meta-Description

In MOOSE, the FAMIX meta-model entities are not only implemented as Smalltalk classes but also described using a simple description mechanism. Such a description is the embryo of a meta-meta-model. Up to now, we reified the information necessary to interpret the information contained into an interexchange file and to automatically save models. Making such information explicit helps us to have a more flexible environment in which new entities can be modelled and manipulated by tools in an automatic manner.

Moreover, certain user and programming interfaces of MOOSE are automatically generated from this meta-information and every time a new meta entity is defined the tools just use this description to update themselves.

An example of a meta-description. The following description shows the description for the FAMIX class Class which in addition to the information inherited from Entity defines two attributes `isAbstract` and `interfaceSignature`.

Such a description is stored as a class method of the MSEClass which represents the FAMIX Class entity.

```
MSEClass class>>initializeClassDescription
   "self initializeClassDescription"
```

```
classDescription := superclass classDescription copy.
classDescription
   isFamixClass: true;
   famixClassName: #Class;
   addAttribute: ((MSEModelAttributeDescriptor new)
                    name: #isAbstract;
                    loadWithMethod: #isAbstract:;
                    typeCode: MSEModelAttributeDescriptor famixBoolean;
                    scope: MSEModelAttributeDescriptor instanceLevel;
                    multiplicity: MSEModelAttributeDescriptor zeroToOne;
                    isDerived: false);
   addAttribute: ((MSEModelMVAttributeDescriptor new)
                    name: #interfaceSignatures;
                    loadWithMethod: #addInterfaceSignature:;
                    typeCode: MSEModelAttributeDescriptor famixName;
                    scope: MSEModelAttributeDescriptor instanceLevel;
                    multiplicity: MSEModelAttributeDescriptor zeroToN;
                    isDerived: false)
```

5.4.3 Meta-description use: XMI

The main purpose of XMI (XML-based Metadata Interchange) is to enable easy interchange of metadata between modeling tools (based on the OMG UML) and metadata repositories (OMG MOF based) in distributed heterogeneous environments. XMI allows metadata to be interchanged as streams or files with a standard format based on XML.

An XMI document is composed by a DTD (Document Type Definition) and a set of data. The DTD describes the structure of data. To produce XMI documents, the XML-based Metadata Interchange proposal has two major components:

- The XML DTD Production Rules for producing XML Document Type Definitions (DTDS). XMI DTDs serve as syntax specification for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents. The chapter XMI DTD Production of [OMG98] describes ow an instance of a MOF can be transformed into a DTD.

- The XML Document Production Rules. It describes a set of rules that encode metadata into an XML compatible format.

As MOOSE contains already a working infrastructure for reading and saving model data based on class description, the reading/saving operations for XMI are based on this infrastructure rather than on the rules described in chapter *XML Document Production*. This decision has the drawback that the XMI reading/saving is tightly coupled to MOOSE. It is due to time concerns and will be fixed in the future.

The Figure 5.5 shows the architecture that we use to provide XML DTD Production Rules support in MOOSE [Sch01]. As the XMI/DTD Saver rely on MOF interfaces, a FAMIX entity is mapped to a MOF interface. Only the necessary behavior was implemented as shown by the Figure 5.6. Then the XMI/DTD saver uses the class description to generate the DTD and the basic MOOSE infrastructure to save data (models).

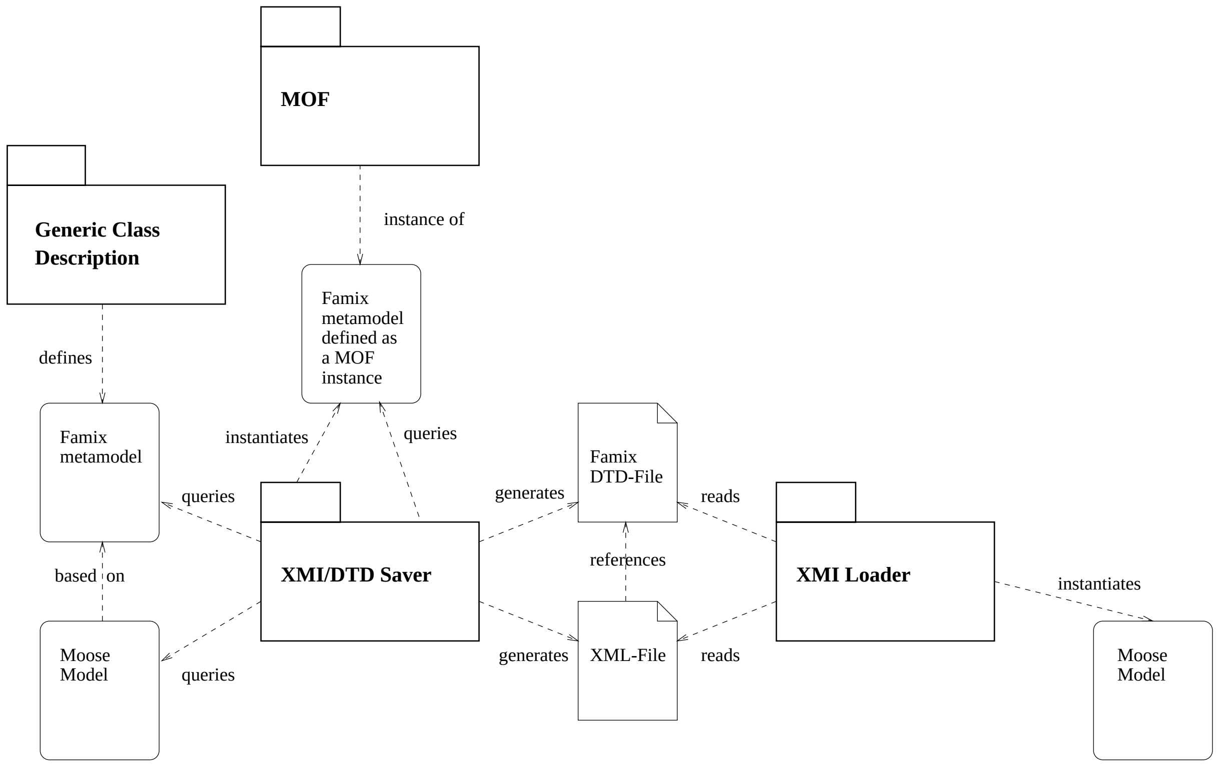


Figure 5.5: The architecture of the XMI MOOSE functionality.

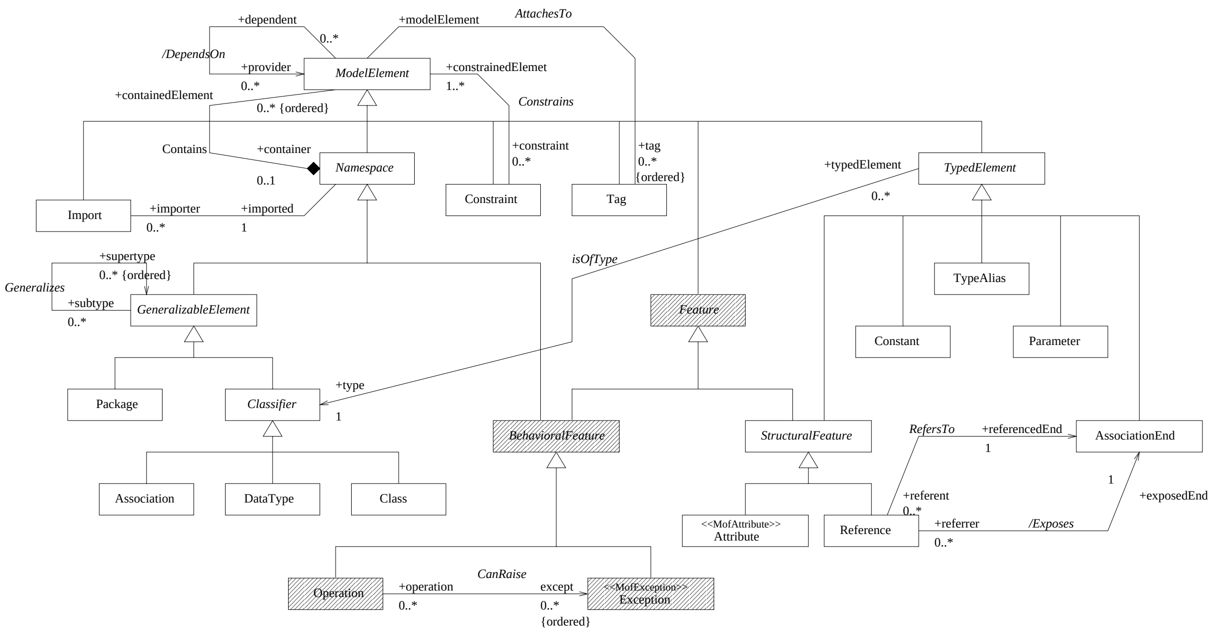


Figure 5.6: Gray boxes represents the minimal MOF implementation we made to support the XML DTD production rules mechanism.

5.5 Validation

MOOSE and its tools have been validated in several academic and industrial experiences, some of which we list in more detail below. The idea was that members of our team went to work on the industrial applications in a “let’s see what they can tell us about our system” way. There was no training of the developers with our tools. The common point about those experiences was that the subject systems were of considerable size and that there was a narrow time constraint for all experiences we describe below (See Table 5.1):

Language	**S**ize	**D**ays
C++	1.2 million LOC (2300 classes)	four days (3 persons)
C++ and Java	120,000 LOC (400 classes)	four days(3 persons)
Smalltalk	600,000 LOC (500 classes)	three days (2 persons)
Cobol	40000 LOC	3 weeks (4 persons)

Table 5.1: Industrial case studies.

The fact that all the industrial case studies where under extreme time pressure lead us to mainly get an understanding of the system and produce overviews [DDL99]. We were also able to point out potential design problems and on the smallest case study we even had the time to propose a possible redesign of the system. Taking the time constraints into account, we obtained very satisfying results. Most of the time, the (often initially sceptical) developers were surprised to learn some unknown aspects of their system. On the other hand, they typically knew already about many problems we found.

We learned that, in addition to the views provided by our tools, code browsing was needed to get a better understanding of specific parts of the applications. Combining metrics, graphical analysis and code browsing proved to be a successful approach to get the results described above.

5.5.1 Memory issues

Up to now we did not have problems regarding the number of entities we loaded into the code repository. The maximum number of entities we loaded was around 700,000 for 70 versions of an application. The workability of MOOSE then depends largely on the amount of RAM of the computer it is running on.

In the industrial context we reached 300,000 entities, which due to the small amount of RAM (128 MB) of the computer made the VisualWorks Smalltalk environment swap information to the hard disk and back. The code repository might run into problems with multi-million line projects.

For that reason we have designed the code repository to support a possible database mapping easily. In that sense the design of the code repository is more database-oriented (with a global entity manager) than object-oriented. In addition, the following considerations have to be taken into account when speaking about memory problems. First, the amount of available memory on the used computer system is, of course, an important factor. Secondly, we have never even tried to heavily optimize our environment neither in access speed nor in memory consumption, because so far we did not really have problems in these areas. Therefore, there is some room for improvement, would it be needed in the future.

A third aspect is that tools that make use of the repository need some memory of their own as well. For instance, CODECRAWLER needs to create a lot of additional objects (representing nodes and edges) for the purpose of visualization.

5.5.2 The requirements revisited

In section 5.1, we listed the main requirements for a reengineering environment. We now discuss how MOOSE evaluates in that context.

Support for reengineering tasks. MOOSE supports all major reengineering tasks, mainly because it has grown while being constantly validated in industrial contexts. Therefore, although coming from an academic background, the whole environment has strong roots in industry.

Extensible. The extensibility of MOOSE is inherent to the extensibility of its meta-model. Its design allows for extensions for language-specific features and for tool-specific information. We have already built several tools which use the functionalities offered by MOOSE.

Exploratory. MOOSE is an object-oriented framework and offers as such a great deal of possible interactions with the represented entities. We implemented several ways to handle and manipulate entities contained in a model, as we have described in the previous sections.

Scalable. The industrial case studies presented at the beginning of this section have proved that MOOSE can deal with large systems in a satisfactory way: we have been able to parse and load large systems in a short time. Since we keep all entities in memory we have fast access times to the model itself. So far we have not encountered memory problems: the largest system loaded contained more than 700,000 entities and could still be held completely in memory without any notable performance penalties.

Information Exchange and Tool Integration. The integration with several external tools has been repeatedly done without major problems. The information can be exchanged with other tool platforms using either CDIF or XMI.

5.6 Ongoing Work

Towards a meta-meta-model. Having explicit class description is powerful because such an information can be used to create self-adaptable tools. Hence, in MOOSE some interfaces, the saving and loading facilities are automatically created from the class description. Still, the class implementation have to be in sync with its class description. We are in the process of creating a more powerful meta-meta-model that tools could fully use. This meta-meta-model is basically an entity-relation one like in CDIF. For meta-models that are merely describing structural information we are able to generate all the Smalltalk classes from such a meta-description. We would like to go in the same direction as Argo, the meta-meta-model developed by Michel Tilman [Til99] which is self-described. However, we know that the difficulty lies on the specification of entity behavior and we do not want to end up with having to interpret an OCL like language and produce Smalltalk code. Our intention is to evaluate how far a meta-description can support the development of our environment. We would like to be able to express different meta-models for example such as different component models as in [FDE$^+$01] and be able to easily or automatically adapt the metrics computation, the visualization of the new entities.

5.7 Future Tracks

Mapping Code to Domain Entities. In a similar fashion that the Reflection model developed by Gail Murphy [MN97] that allows a reengineer to refine its understanding of a system by expressing how code entities map high-level of understanding, we would like to introduce in MOOSE the idea of declarative mapping that would automatically group entities.

Supporting GXL. GXL (Graph eXchange Language) is a collaborative effort from several academic and industrial research institutes to come up with an exchange format and a set of meta-models for information exchange for reengineering tools [HWS00]. We plan to support it in MOOSE.

Java Smalltalk Native Import Facilities. SNiFF+ allowed us avoiding parsing ourself some languages such as C++ or Java. But we had to live with SNiFF+ database bugs, incorrect or incomplete information provided by SNiFF+. This has the positive side effect that we had to integrate in MOOSE some protection against incoherent information that any reengineering tools has to deal with. Moreover, we have to maintain programs in C to query the SNiFF+ api which do not allow us easy changes. We would like to use FROST Java parser to extract FAMIX information. FROST is an open source environment been able to run Java 1.0 code in VisualWorks. FROST offers a Java1.3 parser in VisualWorks.

Complete Saving Facilities based on XMI. As mentioned earlier, the XMI facilities are only used for generating the DTD of a model. The model is saved, using the infrastructure used for CDIF saving. The next step is to change the MOOSE infrastructure to use the XML Document Production Rules for encoding metadata into an XML compatible format. This way the DTD and the model would be only based on meta-description.

5.8 Contributions

Implementations. MOOSE has been used as the foundation of a number of prototypes developed by the Software Composition Group members. To name the principal tools: CODECRAWLER (M. Lanza), MOOSE EXPLORER (P. Malorgio), MOOSE FINDER (L. Steiger), SUPREMO (G. Golomingi), RoseExtractor (D. Schweizer), GAUDI (T. Richner), and new prototypes under development. But MOOSE had also been used by Xavier Alvares during it PhD Thesis at the Ecole des Mines de Nantes and by the LoRE team of the University of Antwerp.

MOOSE is now used by the consultants of Deadalos AG. We are evaluating how we could continue such a collaboration.

Bibliography.

[DLT00] S. Ducasse, M. Lanza, and S. Tichelaar. MOOSE: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[DLT01] S. Ducasse, M. Lanza, and S. Tichelaar. The MOOSE Reengineering Environment. In *The Smalltalk Chronicles*, July 2001.

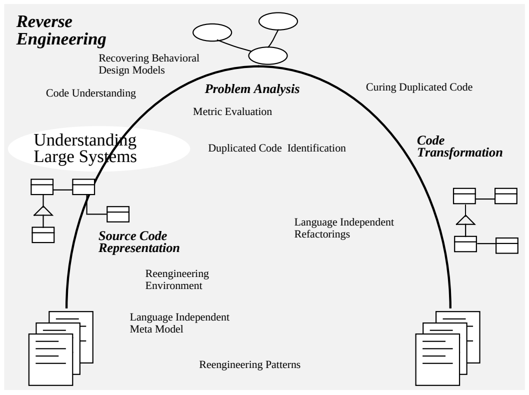
[Schl01] A. Schlapbach, Generix XMI Support for the MOOSE Reengineering Environment, University of Berne, Informatikprojekt, 2001 www.iam.unibe.ch/∼scg/Archive/Projects/schlapbach.pdf.

Chapter 6

Reverse Engineering Very Large Systems

"Continuous visual displays allow users to assimilate information rapidly and to readily identify trends and anomalies. The essential idea is that visual representations can help make understanding software easier." [BE96]

While approaching a new system, one of the first needs of a reengineer is to understand the system with a broad view. It is one of the first steps in reverse engineering.


Reverse Engineering
Recovering Behavioral Design Models
Code Understanding
Problem Analysis
Curing Duplicated Code
Metric Evaluation
Understanding Large Systems
Duplicated Code Identification
Code Transformation
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

In this chapter we first give an overview of the problems encountered when approaching large legacy systems. We briefly survey the state of the art in large scale reverse engineering, then we present our approach that is based on a simple metrics and graphs combination. Then we propose a methodology that supports our approach and propose new research axes.

6.1 Problem and Constraints

Reverse engineering can occur at different levels of granularity: from a single class, to complete system. Then the quality of the understanding can vary from a deep understanding to a superficial one.

The reverse engineering of object-oriented very large legacy systems presents a number of problems typically encountered in large-scale legacy systems:

Lack of overview. While approaching a new legacy system we lack overview of the system in terms of its size, its main components, understanding its basic overall structure. Looking at the code elements textually is like finding his way on a map using a straw.

Lack of understanding. Starting with code elements is a low level approach. Reengineers need tools to add semantics to these code elements so that they have several perspectives on the same application.

Lack of focus. Legacy systems tend to be huge. It is thus really important to focus on certain parts of a system to establish a concrete understanding.

The problem we work on is: *how can we help reengineers to approach very large legacy systems?* By *approach*, we emphasize the fact that the help we want to provide is in the early contact (e.g., the first two weeks) with the system. Moreover, reverse engineering patterns we described in [DDN02] are proposing various techniques that are used by programmers while approaching a new system. These techniques are complementary to the research we present here.

One of the major constraints that we imposed on ourselves is that our results should be applicable in an industrial context, where software reengineering faces many problems, i.e., short time constraints, little tool support, and limited manpower. It is for this reason that we limited ourselves to simple techniques. This is also why, even if the tool that validates our approach – CODECRAWLER –is successful, we do not consider it as the most important aspects of this research. For us the graphs, their simplicity and their relationships are the fundamental results [Lan99], [DL01].

Our solution in a nutshell. To help in reverse engineering large object-oriented legacy systems, we propose a hybrid approach combining the immediate appeal of visualizations with the scalability of metrics [DDL99], [DL01], [Lan99]. To follow our constraints we used *simple metrics* that are easily extracted from source code entities using for example Perl scripts and to use *simple graph layouts* that could be easily implemented using scriptable tools like Rigi [Mül86].

6.2 State of the Art

Among the various approaches that exist today, two seem very interesting for large scale reverse engineering. One is *program visualization*, often applied because good visual displays allow the human brain to study multiple aspects of complex problems in parallel (This is often phrased as ”One picture conveys a thousand words”). Another is *metrics*, because metrics are often applied to help assess the quality of a software system and because they are known to scale up well.

Program visualization. Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids. Various tools provide quite different program visualizations: Graphtrace [KG88], Rigi [Mül86], Hy+ [CMR92], [CM93], SeeSoft [BE96], ISVIS [JSB97], Jinsight [DPHKV93], [DPLVW98] [HP96].

Powerful algorithms have been developed to support such visual program representations: the Sugiyama algorithm to optimize hierarchical layouts [STT81], hyperbolic geometry to navigate through large hierarchies [LRP95], Shrimp views to optimize layouts in general [SM95], libraries providing ranges of algorithms [San96], ternary diagrams to track dynamic interactions between system modules [HMC97], mural techniques to provide large overviews [BE96], [JSB97].

All of these have in common that they seek to visualize programs by applying sophisticated layout algorithms. In some cases, these algorithms even have patents on them! In contrast, we seek to simplify layout algorithms and try to obtain added value by exploiting metrics.

Metrics. Metrics have long been studied as a way to assess the quality of large software systems [FP97] and recently metrics have been applied to object-oriented systems as well [MLM96], [Kon97], [Mar98], [LS98], [Nes88]. However, a simple measurement is not sufficient to assess such complex thing as software quality [HS96], not to mention the reliability of the results [DD99a].

Some of the metric tools visualize information via typical algorithms for statistical data, such as histograms and Kiviat diagrams. Datrix [MLM96], TAC++ [FNP98a], [FNP98b], and Crocodile [LS98] are tools that exhibit such visualization features. However, in all these approaches, the visualization is a mere side-effect of having a lot of numbers to analyze.

In our approach, the visualization is an inherent part of the approach, hence we do not visualize numbers but constructs as they occur in source code.

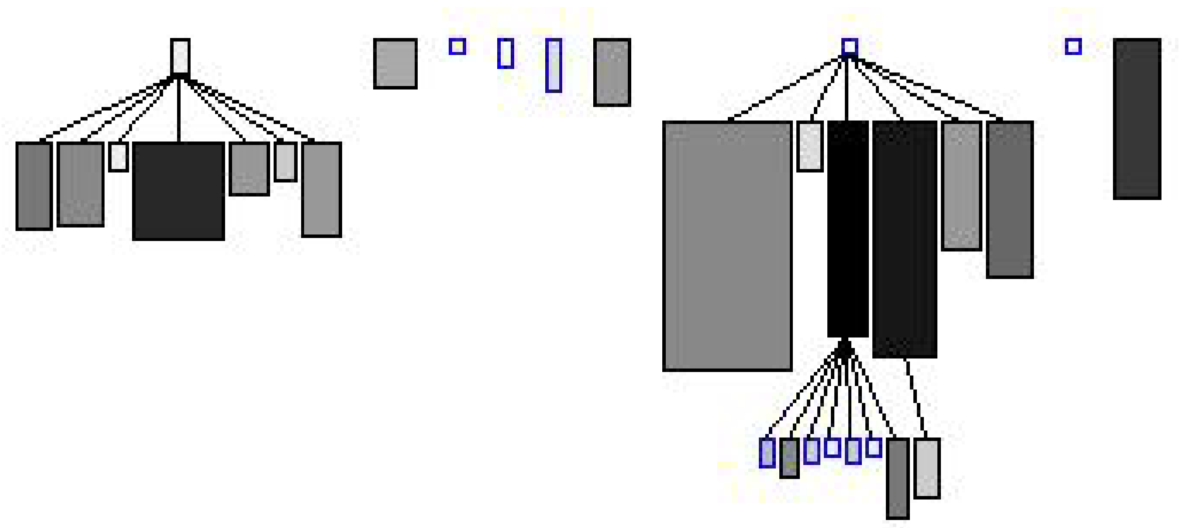


Figure 6.1: Inheritance Tree; node width = Number of Instance Variables, node height = Number of Methods and colour = Number of Class Variables.

6.3 Combining Metrics and Graphs

6.3.1 Principle.

We enrich a simple graph with metric information of the object-oriented entities it represents. Given a two-dimensional graph we can render up to five metrics on a single node simultaneously.

1. **Node Size.** The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting.

2. **Node Position.** The X and Y coordinates of the position of the node can reflect two metric measurements. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all layouts can exploit this dimension.

3. **Node Color.** The colour interval between white and black can display yet another measurement. Here the convention is that the higher the value the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.

Figure 6.1 shows an example of an inheritance tree enriched with metrics information. The nodes represent the classes, the edges represent the inheritance relationships. The size of the nodes reflects the number of instance variables (width) and the number of methods (height) of the class, while the colour tone represent the number of class variables. The position of a node does not reveal a metric as it is used to show the location in the inheritance tree.

6.3.2 Metrics

We make extensive use of object oriented software metrics and measurements [FP97], [HS96]. In the wide array of possible metrics that we could use, we selected so called *Design Metrics* [LK94], i.e., metrics that are *extracted from the source entities themselves*, because the source code is the only information we have and we can trust.

These metrics are used to assess the size and in some cases the quality and complexity of software. Moreover, the goal is to be able to easily reproduce our approach lead us to select simple metrics. In this regard our work is on how to use simple metrics to support program understanding and not on the definition of metrics for code quality assessment.
We chose to use metrics that:

- can be easily extracted from source code entities,
- have a simple and clear definition, and
- are termed as *direct measurement* metrics [FP97], i.e., their computation involves no other attributes or entities.

In Figure 6.1 we list all metrics mentioned in this article. The metrics are divided into three groups, namely class, method and attribute metrics, i.e., these are the entities that the metric measurements are assigned to.

6.3.3 Actual Visualization

The actual visualization depends on three factors:

1. **The graph type:** Its purpose is to emphasize those aspects of a system that are relevant for reverse engineering, e.g., a tree graph is well suited for showing the inheritance relationships in the system.
2. **The layout variation:** Starting from the type of the graph, layout variations further customize the actual visualization. The layout takes into account the choice of the displayed entities and their relationships plus issues like whether the complete graph should fit onto the screen, whether space should be minimized, whether nodes should be sorted, etc.
3. **The metric selection:** Metrics selected from Table 6.1 are incorporated into the graph.

6.4 Useful graphs

Enriching graphs with the set of metrics we presented leads to a huge set of potential graphs out of which only a small number are meaningful. In [Lan99] we identified a collection of *useful graphs*, i.e., a combination of a graph type and the metrics necessary to enrich the graph. Figures 6.2 and 6.3 shows two graphs applied to a large system.

Name	**Description**
	Class Metrics
HNL	Number of classes in superclass chain of class
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendants
NCV	Number of class variables
WNOS	Sum of statements in all method bodies of class
	Method Metrics
NIV	Number of instance variables
LOC	Method lines of code
NMA	Number of methods added, i.e., defined in subclass and not in superclass
MHNL	Class HNL in which method is implemented
NME	Number of methods extended, i.e., redefined in subclass by invoking the same method on a superclass
MSG	Number of method message sends
NOP	Number of parameters
NMAA	Number of accesses on attributes
NMO	Number of methods overridden, i.e., redefined compared to superclass
NOS	Number of statements in method body
NOC	Number of immediate children of a class
	Attribute Metrics
NOM	Count all methods in class
AHNL	Class HNL in which attribute is defined
WLOC	Class lines of code
NAA	Number of times accessed

Table 6.1: Selected Measurements and Metrics.

We present a selected list of useful graphs in tables 6.3 and 6.2. The first two entries are the name and the layout of the useful graph. The third entry describes its metrics as follows: first the node size, then the node color and the node position: width node metric, height node metric, node color metric, x node metric, y node metric. '-' means that no metric is defined. Then the fourth entry describes if the graph is sorted, and if so according to which metric. The last entry describes the scope of graph.

The figures 6.2 and 6.3 show two useful graphs applied to Squeak [Squ01]. Figures 6.2 presents a variation of System Hot Spots where the size of the node reflects the number of methods. The shaded nodes are Smalltalk metaclasses. Figure 6.3 presents Hierarchy Carriers on the Morph hierarchy in Squeak where the node width metric reflects the total number of children, the node height metric the number of methods and the color the total number of children. Such a graph helps to identify classes that have an impact on the inheritance graph (See [Lan99] for a detailled discussion).

6.5 Experimental Validation

We validated our approach during several experiments performed on applications coming from industry, academic or open source mouvement. The following table 6.4 describes the case studies. Note that for the applications written in Smalltalk the metaclasses are not counted.

Name	Layout	Metrics	Sort	Scope
Graphs applied to methods				
Identifies possible candidate methods for further refactorings.				
Coding Impact Histogram	Histogram	LOC, -, LOC, LOC, -	width	Small Subsystem or single class
Gives an overview of the class method distribution.				
Method Size Nest Level	Checker	LOC, NOS, MHNL, -, -	NO	Subsystem: Inheritance Tree
Identifies big method deep into hierarchy.				
Graphs applied to attributes				
Direct Access Attribute Checker	Checker	NAA,NAA,NAA,-,-	NO	Full System
Categorizes attributes in terms of their uses.				
Graphs applied to class internals				
Class Confrontation	Confrontation	method(LOC,NOS,LOC) attribute (NAA,NAA,NAA)	NO	Class
Presents class structure in terms of attribute accesses and identifies method clusters.				
Method Size Correlation	Correlation	-, -, LOC, NOS, LOC	NO	Full System
Gives an overview of the system from a method size point of view. It detects empty, strange, long or non-coherently formatted methods.				

Table 6.2: List of useful graphs applicable on methods and attributes.

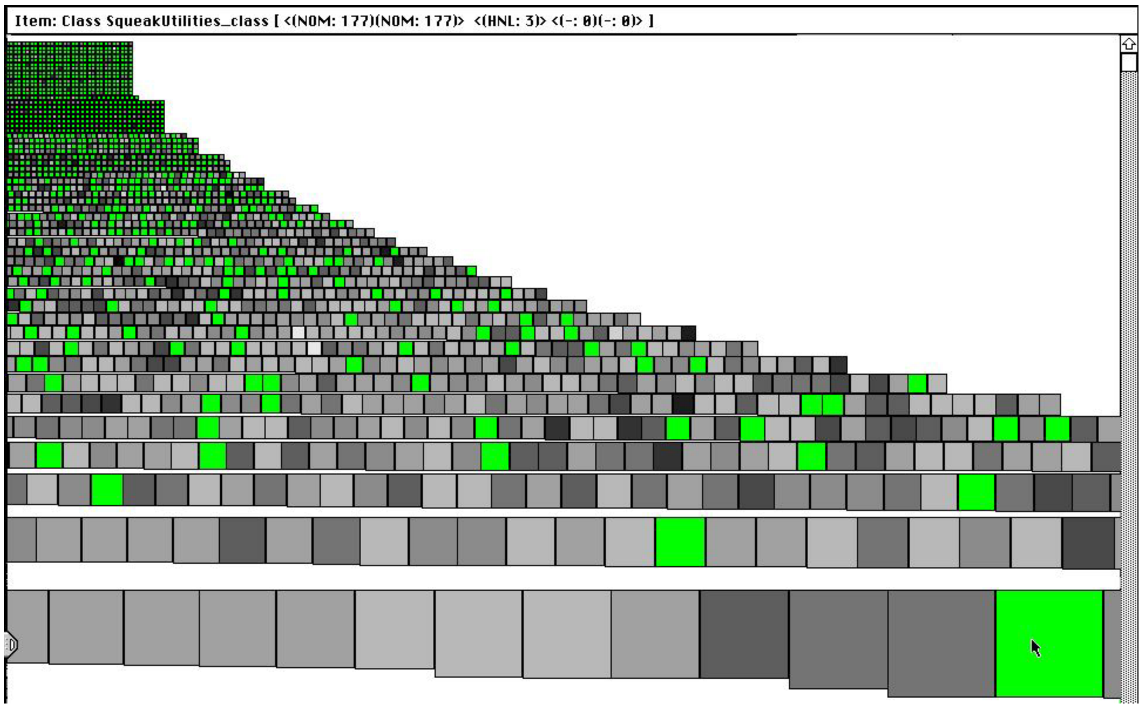

Item: Class SqueakUtilities_class [<(NOM: 177)(NOM: 177)> <(HNL: 3)> <(-: 0)(-: 0)>]

Figure 6.2: A variation of System Hot Spots where the size of the node reflects the number of methods. The two big boxes represents the metaclasses of Preference and Utilities.

6.6 A Methodology

We applied our approach to several large industrial applications ranging from a system of approximately 1.2 Million lines of C++ to a Smalltalk framework of approximately 2100 classes. Our

Name	Layout	Metrics	Sort	Scope
System Complexity	Inheritance Tree	NIV, NOM, LOC,- ,-	NO	Full System
Gives an overview based on the inheritance hierarchies of a whole system. It gives clues on the complexity and structure of the system.				
System Hot Spots	Checker	NOM, NIV, WLOC,- ,-	NO	Very Large System
Displays really simply all the classes according to their number of methods and instance variables.				
Weight Distribution	Histogram	NOM, -, HNL, - ,NOM	NO	Full System
Categorises systems as top-heavy, bottom-heavy or mixed.				
Root Class Detection	Correlation	-, -, -, WNOC, NOC	NO	Very Large Full System
Identifies important classes regarding their impact on their children.				
Service Class Detection	Stapled	NOM, WLOC, NOM,- ,-	width	Full System
Identifies data structure like class containing only read write accessors.				
Cohesion Overview Checker	Checker	NOM, WNAA, NIV,- ,-	width	Full System
Identifies possible god class candidates and highly cohesive classes.				
Hierarchy Carriers	Inheritance tree	WNOC, NOM, WNOC ,- ,-	NO	Subsystem: Inheritance Tree
Identifies classes having a big impact on their subclasses.				
Intermediate Abstract	Inheritance tree	NOM, NMA, NOC,- ,-	NO	Subsystem: Inheritance Tree
Detects abstract classes or nearly-empty classes which are located somewhere in the middle of an inheritance chain. Ideal for Smalltalk.				
Class Size Checker	Checker	LOC, LOC,NIV,-,-	width	Full System
Gives a raw overview of the system in terms of its physical size				
Inheritance Classification	Inheritance Tree	NMA,NMO, NME,-, -	NO	Subsystem
Qualifies inheritance relationships. Subclasses can be only overriding methods, adding functionality or specializing behavior.				

Table 6.3: List of useful graphs which can be applied on a collection of classes or on a complete system.

Name/Language	Approximate size
Academic Case Studies	
Refactoring Browser 3.03 (Smalltalk)	over 150 classes
Duploc 2.0 (Smalltalk)	over 220 classes
Industrial Case Studies	
VisualWorks 3.0 (Smalltalk)	over 700 classes
C++	1.2 MLOC, over 2300 classes
C++/Java	120 kLOC, over 400 classes
Smalltalk	600 kLOC, over 2100 classes
COBOL	40 kLOC
Open Source Case Studies	
Squeak 3.0	over 1500 classes

Table 6.4: Case studies on which we applied CODECRAWLER.

t

experiments demonstrated the strength of our approach. We were able to quickly gain an overall understanding of the analyzed application, identify some problems and point to classes or subsystems

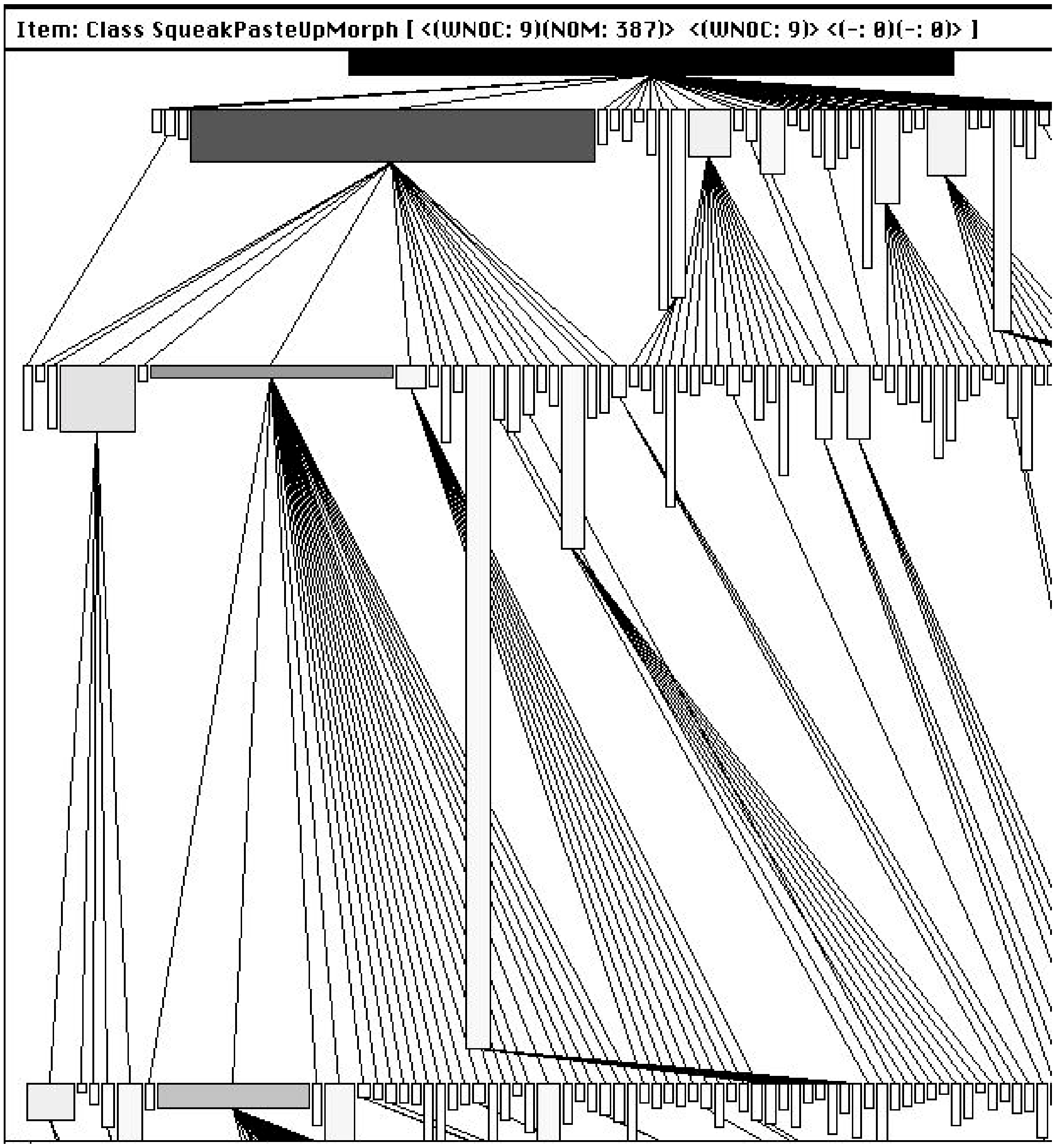

Item: Class SqueakPasteUpMorph [<(WNOC: 9)(NOM: 387)> <(WNOC: 9)> <(-: 0)(-: 0)>]

Figure 6.3: Hierarchy Carriers on the Morph hierarchy in Squeak where the node width metric reflects the total number of children, the node height metric the number of methods and the color the total number of children. The first top node is the class Morph, the second black node is the class BorderedMorph.

for further investigation. Moreover we learned that the approach gives better results during the first contact with the system, and provides maximum benefits during the one or two first weeks of the reverse engineering phase. However, the approach definitively lacked a methodology that would help a reverse engineer to deploy its full potential [DLT00]. Ideally such a methodology should define which graphs to apply depending on the goals of the reverse engineer, what the paths are between the different graphs, and on what selections the next graphs should be applied.

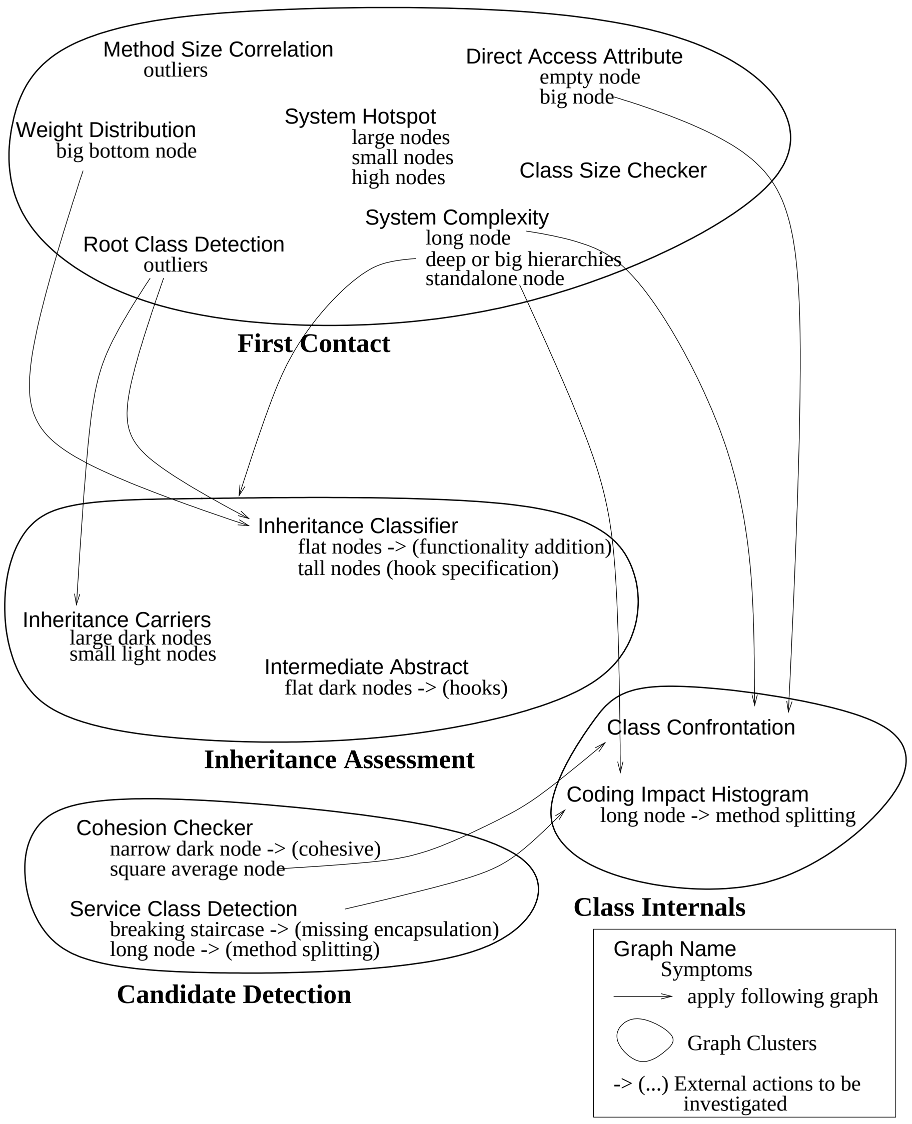


Figure 6.4: The methodology is summarized by a navigation map that identifies the graphs and their symptoms and relates the clusters themselves and the graphs.

Difficulties. Such a methodology is difficult to elaborate for the following reasons:

- There is no one unique or ideal path through the graphs.
- Different graphs can be applied at the same stage depending of the analysis of the previous graphs.
- The decision to apply a graph most of the time depends on some interactions with the current graph.
- The graphs can be applied to different entities implying some back and forth between different graphs.

- A graph displays a system from a certain point of view that emphasizes a particular aspect of the system. However, the graph has to be analyzed and the code understood to determine if the symptoms revealed by the graph are interesting for further investigation.

6.6.1 The Methodology Navigation Map

Each of the graphs presented here exhibits some *symptoms*, like small dark nodes or wide flat nodes. Such symptoms provide information about the analyzed system and also support the choice of the next graphs to further improve this understanding. Graph symptoms point to possible paths that guide the reverse engineer from graph to graph.

Depending on the symptoms the next graph can be applied on the same entities, on a subpart of the currently displayed entities or on the structurally containing entities (a class for a method or an attribute). Not all the symptoms are leading to new graphs but also to some specific reengineering actions that represent the next logical step after the detection of defects. For example detecting a "god class" [1] [Rie96] may lead to a split of the class, long methods may be analyzed to see if they contain code duplication or be split up if they perform several tasks at the same time [RBJ97], [FBB$^+$99].

The presented methodology is based on clusters that group the useful graphs depending on the problem encountered by the reverse engineer and the information provided by the graphs. Each of these clusters is presented in detail in the subsequent section. We identify four clusters:

1. FIRST CONTACT, which provides different overviews of the system.
2. INHERITANCE ASSESSMENT, which qualifies the inheritance relationships and the role played by the classes.
3. CANDIDATE DETECTION, which identifies potential class candidates for future investigation.
4. CLASS INTERNALS, which analyses the classes themselves.

Figure 6.4 presents a map that summarizes the main paths between the different the graphs, the clustered graphs and their symptoms. [DL01] elaborates the methodology.

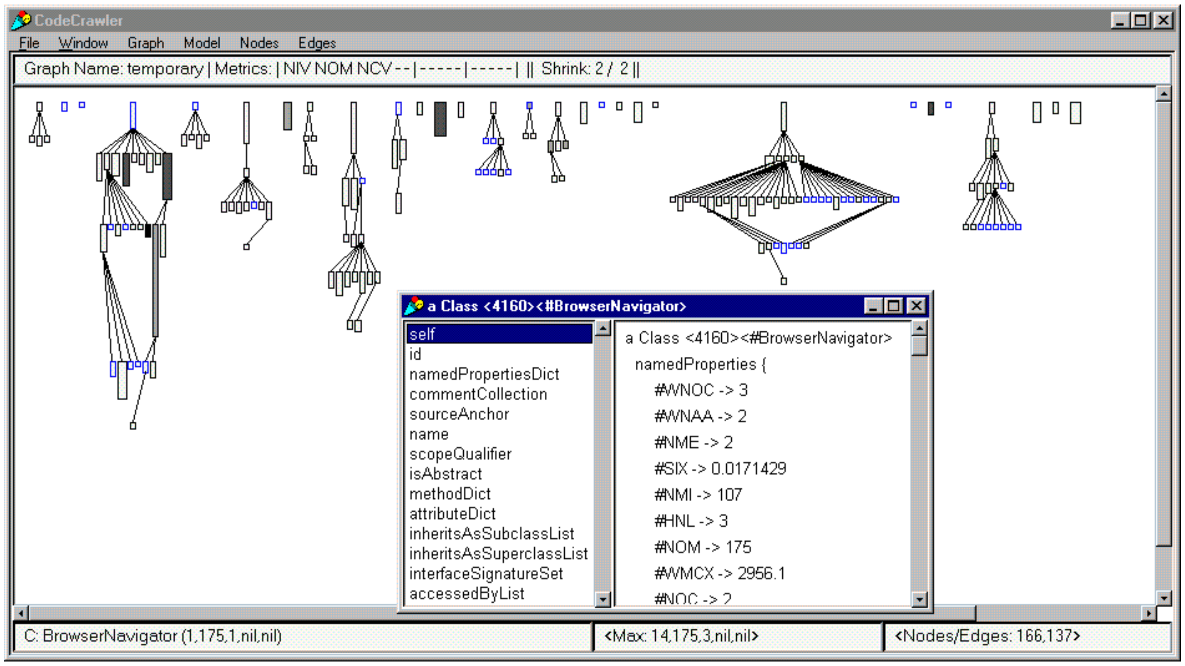
6.7 Tool support: CODECRAWLER

During his Master's thesis [Lan99], Michele Lanza developed an exploratory tool, named CODECRAWLER based on the MOOSE environment (see Chapter 5). CODECRAWLER is an open platform providing a graphical representation of source code combined with object oriented metrics. The figure 6.5 shows a screen dump of the tool in action.

Practical considerations. Besides testing the combinations of graphs with metrics values, we have been confronted with practical considerations like the minimal size of a node or the size of the screen. These considerations have influenced the graph definitions, hence we report them here.

For the node size, we chose to implement the mapping such as to accurately reflect the measurement in the size on the screen with a slight distortion in case the measurement drops below a certain threshold. A minimal node size is a purely practical issue that is necessary when we want the graph to be interactive, since clicking with the mouse pointer on nodes only one or two pixels wide is difficult.

[1] Riel defines a god class as a class that has grown over the years and has been assigned too many responsibilities.



Graph Name: temporary | Metrics: | NIV NOM NCV - - | - - - - - | - - - - - | || Shrink: 2 / 2 ||

```
self
id
namedPropertiesDict
commentCollection
sourceAnchor
name
scopeQualifier
isAbstract
methodDict
attributeDict
inheritsAsSubclassList
inheritsAsSuperclassList
interfaceSignatureSet
accessedByList
```

```
a Class <4160><#BrowserNavigator>
  namedProperties {
    #WNOC -> 3
    #WNAA -> 2
    #NME -> 2
    #SIX -> 0.0171429
    #NMI -> 107
    #HNL -> 3
    #NOM -> 175
    #WMCX -> 2956.1
    #NOC -> 2
```

C: BrowserNavigator (1,175,1,nil,nil) | <Max: 14,175,3,nil,nil> | <Nodes/Edges: 166,137>

Figure 6.5: The CODECRAWLER platform at work: (1) an inheritance tree with x node size = NIV, y node size = NOM, colour = NCV and (2) inspecting the data of a represented entity.

For the node colour, we chose to avoid optical overload by limiting the use of colors and using gray tones. Of course, the usage of different colors is a good way to attract the attention of the eye, but too many of them results in overload. The solution with gray tones has the advantage that numerical information can be transferred by gray values: we map numerical values (e.g., the metric measurements) into a colour interval ranging from white to black. Although this is a good way to display a supplemental metric, we experienced that the perception of a gray tone is less precise than the perception of size. Thus, the gray tone is only useful for the detection of extreme values.

Note that CODECRAWLER supports different distributions (e.g., linear, logarithmic) represent the size of the nodes plus different modes like the shrinking of the graphs to fit the graphs into the size of a screen. It is also able to mark nodes whose metrics exceeds a certain threshold value.

Supporting reverse engineering. CODECRAWLER provides a number of features that greatly enhance reverse engineering activities. First of all, a reverse engineer may configure and save sets of graph parameters. Next, CODECRAWLER shows all the information of the current displayed graph (top border) and the information related to the entity currently selected (bottom border). In Figure 6.5 the metrics are NIV, NOM and NCV applied on class entities, the last investigated class is BrowserNavigator that has 1 instance variable, 175 methods and 1 class variable.

Once the graph is displayed, several operations are possible. These include highlighting all the edges arriving at a specific node, following a particular edge, applying a new graph to a specific node. It is also possible to query the graph to locate nodes via their name. Finally, each graph entity is linked to the code entity that it represents, so the reverse engineer can browse the code related to the displayed entity as well as inspect its metrics.

Implementation. CODECRAWLER is developed within the VisualWorks Smalltalk environment, relying on the HotDraw framework [Joh92] for its visualization. It uses the facilities provided by the

VisualWorks environment for the Smalltalk code parsing. For other languages like C++ and Java it relies on Sniff+ to generate code representation encoded using the FAMIX Model [TD98] (see below).

6.8 Ongoing Work

The approach presented here is complementary to the research made on fine grained code understanding presented in Chapter 7. We currently working on the following extensions of this work:

6.8.1 Collaboration and Grouping Entities

In the current approach the only relation we considered was the inheritance relation. Now we are investigating other relations like aggregation, containment or reference. The idea is to define a first set of other simple graphs that based on containment or reference can help us to understand an application. To ease this work in Smalltalk, we use different type inferencers: a type snooper [2], a prolog based type inferencer based on SOUL [Wuy01] and we are evaluating the possibility of using a dynamic type collector [RBFDD98].

Up until now we limited ourselves to display and analyze basic code level entities such as classes, methods and attributes. We are starting a new research on how grouping such entities together can help understanding. This axis is somehow related to work done in architectural extraction described in Chapter 13. However, here we are focusing on code related information. For example, the idea is to group all classes belonging to a class category or envy-application and the group size represents certain property of its constituents.

6.8.2 Navigating Models

The approach we presented while powerful, displays graphs one after the other ones. What is missing is support for linking these graphs in a navigation that suits the reengineer. Tools such as the TheBrain that are based on config maps [Top] support simple navigation facilities in knowledge based graphs. In the context of reengineering, the navigation is much more complex because of the density of the graph and the numbers of relations. We are investigating how to support the navigation in complex models: we feel that specific neighbor visibility or transitive walking and selection will be necessary.

6.9 Future Tracks

The approach we presented has proven to be useful for getting an overview of a large system. It is successful when applied in the early phase of a reengineering effort. It can also be used during forward development to provide different views of a system to its developer and then providing feedback on the progression of the work. One of the limits of the approach is that condensing information is good but after a while we need to understand the functionality of an application. We started to work on this area and support program understanding the granularity of the classes by proposing *class blueprint* as presented in the Chapter 7 [LD01]. This approach is complementary to the approach presented in Chapter 8 that allows the developer to iteratively [Mur96] query static and dynamic information of the system to support its understanding.

[2]A type snooper uses the reflective facilities of Smalltalk. It queries a certain number of instances of a class and looks at the type of the instance attributes.

From the experience we gained the following topics represent possible future research tracks:

Added Visual Information. We would like to investigate how more information about the source code entity could be represented as node in CODECRAWLER graph. We are thinking to add floating halos à la Squeak that would convey more information about the nodes.

Graphical Layouts for Reengineering. We implemented a spring layout which opens a full range of new explorations. However, from our current experiments we see that the simplicity of the approach can be lost and that identifying useful graphs based on spring layouts is difficult when the graph is dense. We would like to identify the situation and specific graphs using a spring layout that would provide useful information. The graph layout literature is extremely rich and we would like to know which graph layout would provide insightfull views on reengineered software.

Software Architecture and Design Extraction. To support system understanding, it is also necessary to reason in terms of domain entities or architecture. The work we started on grouping entities is a step in that direction. In addition, being able to define domain entities and mappings from the code entities to the domain entities is needed [MN97].

6.10 Contributions

Implementations. CODECRAWLER is a tool that implements the ideas discussed and supports the reverse engineering of large systems. [Lan99] describes an early version of the tool.

Bibliography.

[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, *Procedings of WCRE'99 (6th Working Conference on Reverse Engineering)*, pages 175–187. IEEE Computer Society Press, Oct. 1999.

[DL01] S. Ducasse and M. Lanza. Towards a reverse engineering methodology for object-oriented systems. *Technique et Science Informatique*, 2001, Vol 20, N 4, pages 539–566, in Technique et Science Informatique, Edition Speciale Réutilisation.

[Lan99] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse* Master's thesis, University of Berne, 1999.

[RBFDD98] P. Rapicault, S. Ducasse, M. Blay-Fornarino and A.-M. Dery, Dynamic Type Inference to Support Object-Oriented Reengineering in Smalltalk, In Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543, 1998.

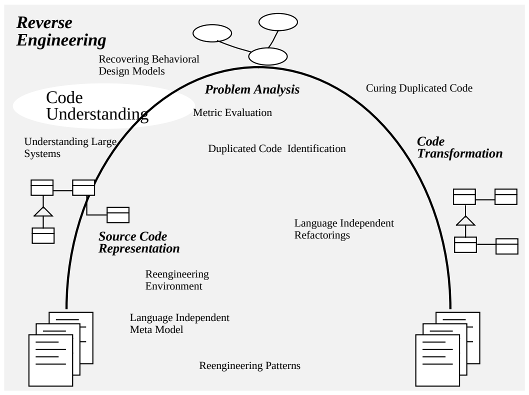
Chapter 7

Supporting the Understanding of Fine Grained Code Elements

"Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one must know both the original design rules, and the newly introduced exceptions to the rules, to understand the product. After many such changes, the original designers no longer understand the product. Those who made the changes, never did. In other words, nobody understands the modified product."[Par94]

Program understanding plays an important role during maintenance of applications. Already in the late seventies it has been reported that about 47% of the maintenance efforts are spent on program understanding whereas only 25% are spent on modification [FH79]. Following other studies, the program understanding effort is said to cover 40% [Hen97], 60% [Sel90] or even up to 90% [Sta84].

Understanding classes is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built. The main problem of this task is to quickly grasp the purpose of a class and its inner structure. It belongs to the reverse engineering area as shown by the following Figure.


Reverse Engineering
Recovering Behavioral Design Models
Code Understanding
Problem Analysis
Curing Duplicated Code
Metric Evaluation
Understanding Large Systems
Duplicated Code Identification
Code Transformation
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

In this chapter we present a first approach we elaborate to help reengineers in their work: class blueprints. Such an approach is complementary to the one we propose in Chapter 8 where a reengineer can iteratively refine its understanding by querying and visualizing static and dynamic information of a system. The class blueprint is a simple visualization technique that allows one to characterize classes.

We briefly present the problem of code understanding and a state of the art in the context of program understanding. Then we present the class blueprints and discuss further research tracks.

7.1 Problem: Code is Just Code

In object-oriented programming, understanding a certain class can be the key to a wider understanding of the system the class is contained in. However, the basic approach to class understanding has basically not changed during the past decades(!). The benefit of high level languages is that source code can be read like a text written in English. Thus, the names the developers use for the classes, methods and attributes can already convey a substantial understanding without requiring an in-depth analysis of the source code. Apart from the obvious difficulties which stem from the use of acronyms and domain specific terminology, it is the use of inheritance in object-oriented software which can make code hard-to-read: Inheritance represents a form of *incremental definition* of classes [WH92]. To fully understand a class one must therefore understand its super- and subclasses as well. Another problem is represented by the dynamicity of *self* calls, whose meaning can completely change if a superclass is changed or a new superclass is inserted in the inheritance hierarchy.

In the first contact with a foreign software system there is a need for a quick, intuitive and thorough understanding of the classes. Note that to *understand* a class you do not need to read every line of its code and you do not need to understand every piece of functionality contained therein.

7.2 State of the Art

Reengineering and Visualization Tools. Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids. Various tools, using dynamic (trace-based) information such as Program Explorer [LN95a], Jinsight and its ancestors [DPHKV93, DPS99], Graphtrace [KG88], or [KC98] and static information, such as Rigi [Mül86], Hy+ [CM93], SeeSoft [ESJ92], TANGO [Sta90] and ShrimpViews [SBM01] [SM95] [SMW96], have been developed to visualize software.

Researchers in the program understanding community focus on the one hand on providing high-level views by grouping entities [Mül86], [KC98], [SM95], [CHRY96], [TFAM96], [FTAM96], [Hol98a]. On the other hand researchers have been working at the construct level. For example, [Bal97] presents a conceptual framework for the definition of outlines of program. Outlines contract the amount of information required to understand a program. The work focuses primarily on understanding loops. [RS97] uses an abstract language to represent program cliches and propose different matching algorithms to identify cliche in the code. [CRW98] identifies units of knowledge required to understand a program. [HIBM97] presents GRASP, a tool to visualize software control structure. Spool supports program comprehension by supporting design information navigation [RSK00].

IDE support. When it came out in early 80's, the source code browser of Smalltalk was simply revolutionary in comparison to the other contemporary language environments which were essentially reduced to a command line. Nowadays good IDEs propose the same kind of functionality for browsing the code. Smalltalk-like source code browsers propose multiple functionality like finding all the methods accessing in read only a given instance variable, or all the senders of a given message. Some of the most advanced IDEs present in blue some constructs like abstract and small icons to distinguish other predefined characteristics. However, generally speaking there is simply no inferred information such as knowing what are the instance variables representing the core of a class or knowing which methods represent a hook [SRMK99] [Dem98].

NeoClasstalk browsers made a step to present more information such as methods inherited, overridden and locally defined in the same class [Riv96a]. The Selector Browser of Squeak proposes the same information. The Classification Browser [DeH98] made a first attempt to categorize instance variables or methods as participating to the core of a class. However, there was only one heuristic to categorize the entities and depending of the coding conventions it was not useful. Whisker is a code browser for Squeak that supports the simultaneous browsing of multiple methods defined in possibly different classes[Way01].

However, most publications and tools that address the problem of large-scale static software visualization, mainly treat classes as the smallest unit in their visualizations, while we provide a visualization of the internal structure of the classes. In this sense our approach proposes a new dimension in the understanding of systems.

7.3 Class BluePrints

To help the reverse engineers in their first contact with a foreign system, we propose a categorization of classes based on the visualization of their internal structure. The contributions of this work are a novel categorization of classes and a visualization of the classes which we call the *class blueprint*.

7.3.1 Class Blueprint Concepts

Class blueprints are based on layers. Figure 7.1 presents a template class blueprint. The rightmost layer is the attribute layer. The second from the right is the layer of the accessor methods. The other layers are placed according to the method invocation sequence from left to right, i.e., a layer and the methods contained therein are invoked at a later point in time than the layers and methods placed to their left.

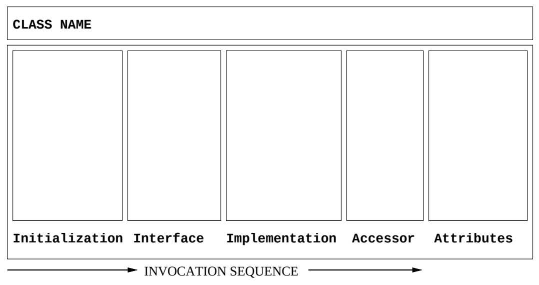


Figure 7.1: The decomposition of a class into layers.

Every class can be mapped on this “template” class blueprint which has the following layers:

1. **Creation/Initialization Layer**. The methods contained in this first layer are responsible for creating an object and initializing the values of the attributes of the object. We consider a method as belonging to the initialization layer, if one of the following conditions holds:

   - The method name contains the substring “initialize” or “init”.
   - The method is a constructor.

- In the case of Smalltalk, where methods can be clustered in so-called method protocols, if the methods are placed within protocols whose name contains the substring "initialize".

2. **(External) Interface Layer**. The methods of this layer can be considered as the *entry points* to the functionality provided by the class. A method belongs to this layer if one the following holds:

   - It is invoked by methods of the initialization layer.
   - In languages like Java and C++ it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, i.e., it is a method invoked from *outside* of the class, either by methods of collaborator classes or subclasses.

   We do not consider accessor methods for this layer, but to a separate layer.

3. **(Internal) Implementation Layer**. The methods within this layer are the ones doing the main work of the class, by assuring that the class can provide the functionality promised by the interface layer. A method belongs to this layer if one of the following holds:

   - In languages like Java and C++ if is declared as *private*.
   - The method is invoked by at least one method within the same class.

4. **Accessor Layer**. This layer is composed of accessor methods, i.e., methods whose *sole* task is to get and set the values of attributes. We take do not take into account lazy initialization of attributes.

5. **Attribute Layer**. The attribute layer contains all attributes of the class. The attributes are connected to the other layers by means of *access relationships*, i.e., the attributes are accessed by methods.

Representing Methods and Attributes in a Class Blueprint. Within the layers of each class we represent methods and attributes using colored boxes of various size and shape.

Size and Shape of Methods and Attributes. We use the width and height of the boxes to reflect metric measurements of the entities which are represented by the boxes, as we see in Figure 7.2. This approach has been presented in [Lan99] and [DDL99]. For the method boxes we use the metric *lines of code (LOC)* for the height and the *number of invocations (NI)* for the width [LK94, HS96]. For the attribute boxes we use the metrics *number of accesses from within the class (NLA)* for the width and *number of accesses from outside of the class (NGA)* for the height [Lan99]. Note that the total number of accesses on an attribute is the sum of NGA and NLA.

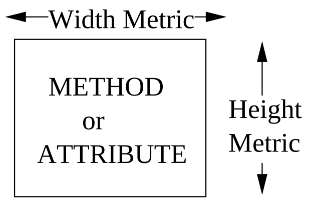


Figure 7.2: A graphical representation of methods and attributes using metrics.

The Use of Colors in a Class Blueprint. Colors are used to display supplementary information in a class blueprint. In the Table 7.1 we present a list of the colors we use in the following figures.

Description	**Color**
Attribute	blue
Abstract method	cyan
Extending method. A method with the same name in the superclass which performs a *super* invocation	orange
Overriding method. A method which completely redefines the behavior of a method in the superclass with the same name *without* invoking the superclass method	brown
Delegating method. A method which delegates the functionality it is supposed to provide, by forwarding the method call to another object	yellow
Constant method. A method which returns a *constant* value	grey
Initialization layer method	green
Interface and Implementation layer method	white
Accessor layer method	red
Invocation of a method	black line
Invocation of an accessor. Semantically it is the same as a direct access	cyan line
Access of an attribute	cyan line

Table 7.1: A color schema for class blueprints.

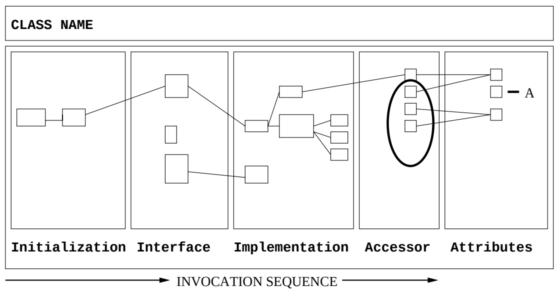


Figure 7.3: The basic filled structure of a class blueprint.

In [LD01] we presented a categorization of classes based on their blueprints, i.e., based on the way they display themselves using the approach described in the previous section. The categorization stems from the experiences we obtained while applying our approach on several case studies. In the following section we categorize the classes based on their internal structure, while in the second part we extend the context to the inheritance hierarchy where the class resides. The only kind of collaboration between classes we discuss in here is inheritance.

7.3.2 The Single Class Perspective

In this part we introduce a categorization of classes based on their blueprint without considering the surrounding sub- and superclasses. Based on the class blueprint we make statements regarding the *internal implementation* aspects of the class. Note that a class can belong to more than one of the

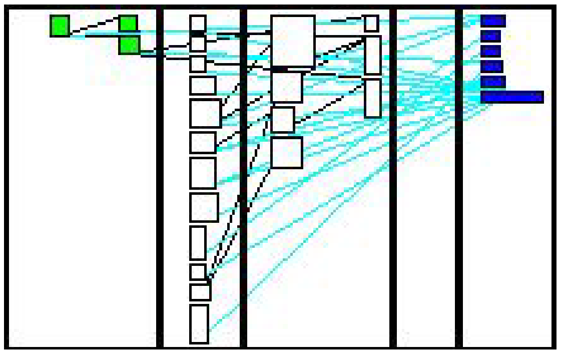


Figure 7.4: An actual blueprint visualization of a class.

categories presented here. We limit ourself to the presentation of two class blueprints, interested readers may refer to [LD01].

Single Entry. We define a *single entry* class as one which has very few or only one entry point to the interface layer. It then has a large implementation layer with several levels of invocation relationships. Such classes are designed to deliver only one yet complex functionality. Classes which implement a specific algorithm belong to this type. In Figure 7.5 we see an actual single entry class.

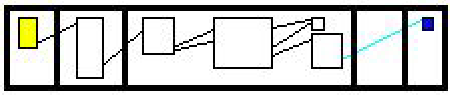


Figure 7.5: The blueprint of the class MSEXMIDTDProducer: a *single entry* class without accessors.

Data Storage. *data storage* is a class which mainly contains attributes whose values can be read and written by using accessor methods. Such a class does not implement any complex behavior, but merely stores and retrieves data for other classes. The implementation layer is often empty, as the class functionality does not need complex mechanisms to be delivered. The attribute layer often contains several attributes which are accessed directly or through accessor methods. In the Figure 7.6 we see an example of a data storage class.

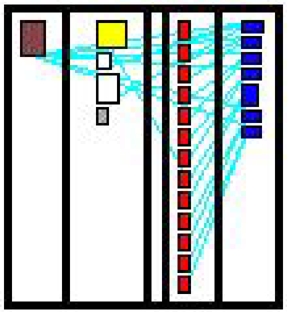


Figure 7.6: The blueprint of the class MSEModelAttributeDescriptor: a *data storage* class. We see that there are many accessors to the many attributes. The internal implementation layer is empty.

7.3.3 The Inheritance Perspective

We expand the categorization of class blueprints by considering the way the classes make use of the inheritance relationships with their ancestors and descendants. This perspective adds considerable meaning to the class, as the functionality which can be provided by the class is in fact distributed across the inheritance chain the class belongs to. In the case of inheritance we visualize every class blueprint separately and put the subclasses below the superclasses, similar to a inheritance tree layout, as we see in Figure 7.7.

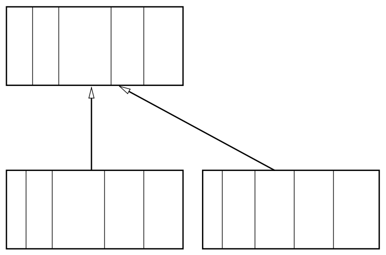


Figure 7.7: The visualization of class blueprints in the context of inheritance.

Taking into account a whole inheritance hierarchy using our approach leads to a whole range of new class categories. In this section we make the following distinction:

1. *Definers* are classes which reside at the top of a hierarchy. They may define some kind of interface behavior for their subclasses, apart from providing functionality of their own.

2. *Specializers* are leaf classes in inheritance hierarchies and implement and refine behavior at the bottom of the hierarchies.

3. *Inbetweeners* are classes which are none of the above. However, often they can be put into one of the two categories nonetheless. For example, if a class in the higher part of an inheritance hierarchy is not a definer class, it can nonetheless be classified as such if it shows the properties of a definer class.

The details of the categorization are described in [LD01]. We only present here an example of class blueprints in the context of inheritance. The Figure 7.8 shows a *constant definer* superclass (i.e., a class defining constants) and two *talking overriders* (i.e., two classes overriding their superclass while invoking superclass behavior), which are also *siamese twins* (i.e., the two classes holds similar taste).

7.4 Ongoing Work

Software Classifications for Browsers. Up to now we experimented class blueprint as a separate tool to understand classes. We would like to see the impact of having class blueprints integrated in an IDE like the Smalltalk showing constantly the class browsed or even been edited. Moreover, we see class blueprints as one way to provide synthesized views on a class we are currently starting research on new IDEs. Indeed, tools are the base for supporting program understanding. To promote code understanding not only as a separate process but rather integrated in the development life cycle we started to work on a development environment supporting code understanding. A first problem that

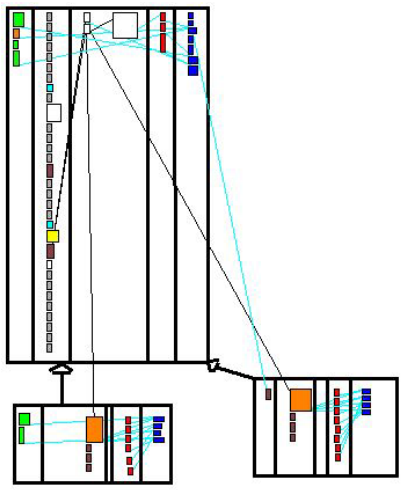


Figure 7.8: The blueprints of classes from the MOOSE case study show a *constant definer* superclass and two *talking overriders*, which are also *siamese twins*.

we address is the lack of uniformity of the proposed information. Smalltalk is a good example of such a lack, for example, the developer can browse the code using the class browser, he can ask for all the senders or implementors of a given method selector which is really powerful and the base of browsing code. However, the results cannot be manipulated and stored in the basic class browsers. As a result, different browsers have to be opened. As a first step towards a unified environment, we defined a class browser which supports software classification [WD01]. A classification is a group of entities which can be classes, applications, categories, method senders, method implementers and even inspected objects. The Classification Browser is based on a minimal classification framework that we developed. The idea was to create the simplest approach in reaction of the Classification Browser [DeH98] that was a far too complex browser. In Chapter 13 we elaborate more some other aspects that new IDE could have to support code understanding.

Revealers. We are investigating the concept of *code revealers* [Mal01]. The idea is to identify some simple heuristics that when applied to a class can change the perception we have and give meaningful information to understand it. The idea however is not to identify cluster of method ready for class splitting but to identify basic groups that a developer aware of the meaning of the heuristics could interpret and combine. For example, a possible revealer is a function that groups all the methods according to the way they access simultaneously the instance variables of a class. Another one is grouping all the methods only calling methods defined locally or redefined locally. Code revealers are simple heuristics that decompose classes in different clusters that the reengineers have to interpret. We are currently investigating to find a set of useful revealers. MOOSE EXPLORER besides being a generic browser allowing to navigate FAMIX models, serves as a laboratory for identifying code revealers that we would to integrate into the IDE we are building.

7.5 Future Tracks

Collaboration blueprints. Extending class blueprints to take into account classes collaboration is definitively an important future work. However, it is certainly more difficult because of (1) the different types of collaborations between objects represented already by the references they maintain to each other (instance variables, method arguments...), and (2) the need of temporal information to sort method call.

7.6 Contributions

Implementations. CodeCrawler the tool presented in the chapter 6 has been extended to support the concept of blueprint. The Classification frameworks is available for different Smalltalk dialects.

Bibliography.

[LD01] M. Lanza, S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In Proceedings of OOPSLA'2001 *(International Conference on Object-Oriented Programming Systems, Languages and Applications).*, 2001.

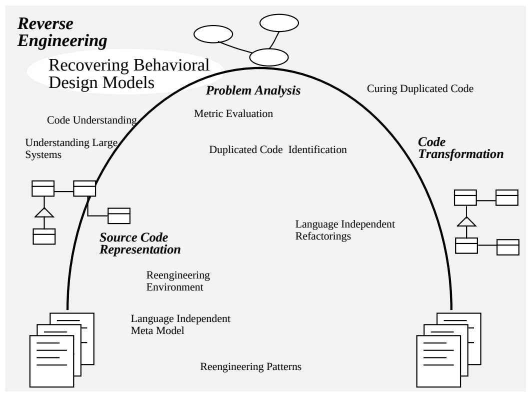
[Malo01] P . Malorgio Heuristics for Object-Oriented Program Analysis and Understanding (temporary title). Master Thesis of the University of Berne.

[WD01] R. Wuyts and S. Ducasse Software Classifications: a Uniform Way to Support Flexible IDEs. 2001, Working Paper.

Chapter 8

Iteratively Recovering Behavioral Views

Reverse engineering techniques traditionally make use of static information only, restricting the kinds of questions which can be answered. On the other hand, program understanding approaches which use dynamic information present very fine-grained views of an application. Such approaches are confronted with the challenge of extracting high-level views from a program trace. We present how we combined static and dynamic information to support the extraction of behavioral views in an iterative appraoch similar to the reflection model proposed in [MN97].


Reverse Engineering
Recovering Behavioral Design Models
Problem Analysis
Curing Duplicated Code
Code Understanding
Metric Evaluation
Understanding Large Systems
Duplicated Code Identification
Code Transformation
Source Code Representation
Language Independent Refactorings
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

After briefly presenting the state of the art, we present how we *combine* dynamic information with static information to support the generation of *tailorable* software views [RD99] [Ric01].

8.1 Problem

Understanding the structure and behavior of an application being developed or maintained is essential throughout the software development cycle. While documentation should address this need, it is often neither complete nor up-to-date and may not at all address the particular questions an engineer is interested in. Recovering such information from an existing application is an important aid to engineers confronted with a variety of software engineering tasks.

Recovering architectural documentation from code is crucial to maintaining and reengineering software systems. Reverse engineering and program understanding approaches are often limited by the fact that (1) they propose a fixed set of predefined views and (2) they consider either purely static or purely dynamic views of the application.

8.2 State of the Art

Although some work on program analysis tools [CCdCDL98] and debuggers [LHS97] is related to ours, these kinds of tools have a different goal than reverse engineering tools and are not well adapted to creating high-level models and revealing overall structures in the software. Here we relate our work to research on reverse engineering, the use of dynamic information for program understanding and declarative reasoning about program structure.

Reverse engineering. Reverse engineering approaches all seek to represent the software at a higher level than that of the information which is directly extracted from the code. They differ, however, in their solutions to the following main issues: the data model on which the tool operates, the strategy for creating a high-level model and the kinds of views offered.

The MANSART tool [HYR96], as well as the approach described in [FTAM96], requires information obtainable from an abstract syntax tree (AST) of the program, and uses "recognizers" to detect language-specific clichés associated with specific architectural styles. Each style can then be viewed as a simple graph. Rigi [Mül86], [WTMS95] and the reflexion model tool [MN97] both use any set of relations extracted from the code. The Rigi tool allows user defined grouping of the source model. The reflexion model expects an engineer to define a high-level model and a declarative mapping from the source relations to this model. Its view then reports how close the high-level model comes to describing the source code. Dali [KC98] is a workbench which integrates several extraction tools and allows for the combination of the views obtained from these different sources.

Use of dynamic information in program understanding. Program traces have been used in software maintenance to locate code implementing a particular program feature [WS95], to extract business rules from COBOL programs [RS93], and to discover program invariants[Ern99]. [Sys99], [Sys00a], [Sys00b] use program trace and dynamic information to understand Java programs, in particular state diagrams can be automatically generated.

For the understanding of object-oriented software, much of the work on using dynamic information has focused on techniques for visualizing the large amount of information [DPHKV93] [DPKV94] [KM96]. [DPS99] visualizes execution patterns for identifying memory leaks. Program Explorer [LN95a] allows the filtering of events or objects of interest using static and dynamic information, but abstractions of a granularity greater than a class cannot be viewed. ISViS [JSB97] is a visualization tool which displays interaction diagrams using a mural technique and also offers pattern matching capabilities to aid in identifying recurring patterns of events. Few tools offer architectural level visualizations. Sefika et al.[Sef96] can display the interactions of architectural units such as subsystems, but their approach requires an instrumentation specific to the application. More recently, Walker et al.[WMFB$^+$98] use program animation techniques to display the number of objects involved in the execution, and the interaction between them through user-defined high-level models.

Declarative reasoning about program structure. Prolog has been used as an inference engine for other reverse engineering approaches, such as for the understanding of dataflow in Pascal programs [CCdC92] and for reasoning about dependency relationships between modules [CMR92]. It has also been used to query static program information to find structural design patterns [KP96] and to detect violations of programming conventions and rules [Sef96] [Wuy98] [Wuy01]. The detection of behavioral design patterns may be easier with the incorporation of dynamic information; the discovery of architectural patterns [BMR$^+$96] presents a more challenging problem. In SOUL [Wuy01] a logic-programming language is integrated in the Smalltalk development environment, allowing one to reason about static structure and to enforce design rules and conventions.

8.3 Iteratively Recovering High-Level Views from Static and Dynamic Information

Our approach is based on the combination of user-defined queries which allow an engineer to create high-level abstractions and to produce views using these abstractions. Such queries can manipulate static and dynamic information [RD99].

The contribution of our approach is to allow an engineer to steer her investigation of the code through an iterative process similar to that described by Murphy [MN97]. This is possible because the engineer can specify declaratively the kinds of views which are of interest. An initial view answers some questions and introduces new ones, and views can be refined to different levels of abstraction. Since dynamic information is available, as well as static, a large range of questions can be answered.

8.3.1 Modeling OO Programs and Their Execution

We model static and dynamic aspects of an object-oriented application in terms of logic facts. Both static and dynamic information are stored in a single logic database.

Static and Dynamic Relations. We represent static information about an application using Prolog facts (see table 8.1). This information is extracted using static analysis tools and represented in the FAMIX model (see Chapter 4). Dynamic information is represented as facts about method invocations in a program's execution. These are numbered according to sequence order (SN) and stack level (SL). Each send or indirectsend fact corresponds to to the invocation of an observed method on an instance of a class.

Static Information
class(ClassName, SourceAnchor) a class and its source artifact
superclass(SuperClass, SubClass) an inheritance relationship
attribute(Class, AttributeName, AttributeType) class defines an attribute of a certain type
method(Class, MethodName, IsClassMethod, Category) a class defines a method belonging to a category
access(Class2, Attribute, Class1, Method) an attribute of Class1 is accessed by Method of Class2
invocation(Sender, Method, ReceivedMethod, Candidates) Method of Sender invokes ReceivedMethod on one of the Candidates
Dynamic Information
send(SN, SL, Class1, I1, Class2, I2, M2) an instance I1 of Class1 invokes method M2 on instance I2 of Class2. SN is the sequence number of the event, and SL is the stack level of the method call.
indirectsend(SN, SL, Class1, I1, M1, Class2, I2, M2) an instance I1 of Class1 sends the message M1, which is unobserved. The next observed invocation is the execution of method M2 on instance I2 of Class2.

Table 8.1: The basic static and dynamic relations.

Derived Relations. On top of these basic facts, we built a set of rules that form a logic layer above the database facts. The following rules illustrate the approach. Rule 1 below specifies that an instance of Class1 invokes a Method on an instance of a metaclass of Class2, where the Smalltalk category of Method is instance creation (Smalltalk creation methods, defined at the class level, appear as methods of a metaclass in our model. There is an implicit Smalltalk convention to group instance creation methods into a category named instance creation). Rule 2 allows one to go up the inheritance hierarchy to find the Smalltalk category of the Method, in case it is not defined by Metaclass. Rule 5 thus defines a *create* relationship between two classes – an instance of class Class1 creates an instance of Class2.

```
rule1:     sendsCreate(Class1,Class2) :-
              invokesMethodClass(Class1,MetaClass,Method),
              metaclassOf(MetaClass,Class2),
              methodCategory(MetaClass,Method,'instance creation').

rule2:     methodCategory(Class,Method,Category) :-
              method(Class,Method,_,Category).

           methodCategory(Class,Method,Category) :-
              inHierarchy(Superclass,Class),
              method(Superclass,Method,_,Category).
```

Discussion. Though the model we present for representing OO programs and their execution is language-independent, its interpretation is not. For example, in table 8.1, the interpretation of SourceAnchor in the relation class(ClassName,SourceAnchor) is language-specific. In Smalltalk this corresponds to a class category, in C++ to a file name and in Java to a package name. Furthermore, whereas in Smalltalk no type information is available for attributes, or for precise identification of receiver candidates in an invocation, this kind of information can be easily obtained from statically typed languages like C++ and Java. Thus, some of the rules for the derived relations hold for all OO languages, for example overrides(Class,Subclass,Method), whereas others, such as sendsCreate(Class1,Class2) are more language specific.

A querying approach similar to ours, but using static information only, has also been applied by Kramer [KP96], Sefika [Sef96], and Wuyts[Wuy98] where rules are used for the detection of certain structures or their violations. In Program Explorer [LN95a] both static and dynamic information are represented as logic facts. The static information is used to filter out execution events to be visualized, in order to support the discovery of design pattern in C++ code.

Our goal, however, is not only to detect certain structures. Our approach supports the specification and generation of a wide range of views of an application to aid in architectural recovery. The next section discusses the specification and generation of views.

8.4 Generating High-Level Views

We define a view of an application as a set of components and the connectors between them [SG96]. In a view two components C_1 and C_2 are connected by a connector of type R if there exists at least one member e_1 of C_1 and one member e_2 of C_2 such that $R(e_1,e_2)$ holds. An element e is a member of at most one component C. To specify a view we then use Prolog rules to define:

(1) a clustering to components $C_1, C_2, ..., C_N$, which defines a partition on $E = \{e_1, e_2, ...e_m\}$, the set of all the elements e in our model.

(2) a relation $R : E\text{x}E \rightarrow \{0, 1\}$, which specifies whether or not a certain relationship holds between two elements.

For example, to generate a view that shows the message sends (from an executed scenario) between the Smalltalk class categories of the HotDraw framework we define a relation R and a clustering C using rule 3 and 4 respectively, as shown below. In this case E is the set of all classes in the HotDraw framework, and is defined implicitly by rule 3. The Prolog query createView(invokesClass,allInCategory) then generates a view which is displayed as a graph (using the dot tool [KN]), as seen in Figure 8.1. Each node in the graph corresponds to a HotDraw class category and each directed edge $A \rightarrow B$ means that at least one instance of a class in category A invokes a method on an instance of a class in category B.

```
createView(invokesClass,allInCategory).

R rule3:   invokesClass(Class1,Class2) :-
             invokesMethodClass(Class1,Class2,_).

C rule4:  allInCategory(Category,ListOfClasses) :-
             setof(Class,class(Class,Category),ListOfClasses).
```

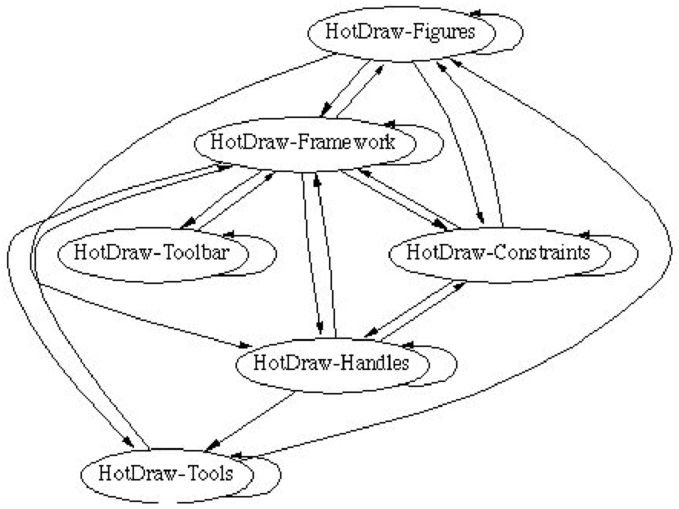


Figure 8.1: Invocations between class categories.

This view gives us a coarse idea of the communication between parts of the HotDraw framework. However, since the category HotDraw-Framework groups together some of the main classes, this clustering must be broken down to get a better understanding of the behavior of the application (see [RD99] for a complete scenario). The figures 8.2 and 8.3 show other views that have been generated during a scenario to understand HotDraw. The Figure 8.2 presents a new view which gives us the creation relationships between the classes, rather than the components. The Figure 8.3 presents a trace sequence corresponding to the creation of an instance of RectangleFigure. Our prototype allows us to display also relationships in which arcs between components are labeled. We are as well investigating different kinds of views and how they are best displayed graphically, within the limitations of a simple graph layout tool.

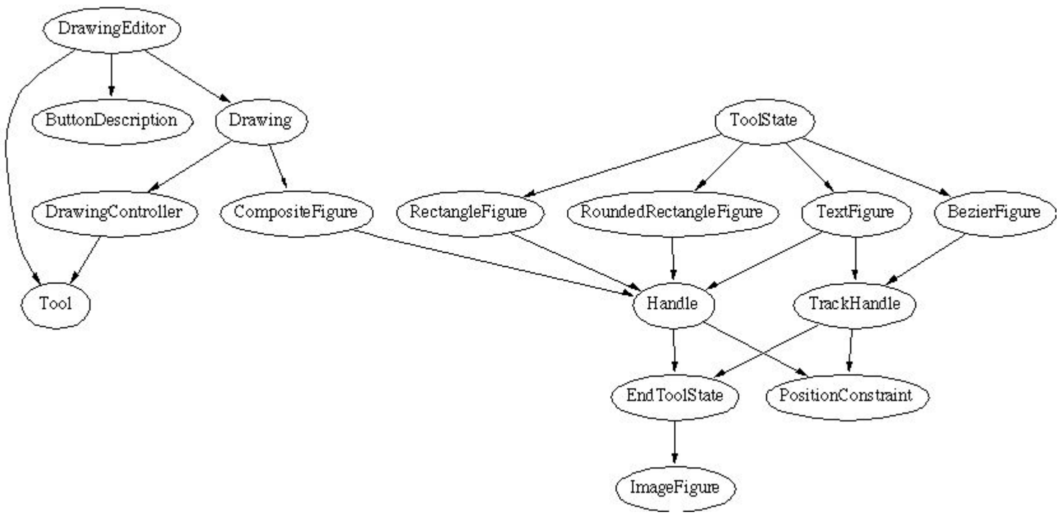


Figure 8.2: A filled edge $A \rightarrow B$ means that there is an instance of class A which creates only one instance of class B. A dashed edge $A \rightarrow B$ means that there is an instance of Class A which creates several instances of class B.

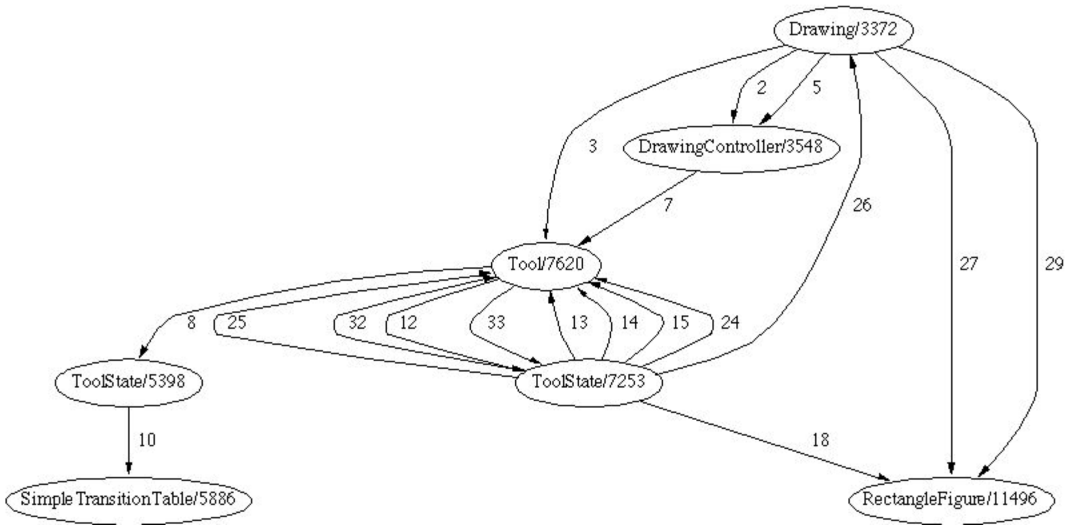


Figure 8.3: Instance invocations around creation of a Rectangle figure : invocations are numbered sequentially in the order they occur; missing numbers correspond to self sends - these have been omitted in order to make the graph more comprehensible. The one dashed edge corresponds to a creation event.

8.5 Validation and Discussion

We validate this approach on several case studies. One of them is, HotDraw, a framework for semantic graphic editors [BJ94]. To fill the logic database we parsed the code to obtain a static model of HotDraw. To obtain the dynamic information we instrumented all the methods defined for classes in the HotDraw categories using Method Wrappers[BFJR98], and ran a typical scenario on one of the sample applications, DrawingEditor, to generate an execution trace [RD99].

A view works as a catalyst for generating questions about the studied system and helps the engineer to focus the investigation of the code. It confronts the engineer with a new model to compare to his or her initial mental model and assumptions about the system. A view not only helped us to understand HotDraw by showing relationships between the entities, such as the creation between components, but also by provoking questions about the absence of expected relationships or the presence of unexpected ones.

Discussion. Dynamic information, though incomplete, is useful in reverse engineering because it acts like a program slice. Obtaining dynamic information, however, requires that the system be executable (and instrumentable) – and so this approach will not work for parts of systems, or other code which cannot be executed.

The difficulties of our approach are first the fact that the queries are expressed in Prolog and that not everybody can express views where backtracking is involved. As a partial solution, we propose a set of predefined views. However, although using Prolog was a good approach to experiment, we think that the unification is not central to the approach. Hence, alternative approaches could be used to express the queries. The second problem is that, as we any approach dealing with dynamic information, the quality of the scenario has an impact on the result. However, most of the time, we are interested in a particular aspect of a system, so the scenario can be clearly identified. The third point is more technical and related to the scalability of our approach. Dynamic information tends to produce extremely huge amount of data. Having to treat them can slow down the fast feedback loop that our approach requires. To reduce the amount of trace information generated, tracing can be more sparingly used, for example by instrumenting the application at only some of the methods or classes. However, such a solution may produce "black hole" in the dynamic trace where a set of invocations is lost because not recorded. An alternative could be to filter the ynamic information before analysis to keep only the relevant events. A clever solution would be a feedback from the query results to the instrumentation so that only relevant methods are instrumented.

The strength of our approach is the flexibility it offers in two respects: the kinds of views which can be recovered, and the kinds of questions which can be answered. First, our approach is not restricted to generating a fixed set of views, but allows an engineer to define views of interest by declaratively defining the relations to be displayed and the clustering to be applied. This makes it possible to create views of varying granularity and so to ask questions at different abstraction levels. Second, since our approach uses both static and dynamic information, it can answer a large range of questions about an application: where static information is less focused it is complemented by dynamic information; static information is used to cluster dynamic information into more manageable components and dynamic information provides answers to questions which cannot be answered with static information only.

8.6 Implementation Issues

Obtaining dynamic information is linked to the instrumentation of code or use of reflective language capabilities. Hence, in C++, it implies preprocessing of the code [DPHKV93] [LN95b] which can be complex or using a reflective extension [Chi95]. In Java, the extension of the virtual machine is one approach [DPS99] or to use a reflective extension of the language [Chi00]. Reflective languages ease the instrumentation of code. For example, in CLOS, the MOP provides the necessary entry points [KdRB91]. While Smalltalk is a reflective language [FJ89], [Riv96b], it never really took benefit of advances in Meta-Object Protocol design to provide a clean protocol to the programmer.

Hence in Smalltalk, there are several possible ways to control message passing each with its own pros and cons. In [Duc97] and [Duc99] we define a conceptual framework to compare the different approaches stressing the trade-offs of each. The different criterias are the *objects granularity*, i.e., can we control a specific instance, a set of object or all the instance of a class, the *control granularity*, i.e., can we control any message sent even the unknown ones, some specific methods, or all the methods defined in a class, and the environment integration, i.e., can we control any part of the system in a transparent way. With these criterias, we compared three main techniques: specialization of error handling, exploiting the VM method lookup implementation, and method substitution.

8.7 Ongoing Work

Roles and Collaboration Extraction. In contrast to procedural applications, where a specific functionality is often identified with a subsystem or module, the functionality in object-oriented systems comes from the cooperation of interacting objects and methods [WMH93], [Lau98]. In designing object-oriented applications, the importance of modelling how objects cooperate to achieve a specific task is well recognized [WBW89], [BC89], [HHG90], [Ree96], [RG98], [BRJ99]. We propose an iterative approach based on the filtering and manipulation of dynamic information to recover roles and collaborations. We are developing a tool which uses an execution trace of a program to find meaningful collaborations and help the reengineering identifying the roles that classes play in certain collaborations[RD01a].

Dynamic Information to Support SUnit Generation. Being able to specify tests before modifying a system provides a security than any reengineer would like to have. Moreover, new development methodology like eXtreme Programming promotes the definition SUnit tests with any class [Bec99]. However, when facing a legacy system, most of the time tests do not exist and it is really hard to derive tests. Based on the hypothesis that at least we can run some scenarios, we propose to use and annotate dynamic information generated by these scenarios to help the reengineer to identify and generate SUnit tests.

8.8 Future Tracks

The experiences we made show the value of the proposed approach. We think that dynamic information is a really important source of information for reengineering and program understanding. However, it may be difficult to work with due to the huge amount of information it provides. We think that it would be interesting to apply static pattern matching approaches such as [PP94] to dynamic information.

Dynamic information could be used to qualify the relationships between objects [Pen95] in the context of design extraction or design patterns [Bro96]. We are also thinking that advanced debuggers that would customize the information shown could be an interesting research axe [LHS97], [LHS99].

8.9 Contributions

Bibliography.

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

[Duc99b] S. Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[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)*, pages 13–22. IEEE, September 1999.

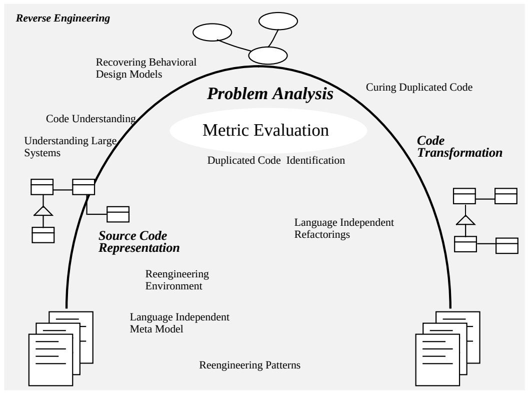
[RD01a] T. Richner and S. Ducasse, Iterative Recovery of Collaborations and Roles in Dynamically Typed Object-Oriented Languages, 2001, Technical Report University of Berne.

[Ric01] T. Richner Recovering Behavioral Design Models: A Query-based Approach, PhD Thesis of the University of Berne, 2001.

Chapter 9

Metrics for Reengineering

The visualization of applications we developed in Chapter 6 was based on metrics. However, metrics were used as a mechanism to change the view we have and add some semantical information on the graph. Metrics are one the simple code analysis that can be applied to identify problems in legacy application. The work presented here belongs to the *Problem Analysis* area as shown by the following Figure.



Reverse Engineering
Recovering Behavioral Design Models
Problem Analysis
Curing Duplicated Code
Code Understanding
Metric Evaluation
Understanding Large Systems
Code Transformation
Duplicated Code Identification
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

In this chapter we present the experiments we ran to evaluate how metrics can be used as tools for (1) steering reengineering efforts, mainly identify defects, (2) steering framework development efforts and (3) reverse engineering applications. We start by explaining the context of the work, then we survey the metrics literature and present the empirical studies we made: we show that size and inheritance metrics are not reliable for identifying problematic entities, then we show that these metrics can be used to assess the stabilization of a system. Finally we show how simple metrics can be used to reverse engineer code refactorings.

9.1 Problems

Research and industry pay a lot of attention to object-oriented application frameworks [JF88], [FS97]. However, identifying the right combination of reusable abstractions and consequently coming up with the right set of hotspots is known to be difficult and is best achieved via an iterative development process [GR95], [JGJ97]. However, project managers are often reluctant to adopt such an iterative approach because it is quite difficult to control without a reliable set of software metrics. To support project management, a metric-based analyzing system should not only help in assigning priorities to the parts of the framework that need to be redesigned first, it should also tell whether the framework design is improving. That is, a metric-based analyzing system should *detect problems* and *measure stability*.

To become a reliable project management support tool, metric-based analysis should be backed up by empirical data and case studies. From a literature survey, we learn that quite a number of object-oriented metrics have been proposed, but that little empirical data on their application is available [HS96], [MN96], [Mar97]. Moreover, no one mentioned the application of metrics in a context of framework development.

This shouldn't come as a surprise since framework technology is still relatively young and few mature industrial frameworks exist. Moreover, to validate a metrics analyzing system against an iterative development approach, one must have access to different releases of the same framework and compare measurements of each of them. Also, existing frameworks represent important strategic assets of companies and as such the framework source code is almost never accessible without signing non-disclosure agreements. Summarizing it is hard to do empirical studies, it is even harder to publish results and consequently empirical studies are seldom compared or reproduced.

9.2 State of the Art

Nowadays there is a plethora of metric definitions to evaluate design, code, productivity, and project cost [FP97], [HS96]. Metrics have long been studied as a way to assess the quality of large software systems [FP97] and recently this has been applied to object-oriented systems as well [LK94], [Mar98], [LS98], [Nes88], [HM95]. Metrics have been used to identify duplicated code [MLM96], [Kon97]. Some theories exist to assess the well-foundedness of a sound metrics [BMB96] [BDW99]. However, the following problems:

- (a) Limited empirical validation of the proposed software metrics exists. For example, [CK94] made the effort to define object-oriented metrics with a firm base in theorical measurements – which is heavily criticized [HM96] [BMB96]. However, an empirical validation of their results is still incomplete. Some data are collected but not really analyzed.
- (b) The use of software metrics to steer framework development is inexistent.

9.3 Metrics For Problem Detection and Stability Assessment

We took the approach to evaluate the current software metrics against real case studies. We wanted to evaluate if metrics could be used to detect problems in applications, to assess the stabilization of a system [DD98] [DD99a] and to determine if metrics could be used to understand the evolution of a system [DDN00a]. The research presented here does not aim at defining new metrics but at evaluating which of the simple metrics can be used reliably by engineers in a reengineering effort or framework development. As we only have access to the source code of applications we concentrated our attention on code metrics

Our approach can be perceived as naive by the experts of software metrics research. However, we deliberately choose to select simple software metrics for the following reasons:

- Simplicity of evaluation. With a simple definition, the evaluation of the metrics can be assessed. Sometimes, complex metrics are hard to understand.

- Simplicity of use. Simple metrics are normally simple to extract from source code. So reengineers do not need specific tools to extract them, and could reproduce our findings.

Selected Metrics. Table 9.1 lists the metrics evaluated in the study, including a short description and a reference to the definition of the metrics. All of the metrics are proposed by Chidamber & Kimerer [CK94] ($\diamond$) or by Lorenz & Kidd [LK94] ($\star$). However, we rule out some of the proposed metrics because they received serious critique in the literature (LCOM and RFC [CK94]), because the definition isn't clear (MCX, CCO, CCP, CRE [LK94]; LCOM [CK94], [EDL98]), because the lack of static typing in Smalltalk prohibits the computation of the metric (CBO [CK94]), because the metric is too similar with another metric included in the list (NIM, NCM and PIM in [LK94] resemble WMC-NOM in [CK94]), or simply because the metric is deemed inappropriate (NAC, SIX, MUI, FFU, FOC, CLM, PCM, PRC [LK94]).

Name/Ref.	Description
Class Size (computed for each class)	
WMC-NOM $\diamond$	Count all methods in class. This is the WMC metric of [CK94] where the weight of each method is 1.
WMC-MSG $\diamond$	Sum of number of message sends in all method bodies of class.
WMC-NOS $\diamond$	Sum of number of statements in all method bodies of class.
WMC-LOC $\diamond$	Sum of all lines of code in all method bodies of class
NIV, NCV $\star$	Number of instance variables, number of class variables.
Method Size (computed for each method)	
Mthd-MSG $\star$	Number of message sends in method body.
Mthd-NOS $\star$	Number of statements in method body.
Mthd-LOC $\star$	Lines of code in method body.
Inheritance (computed for each class)	
HNL (DIT) $\star$ and $\diamond$	Count number of classes in superclass chain of class. In case of multiple inheritance, count number of classes in longest chain.
NOC $\diamond$	Number of immediate children of a class.
NMO $\star$	Number of methods overridden, i.e., redefined compared to superclass.
NME $\diamond$	Number of methods extended, i.e., redefined in subclass by invoking a method with the same name on a superclass.
NMI $\star$	Number of methods inherited, i.e., defined in superclass and inherited unmodified.
NMA $\star$	Number of methods added, i.e., defined in subclass and not in superclass.

Table 9.1: Metrics selected for our experiments.

9.3.1 Experimental Setup

Besides being an industrial framework that provides full access to different releases of its source code, the VisualWorks user-interface framework offers some extra features which make it an excellent case for studying iterative framework development. First, it is available to anyone who is willing to purchase VisualWorks, which ensures that the results are *reproducible* [1]. Second, the changes between the releases are documented, which makes it possible to *validate experimental findings*. Finally, since

[1]To allow other researchers to reproduce our results, we clarify that we define the VisualWorks framework as the part of the Smalltalk system corresponding with the categories 'Graphics-Visual Objects', 'Interface-*', 'UI-*' (excluding examples) and 'Globalization' when present. This definition is based on the manuals and[How95].

the first three releases (1.0, 2.0 & 2.5) of the VisualWorks frameworks together represent a fairly typical example of a framework life-cycle [GHJV95], it is *representative*.

Indeed, VW1.0 –released in 1992– was pretty ‘rough’, but contained already all the core ingredients for a look-and-feel independent UI-builder. After releasing it to the market, addition of extra functionality demanded some major refactoring of the framework resulting in VW2.0 [Hau95]. The subsequent VW2.5 –released in 1995– included extensions to the framework which did not involve a lot of framework redesign and as such represents a consolidated design. Table 9.2 provides an overview of the evolution between the three releases. As can be seen, the shift from release 1.0 to 2.0 involved a lot of extra classes and methods, while the shift from 2.0 to 2.5 involved some extra methods and a few extra classes. This conforms with the documented amount of alterations between 1.0 and 2.0 (a lot) and 2.0 and 2.5 (a few).

	VW 1.0	VW 2.0	VW 2.5
Total number of methods	5282	7784	8305
Total number of classes	576	716	732

Table 9.2: Overview of the evolution between VisualWorks releases (metaclasses excluded).

9.3.2 Results

One important aspect of an iterative framework development process is *problem detection*, i.e., the ability to detect those parts of the framework that hinder future iterations. To evaluate the usefulness of a given metric for problem detection, we apply them on one release and check whether the parts that are rated ‘too complex’ are indeed improved in the subsequent release. If most of those parts are actually improved, it implies that the metric is reliable to detect flawed parts in the design. If most of the parts are not improved, it implies that the metric is unreliable, since it means parts are marked too complex that do not harm the natural evolution of the framework. Moreover, we are also interested in the effect of threshold values. Since all of the metrics flag a part ‘too complex’ when it exceeds a certain threshold value, we wonder whether the threshold value affects the quality of the metric. Therefore, we run every test with several threshold values.

In the context of our experiment: three releases of a medium sized framework (VisualWorks), we concluded that today’s size and inheritance metrics are not *reliable for detecting concrete problems* in the framework design.

However, those metrics are very good for measuring the differences between two releases and as such can be used to measure stability [DD99a]. For example, the metrics helps us to understand the way refactoring affected the inheritance tree. With VW1.0→VW2.0, a significant amount of classes experienced a change in HNL, implying that the inheritance tree has been refactored. However, with VW2.0→VW2.5 all classes retain their HNL value, thus the inheritance structure remained intact. Nevertheless, there are changes in NOC values, indicating that the inheritance tree has been changed by adding or removing leaves. This is a positive sign, showing that the inheritance tree in VW2.0 has indeed stabilized.

Concerning the application of metrics for problem detection, we point out that the distribution of the measurements showed that 80% of the parts occupy the lowest 20% of the distribution scale. From this observation, we forward the rule of the thumb that ”any metric that results in a 80/20 distribution for a particular framework design, should not be used for problem detection”.

9.4 Heuristics for Evolution Understanding

In contrast to more traditional techniques which analyze a single snapshot of a system, we focused the reverse engineering effort by determining where the implementation has changed. First we focused on evaluating if heuristics based on metrics could detect refactorings [DDN00a]. Then we built a flexible query engine to be able to express complexer queries over multiple versions of a system [Ste01] [DLS00].

Experimental Set. We validate the applicability of the heuristics via an experiment involving three case studies: the VisualWorks framework ([Hau95], [How95]), the HotDraw framework ([BJ94], [Joh92]) and finally the Refactoring Browser ([RBJ97]). These case studies have been selected because they satisfy the following criteria.

- Accessible. The source code for the different versions of these frameworks are publicly accessible, thus other researchers can reproduce or falsify our results.
- Representative. Each of the three case studies is a successful software system which has undergone successive refactorings to address changing requirements.
- Independent. All frameworks were developed independently of our work, which implies that our experiment has not influenced the development process.
- Documented. The features that changed between the different versions are documented, making it possible to validate some experimental findings.

All of these case studies are implemented in Smalltalk. Nevertheless, each of these case studies represents a different kind of software system.

- VisualWorks is the only industrial system in the experiment. VisualWorks is a framework for visual composition of user-interfaces independent of the Look and feel of the deployment platform. It is a typical black-box framework, in the sense that programmers using the framework are supposed to customise its behaviour by (visually) configuring objects. The framework has quite a large customer base, so the framework designers must consider backward compatibility issues when releasing new versions. The versions we measured during the experiment are the versions that have been released during the time span of 1992-1996.
- HotDraw is a framework for building 2-dimensional graphical editors. HotDraw is one of the better known framework experiments, among others because of its pattern style documentation. It is a typical white-box framework, as users of the framework are supposed to subclass framework classes in order to reuse the HotDraw design. The HotDraw framework itself was implemented in VisualWorks/Smalltalk. Therefore, we numbered the versions according to their corresponding VisualWorks release.
- Refactoring Browser. This case study is the only software system which is not a framework. Nevertheless, it is interesting because it is a good example of an iterative development process. Rather than measuring every single increment, we compared a first stable version with the latest version that was available at the time of our experiment. Our first version corresponds to the 2.0 release (April 97) while the second version is the 2.1 release (March 98).

Finding Refactorings by Metric Changes. Since changes of object-oriented software are often phrased in terms of refactorings, we proposed a set of heuristics for detecting refactorings by applying lightweight, object-oriented metrics to successive versions of a software system. We examined four heuristics for identifying refactorings and investigated how this knowledge helps in program understanding. Each heuristic is defined as a combination of change metrics which reveals refactorings of a certain kind. One heuristic may occasionally miss refactorings, or misclassify them, but such mistakes are typically corrected by one of the other heuristics.

The four heuristics are described below and illustrated in the Figure 9.1:

- *Split into Superclass / Merge with Superclass*. This heuristic searches for refactorings that optimise the class hierarchy by splitting functionality from a class into a newly created superclass, or that merge a superclass into one or more of its subclasses. That is, we look for the creation or removal of a superclass, together with a number of pull-ups or push-downs of methods and attributes.

- *Split into Subclass / Merge with Subclass.* This heuristic searches for refactorings that optimise the class hierarchy. However, it takes the viewpoint of the superclass (using changes in the number of children as the main symptom), while the other provides the perspective of the subclass (triggering on changes in the length of the inheritance chain). Thus the refactorings we are looking for here split functionality from a class into a newly created subclass, or merge a subclass with one or more of its subclasses. That is, we look for the creation or removal of a subclass, together with a number of pull-ups or push-downs of methods and attributes.

- *Move to Other Class (Superclass, Subclass or Sibling Class).* This heuristic searches for refactorings that move functionality from one class to another. This other class may either be a subclass, a superclass, or a sibling class (i.e., a class which does not participate in an inheritance relationship with the target class, although it usually has a common ancestor). That is, we look for removal of methods, instance variables or class variables and use browsing to identify where this functionality is moved to.

- *Split Method / Factor Out Common Functionality.* This heuristic searches for refactorings that split methods into one or more methods defined on the same class. That is, we look for decreases in method size and try to identify where that code has been moved to.

We experimentally validate the applicability of these four heuristics by testing them on three case studies. Each case study is representative, in the sense that each is a successful software system which has undergone successive refactorings to address changing requirements. Our results indicate that our approach can be successfully applied to discover which parts of a system have changed, how they have changed, and, to some extent, why the designers chose to change them. [DDN00a] presents a deep analysis of the approach whose main features are:

- It concentrates on the relevant parts, because the refactorings point us to those places where the design is expanding or consolidating.

- It provides an unbiased view of the system, as the reverse engineer does not have to formulate models of what is expected in the software.

- It gives an insight in the way classes interact, because the refactorings reveal how functionality is redistributed among classes.

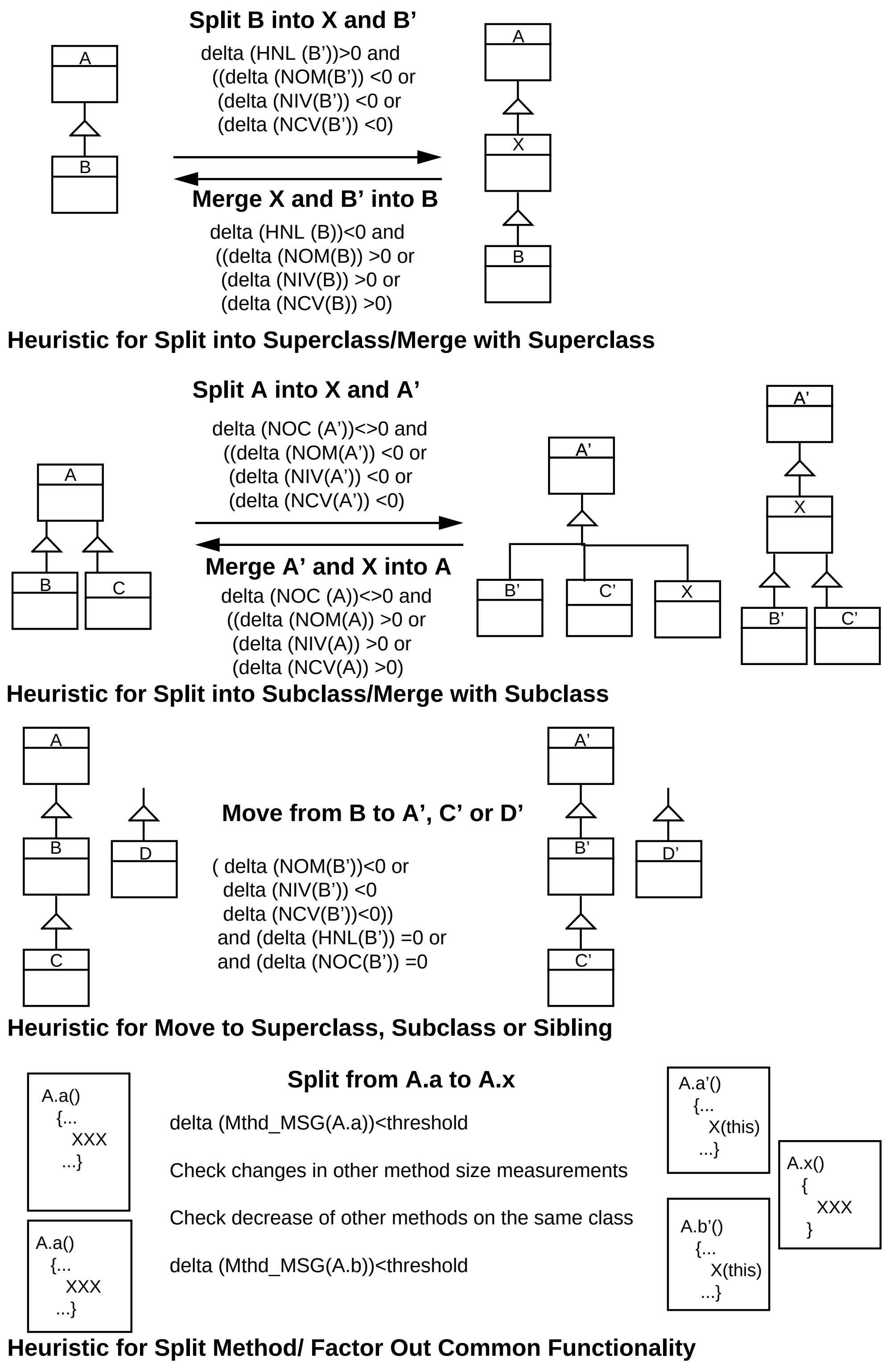


Figure 9.1: Heuristics for refactorings identification.

9.5 A Query Engine for Evolution Recovery

In [Ste01] we evaluate how metrics can (1) help to understand one single application by filtering and selecting entities and (2) the evolution of an application by proposing a catalog of queries that act one or multiple versions of a system. The approach is based on the definition flexible query engine that can interrogate multiple versions of an application [DLS00]. This catalog of queries contains different sort of queries. Each query enables us to investigate a part of the code structure or to filter the model from irrelevant data. We group our queries in the following functional categories:

- Filtering: Source code entities are filtered to obtain a working model. The selection criteria can be a metric threshold for example.
- Change: size and change metrics are calculated.
- Subsystem Analysis: Entities are grouped into subsystems based on criteria and dependencies between subsystems are identified.
- Hierarchy Analysis: Changes in class hierarchy (attributes, methods push up down,...) are identified.
- Move: Features moved between classes is identified.
- Renaming: Renamed entities are identified using metrics signatures.

This approach has been validated by L. Steiger on C++ frameworks during a 6 months visit at Nokia Research Center. However, even if the approach of having a flexible query engine was successful, the validation of the findings with the C++ engineers was not effective because of miscommunication between the business units and the research center.

9.6 Future Tracks

- We want to increase the number of case studies. However, this is a difficult task because we have to find different versions of a framework. We are currently investigating Swing and Borland C++ librairies to see if our results get confirmed. We would like to see if the rule of the 20%/80% rule is confirmed for other frameworks.
- When dealing with evolution one important problems is the large amount of information. Therefore we would like to evaluate how the approach proposed in Chapter 6 which condenses information can be applied to understand the evolution of an application.
- One of the main problem while working on several versions of a system is the renaming of entities mainly classes or methods. Different approaches (like metrics based identification, AST matching [Wuy98], [Wuy01]) can be applied to identify if the entity has been removed or simply renamed in subsequent version. We would like to evaluate them.
- One of the problem we encountered when we introduced grouping facilities into MOOSE was that we cannot simply compute metrics for a group because it may contain various elements. We would like to see if based on the meta-model description we are currently designing for MOOSE, we can compute simple size metrics in an automatic manner. This would allow us to propose simple metrics to support the visualization of groups in CODECRAWLER.

9.7 Contributions

Implementations. MOOSE supports the extraction and computation of a large number of software metrics. Such metrics are used by CODECRAWLER for displaying purposes. L. Steiger designed MOOSE FINDER, a simple and flexible query engine that allows one to define and compose metric queries over multiple versions of an application.

Bibliography.

[DD98] S. Demeyer and S. Ducasse. Do metrics support framework development? In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, 1998.

[DD99] S. Demeyer and S. Ducasse. Metrics, do they really help? In J. Malenfant, editor, *Proceedings LMO'99 (Languages et Modèles à Objets)*, pages 69–82. HERMES Science Publications, Paris, 1999.

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

[DLS00] S. Ducasse, M. Lanza, and L. Steiger. A Query-Based Approach to Support Software Evolution In Proceedings of *ECOOP'2000 International Workshop of Architecture Evolution*, 2000.

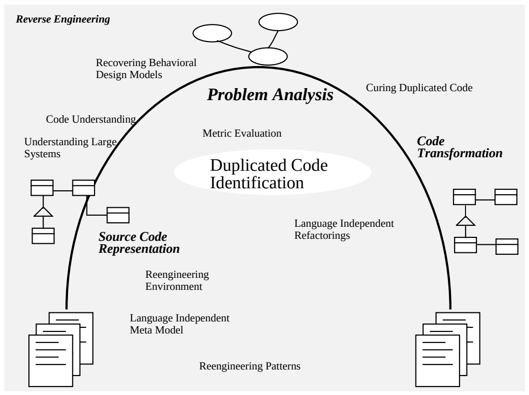
[Ste01] L. Steiger. Recovering the Evolution of Object-Oriented Software Systems Using a Flexible Query Engine Master Thesis of the University of Berne.

Chapter 10

Language Independent Detection of Code Duplication

"Parsing the program suite of interest requires a parser for the language dialect *of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities."* [BYM+98].

Identifying duplicated code belongs to the *problem analysis* area as illustrated by the following Figure. Our approach of duplicated code identification is not based on the meta-model we developed since it uses textual representation as source of input. Other researchers have used metrics which could have been expressed using FAMIX (See Chapter 9).


Reverse Engineering
Recovering Behavioral Design Models
Problem Analysis
Curing Duplicated Code
Code Understanding
Metric Evaluation
Understanding Large Systems
Code Transformation
Duplicated Code Identification
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

In this chapter after presenting the relevance of duplicated code identification, we give an overview of the state of the art in duplicated code identification. Then we present our approach that is based on language independence and simple line matching. We then discuss how duplication information should be reported and propose a methodology to support the planning of the tasks to cure it. In the next chapter we show how in the specific context of object-oriented programming analysis based on scenario can help to understand duplicated code and steer code refactorings.

10.1 Duplicated Code

In [FBB$^+$99], code duplication is mentioned as the top ten code smells, i.e., symptoms that the code should be refactored. Existing research suggests that a considerable fraction (5%-10%) of the source ode of large-scale computer programs is duplicated code [LPM$^+$97], [Bak95]. [BYM$^+$98] reports 12.7% of duplication. This was confirmed by the results the industrial case studies we evaluated during this research where we found up to 25% of duplication. It is worth to be noted that such a practice is broadly used independently of the language used — Cobol, C++, Java Smalltalk and Python in our case.

Although code duplication can have its justifications, it is considered bad practice. Especially during maintenance (estimated at 70% of the overall effort for producing a software system [Som96]) unjustified duplicated code gives rise to severe problems:

(a) If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked.

(b) Code duplication increases the size of the code, extending compile time and expanding the size of the executable.

(c) Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality.

Consequently, there is a need for approaches that support software maintainers in detecting duplicated code. While this may seem simple at first, our experience has shown that it is not, due to the following reasons:

- **Size.** There is a large amount of code to be checked, easily hundreds of files comprising thousands of lines of code. Consequently, any representation of that code (especially parse trees) quickly consume large amounts of memory or disk space.
- **Lack of search patterns.** As we do not know *a priori* what has been duplicated, a tool has to find *what* has been copied as well as *where* it has been copied. Searching for specific patterns in a `grep` like approach will not work.
- **Parsing Technology.** Different approaches for detecting duplicated code transform the source code into an intermediate representations. This implies having lexical analyzers [Bak92] or parsers [PP94], [Kon97], [BYM$^+$98] for the implementation language of the software system. Not only is parsing technology very brittle, it also introduces a language dependency in the tool which makes it hard to reuse across systems.

10.2 State of the Art

The analysis of code to identify copy and paste and plagiarism [Hal77], [Gri81], [Mad85], [Jan88] is broad. Various techniques are used: structural comparison using pattern matching [PP94], metrics [MLM96], [Kon97] or statistical analysis of the code, code fingerprints [Man94], [Joh93], [Joh94]. Here is a non exhaustive list of works covering the various techniques used. The table 10.1 summarizes the different approaches used. [Hal77, Gri81] detect student plagiarism using statistical comparisons of style characteristics such as the use of operators, use of special symbols, frequency of occurrences of references to variables or the order in which procedures are called. [Jan88] uses the static execution tree (the call graph) of a program to determine a fingerprint of the program.

In [PP94], a regular language is proposed to identify programming patterns. Cloning can be detected if we assume that if two code fragments can be generated by the same patterns then they could be clones. Johnson[Joh93] uses a specific heuristic, using constraints for the number of characters as well as the number of lines, to gather a number of lines a *snip* of source code on which he applies the fingerprint algorithm. sif [Man94] that is also based on the same idea. Johnson uses the identified duplication to understand change [Joh95]. [Kon97] evaluates the use of five data and control flow related metrics for identifying similar code fragments. The metrics are used as signatures for a code fragment. The technique supports change in the copied code. However it is not language independent because it is based on Abstract Syntax Tree annotation. dup [Bak92] is a program that detects parameterized matches and generates reports on the found matches. This work is however mostly focused on the algorithmic aspects of detecting parameterized duplication and not on the application of the technique in an actual software maintenance and reengineering context. [BYM+98] transforms source code into abstract syntax trees and detects clones and near miss clones among trees. It reports similar code sequences and proposes unifying macros to replace the found clones. Their approach requires, however, a full-fledged parser. See Table 10.1 for a categorization of a few approaches regardings these two steps.

A common process. All these approaches follow the following process:

Source code transformation. The source code is transformed into a intermediate representation. Depending the information needed, the operation can be at the lexical or syntactical level. For example, comments and other uninteresting code fragments can be removed from the code, or abstract syntax tree can be built from code.

Code Comparison. On the transformed code the actual comparison is done. It can be AST matching, line or metrics comparison.

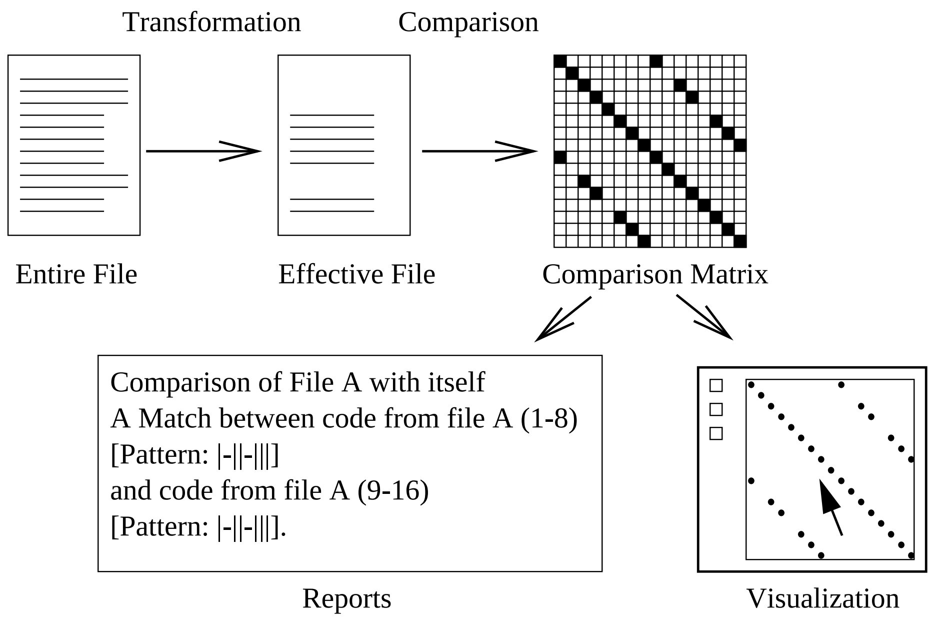
Author	**Level**	**Transformed Code**	**Comparison Technique**
[Joh94]	Lexical	Substrings	String-Matching
[Duc99]	Lexical	Normalized Strings	String-Matching
[Bak92]	Syntactical	Parameterized Strings	String-Matching
[MLM96]	Syntactical	Metric Tuples	Discrete comparison
[KDM+96]	Syntactical	Metric Tuples	Euclidean distance
[BYM+98]	Syntactical	AST	Tree- Matching

Table 10.1: Overview of the approaches for detecting duplication of code.

10.3 Our Approach: Language Independence

Most of the approaches [Joh94], [PP94], [Kon97], [BYM$^+$98] are based on parsing techniques and thus rely on having the *right* parser for the right dialect for *every* language that is used within an organization. Our approach developed in the context of the Famoos project by Matthias Rieger was to consider *simple strings matching* [RD98] [DRD99] [RD01b] [Rie01].

We detail the principles behind our approach under the three aspects *algorithms* used to compute the comparisons data, *visualization* of the comparison data, and *pattern matching* to condense the data. Figure 10.1 shows an overview of the steps that take us from source code to duplication data.



Comparison of File A with itself
A Match between code from file A (1-8)
[Pattern: |-||-|||]
and code from file A (9-16)
[Pattern: |-||-|||].

Figure 10.1: Overview of the approach.

10.3.1 Algorithmic Aspects

Clone detection is always a two-step process. First, source code is transformed into an internal format. Second, a more or less sophisticated comparison algorithm is then performed on the internal data. In our case, the code is transformed using string manipulation operations. To compare the transformed lines, we use basic string matching.

Transformation. We choose *one line* of source code as code fragment entity on which we base our algorithm. The choice is on the one hand motivated by the consideration that the important copy and paste performed by programmers include one or more lines, and on the other hand that prepocessing can be kept simple (see [Joh94] for an approach that uses multiple lines as fragment size).

The transformation we apply to a code fragment is minimal and stays in the realm of string manipulation: We remove comments and all white space until we get a condensed form of the line.As an example, the C line

```
if( code & pcObjType )  { /* print type */
```

is condensed to

```
if(code&pcObjType){
```

To stay language independent, we refrain from code transformation into more abstract formats like AST's [BYM+98] which have to employ parsing, or *parameterized strings* [Bak92] which need at least a lexer. As a consequence, the code reader which does the transformation is adapted to any new language in a few minutes.

The transformation reduces the entire file to an ordered collection of *effective* lines (see Figure 10.1) that will be compared against itself and line collections from other files.

Comparison Algorithm. Since we do not know what to look for, we cannot apply grep-like pattern matching algorithm but have to compare every entity (transformed source line) with every other entity. The comparison of two entities is done by string matching.

The result is a boolean `true` for an exact match and a `false` otherwise. This value is stored in a matrix (see Figure 10.1), taking the coordinates that the two compared entities have in their respective ordered collections as the matrix coordinates for the comparison result. Note that after the comparison process, the matrix only contains matches of individual lines and not yet of whole sequences. They will have to be extracted from the matrix in a separate pass (see section 10.3.3).

Optimization. The search space spanned by an input of n lines is quite uncomfortable ($\Omega(n^2)$). We thus reduce it by preprocessing the transformed lines a second time: the lines are hashed into B buckets. The string matching is then applied on all possible pairs of one hash bucket. Equal lines have the same hash value and are thus thrown into the same hash bucket, so no false negatives occur. This procedure cuts the processing time by the factor B (the same optimization is used in [BYM+98]).

10.3.2 Sequence Identification

The matrix created by the comparison is then analyzed to identify duplicated code.

To help the reader, the Figure 10.2 shows some typical situations that occurs. Each element is the matrix represents a line. As shown later in 10.5 such a matrix can be visualized using scatter-plots [Hel95]. Note however, that the shown matrixes are conceptual and can be completely decoupled from the visualization of duplicated code.

a) diagonals of dots indicate copied sequences of source code (see Figure 10.2 *a)*).

b) sequences that have holes in them indicate that a portion of a copied sequence has been changed (see Figure 10.2 *b)*).

c) broken sequences with lower parts shifted indicate that a new portion of code has been inserted (see Figure 10.2 *c), above* the main diagonal), or removed, respectively (Figure 10.2 *c), below* the main diagonal).

d) rectangular configurations indicate periodic occurrences of the same code (see Figure 10.2 *d)*). An example is the `break;` at the end of the individual `case`s in a C/C++ `switch` statement or recurring preprocessor commands.

Note that due to the line-based comparison of C switch statements, repeated matches of either structural code elements that occur alone on one line (e.g., the `break;`) or minimal "idioms" (e.g., the

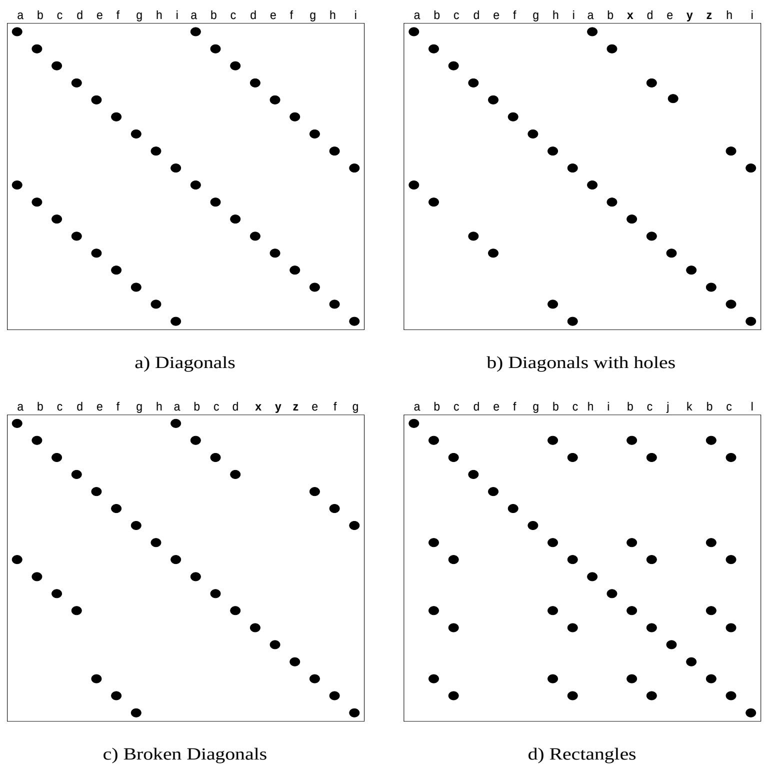

a b c d e f g h i a b c d e f g h i
a b c d e f g h i a b x d e y z h i
a) Diagonals
b) Diagonals with holes
a b c d e f g h a b c d x y z e f g
a b c d e f g b c h i b c j k b c l
c) Broken Diagonals
d) Rectangles

Figure 10.2: Different Configurations of Dots.

frequent C-line `int` i;) spread spurious dots all over the matrix. To get rid of this *noise* rather than duplication, we have two possibilities: (a) Remove such lines up front by running a filter over the input before the comparison process. (b) Use a pattern matcher to sweep over the matrix and remove single dots. We do both.

10.3.3 Pattern Matching to Extract Copied Sequences

The algorithm as stated above does not catch duplicated code that was changed inside one line of code. In a sequence of copied code that is compared with the original sequence, a changed line shows up as a hole in the diagonal match pattern (see Figure 10.2 *b)*). To cope for this weakness when extracting whole copied sequences, a pattern matcher is run over the matrix which captures diagonal lines and allows holes up to a certain size in the middle of the line. We are evaluating the impact of such a pattern matcher on the quality of the identified duplicated code (see Section 10.6).

10.4 Some Empirical Experiments

The goal of our case studies is to stress the language independent aspects and to prove the potential of our approach. In order to choose the case studies we took four criteria into account: first the *implementation language*, second the *potential of duplication*, third the *size* of the system and fourth the possibility of *reproduction of the experiment* by other researchers.

Implementation Languages. We selected languages that have clearly different syntaxes: C, Smalltalk,

Average percentage of duplication found per file				
Case	gcc	Database S.	Payroll	Message B.
effective LOC	8.7%	36.4%	59.3%	29.4%
entire LOC	5.9%	23.3%	25.4%	17.4%
# of files with duplication	143	464	13	24
Total # of Files	170	593	13	36

Table 10.2: Some selected case studies for duplication identification.

Python and Cobol. Smalltalk is well-known to have a simple and uniform, keyword-based syntax. The syntax of Python is based on indentation which replaces block delimiters. In the overly verbose Cobol syntax, line numbers exist and identifiers that are attached at the end of each line. It is obvious that writing a parser for these diverse languages would be a entire new endeavor in each case [RPR93].

Potential Duplication. We used case studies from different sources to maximize the potential range of the duplication. We took (a) two industrial case studies for which it was known that they contained a lot of duplication, (b) one case study where the duplication of code was suspected to be low, and (c) a small application from the public domain for which we had no knowledge about the duplication situation.

Size. The scalability of our approach has to be considered under two aspects: First, does it scale given the size of the source code? Second, does it scale given the amount of duplication that is found? To prove that our approach is scalable under the first aspect, we took the full GNU GCC source code whose size is 13Mb. That our approach also scales regarding the second aspect was proved when one of the industrial applications happened to contain an enormous amount of duplication.

Reproducibility of the Experiment. We chose one well-known and freely available case study: the Free Software Foundation C compiler gcc[1] to make our experiments reproducible by others.

The Case Study Material. The chosen case studies are: the implementation files of the GNU GCC source (written in C), a web-based message board (Python), parts of a payroll application (Cobol) and a database server (Smalltalk). The statistics of the case studies are given in the table 10.2. Note that in the database server case one file contained one class.

Case	*Language*	*Size*	*# Files*	*LOC*
gcc	C	13.4 Mb	221	460000
Database Server	Smalltalk	7.1 Mb	593	245000
Payroll	Cobol	3 Mb	13	40000
Message Board	Python	265 K	36	6500

These overall percentages show the relevance of the identification of duplicated code. We have been successful in identifying duplicated code. The next question to be addressed is how such duplicated code can cured. We started to address this problem in the context of object-oriented programming language as discussed in next section.

[1]ftp://prepr.ai.mit.edu/pub/gnu/

10.5 Supporting Understanding and Reporting Duplication

Besides detecting duplicated code, supporting understanding and reporting duplicated code of the reengineer is also necessary. This is especially true in the context of legacy systems that are commonly huge and where reengineers can be flooded by information.

We used three approaches to help the understanding of duplicated code: (1) the visualization of duplicated code [DRD99], [Mal99], (2) the generation of textual report, and (3) the synthesis of duplication percentages.

10.5.1 Support for Code Duplication Understanding

The matrix created by the comparison can be visualized using scatter-plots [Hel95], which were first used by geneticists looking for similar strings of DNA. Such “dot drawings” allow one to add another perception on the duplicated code. Here are some examples extracted from our experimental validations. Such examples show the added-value provided by the visualization of the duplicated code. The visualization supports the understanding of the duplication within the context of an entire system. It describes the occurrences within a specific context.

Figure 10.4 presents two versions of apparently the same file. It is immediately apparent from the picture, where and how much of the code has been changed. In Figure 10.5 we see five classes that are identical copies (except class E which exhibits some minor changes). The spurious dots and rectangular patterns found in the plots act as a kind of visual fingerprint, which would help to find the members of this “club” in a larger matrix even if they were not clustered together like in Figure 10.5.

10.5.2 Textual Reports

To support the possibility to access quickly the information of the duplicated code fragments found in a system, we work on the elaboration of ascii reports that can be analyzed using perl-scripts.

The report extract shown below presents matched sequences that were found in two comparisons. First, the participants of the comparison are identified. Then, a summary of all the sequences found is presented. Finally, all matched sequences are listed in detail. In the `Pattern`-string of a match, a vertical bar stands for two lines that matched, and a horizontal bar marks two lines that did not match. A dot in the pattern string represents a source line that contained comments or white space and was removed from the input before the comparison. Such lines are not taken into account during the pattern matching and are printed so that reported source line numbers correspond to the length of the pattern.

```
Comparison of
~/gcc/gcc-2.8.1/cp/pt.c
with itself

Sequence length: 13 Number of Sequences: 1

A Match between code from file ’pt.c’
(from line 2603 to line 2615)  [Pattern: ||-||-||-||||]
and code from file ’pt.c’
(from line 68 to line 80)  [Pattern: ||-||-||-||||].
(Stretch = 13 , RelevantLines = 10)

Comparison of
~/gcc/gcc-2.8.1/config/i386/i386.c
```

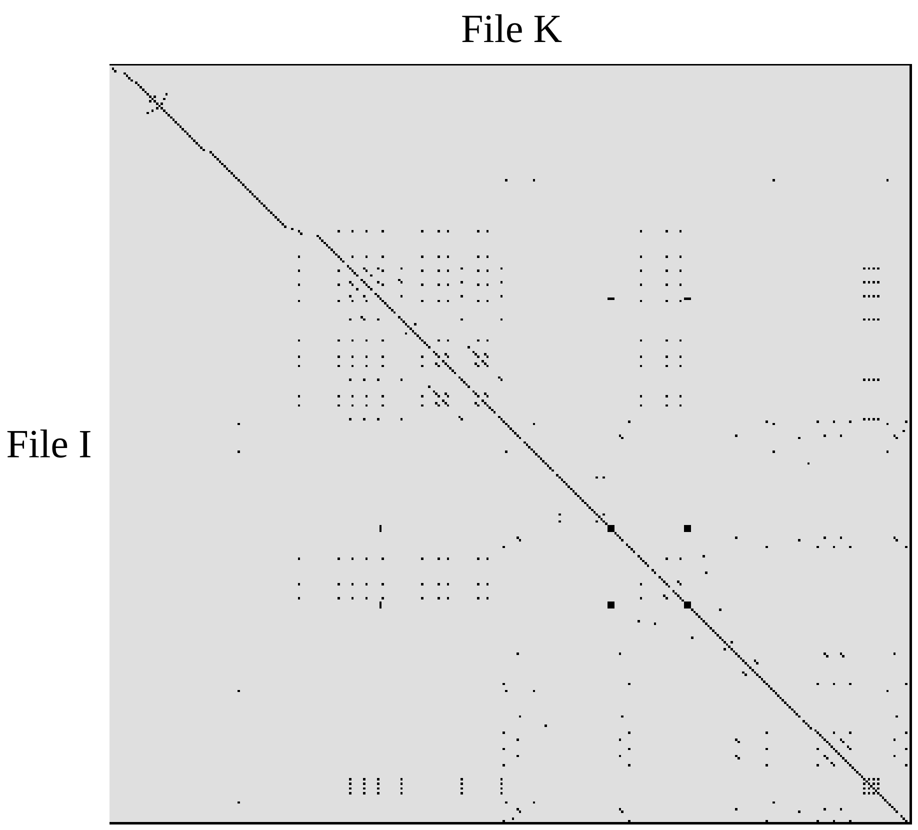


Figure 10.3: Two Cobol files. Although the diagonal suggests a comparison of a file with itself, the gaps show that two different files have been compared.

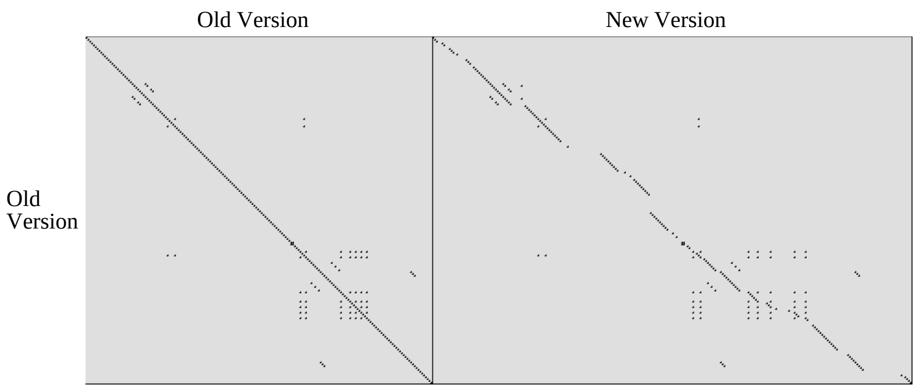


Figure 10.4: Two versions of a Python file. In the left rectangle, the old file is compared with itself, on the right the old file is compared with the new version.

```
with
~/gcc/gcc-2.8.1/config/h8300/h8300.c

Sequence length: 17 Number of Sequences: 1
```

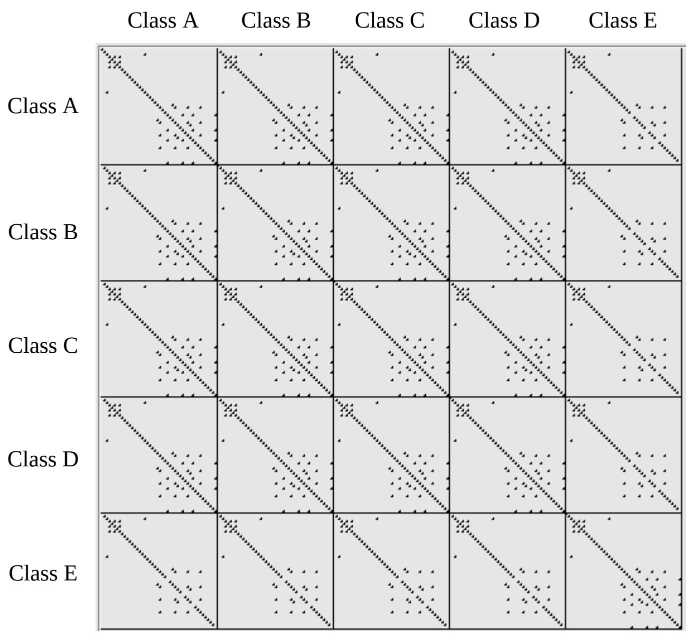


Figure 10.5: Five sibling classes in a database server application. The five rectangles containing the main diagonal represent the comparisons of each class with itself.

```
A Match between code from file ’i386.c’
(from line 3060 to line 3092)
[Pattern: |-..|-..|-..|-..|-..|-..|-..|-..|]
and code from file ’h8300.c’
(from line 1071 to line 1087)
[Pattern: |-|-|-|-|-|-|-|-|].
(Stretch = 17 , RelevantLines = 9)
```

Reports are important since they provide the exact location of the duplicated code. This information is handy for the maintainer that has to work with the code. Even, if it lacks the appeal of visualization, it is convenient and easy to access information.

10.5.3 Synthesis of Code Duplication

Having a global percentage of duplication per application or per file is the first indication of the state of an application. However, this kind of information is quite raw. That’s why in addition, we propose to use matrixes showing a file per file the duplication percentage. We performed some experiences to evaluate how such an overview should be stated to be useful and extracted some results to illustrate our approach [RD01b].

Overview of the Duplication

The table 10.2 presents the average percentage of duplication per file. We also include the percentage in terms of entire code (i.e., files including comments) so that readers can have their own ideas about the relevance of the duplication detection. The third line shows the number of files that effectively contain duplicated code under the following constraints.

First, to remove accidental duplication of small fragments, we limited the detection to sequences of lines having 10 or more lines. Second, to avoid missing duplicated sequences with changed parts, we allowed holes in the copied sequences up to 20% of the total length of the sequence. Third, since reengineers are looking for the duplication of *functional* code elements, we chose to present the percentage of duplication in terms of *effective* lines of code. This means that we computed the percentage over the set of lines from which comments and white space have been removed (see section 10.3.1). This way we minimize the impact of comments in the percentage computation. As a consequence, a file can be integrally copied into a second, but if this second file contains a lot of comments the percentage will not reflect this situation.

In the table 10.2, note that inferior percentages for the entire code is normal because comments and white space can make up for a lot of lines.

The quite high average percentage found for the two industrial case studies (Cobol payroll system and database server) is not totally surprising considering fact that these were given to us because it was well known that they contained a lot of duplication. Nevertheless we were astounded by their overall duplication ratio. The web message board system shows some duplication elements that are result from evolutionary *clones*, since the system was given to us as a snapshot in the middle of an extension, thus containing old as well as new code side by side. The GNU GCC source code has the lowest ratio.

Percentage per File

Now we refine our overall analysis by looking at the duplication percentage per file. We present the payroll system and GNU GCC because they cover the extremes in the range of our case studies. Note that the tables in Figures 10.6 and 10.7 only display the files containing the effective duplication.

Percentage per File: the Payroll Case. For the payroll system, the overview (Figure 10.6) immediately identifies three main groups according to the degree of duplication: (a) few duplication (around 5% in file F), (2) some duplication (from 25% to 50% in files A, B, D, E and J) and (3) mostly duplicated (up to 70% in files C, G, H, I, K, L and M).

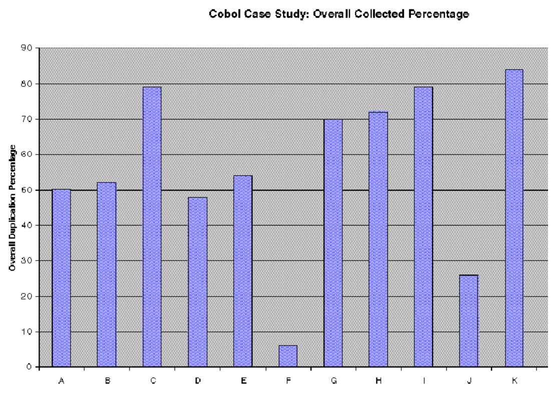


Figure 10.6: Duplication Percentage per Files in the payroll case study.

Percentage per File: the GNU GCC **Case.** Even if the average percentage showed that GNU GCC has one of the lowest percentage of duplication, looking at the percentage per file (Figure 10.7) gives another view. We see that two files have more than 60% of duplication (actually it is actually generated code), that 6 files have more than 50% of duplication and that a number of files have more than 20% of duplication.

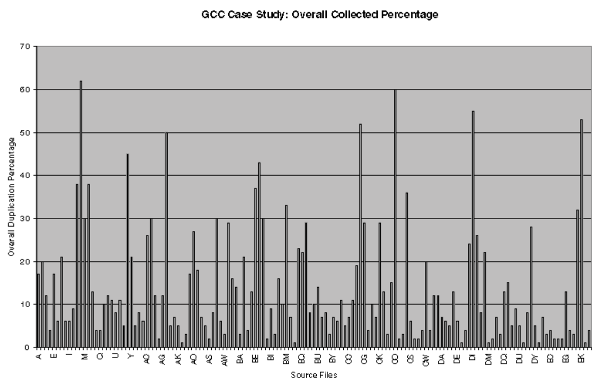


Figure 10.7: Duplication Percentage per Files in the GNU GCC case study.

File per File Comparison.

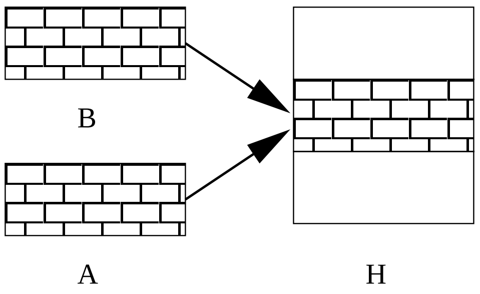
What is definitively missing with the global overviews is to qualify somehow the duplication relationship that exist between files, i.e., to know which file contains code of which other files. To answer such a question we extracted from the generated report a file per file duplication percentage, i.e., for each file we computed the percentage of code that is also found in another file.

Payroll Case. We obtained the following table. This table expresses the fact that a certain percentage of one file has been found in another one. For example 84% of K are also found into I.

	I	H	K	M	C	G	B	E	J	L	A	D
I		12	**79**	3	3	15						
H	7		9	2	2	**72**						
K	**84**	16		3	3	16						
M	2	2	2	15	**77**	2						
C	2	2	2	**79**	*16*	2						
G	9	**70**	9	2	2							
B							8	39	23	44	32	19
E							**30**	9	**19**	**47**	**30**	**19**
J							15	16	5	15	14	14
L							**45**	**62**	**24**	7	**47**	**26**
A							23	28	15	33	8	31
D							17	22	18	22	37	8

	I	H	K	F	M	C	G	B	E	J	L	A	D
I		12	**79**		3	3	15						
H	7		9		2	2	**72**						
K	**84**	16			3	3	16						
M	2	2	2		15	**77**	2						
C	2	2	2		**79**	*16*	2						
G	9	**70**	9		2	2							
B				1				8	39	23	44	32	19
E				1				**30**	9	**19**	**47**	**30**	**19**
J				1				15	16	5	15	14	14
L				1				**45**	**62**	**24**	7	**47**	**26**
A				1				23	28	15	33	8	31
D				1				17	22	18	22	37	8

How to Interpret such Matrix? Looking at such a matrix questions arise like why the total number of percentage is not 100%. The Figure 10.8 shows that given three files A, B and H, where A and B are the same and has been integrally copied into H, and H is three times bigger than A, we obtain the companion matrix expressing that 100% of A is also found in H and that 30% of H are found into A and B.



	H	A	B
A	100%		100%
B	100%	100%	
H		30%	30%

Figure 10.8: Help for the interpretation of the file per file matrix.

File per File Matrix Analysis. Interpreting the matrix we obtained the following information:

1. **Few file internal duplication.** When we look at the diagonal that represents a file compared with itself, we see that the percentage of duplication is not high. The maximum is only 16% for the file C. We conclude that in this case study the duplicated code really occurs between different files.

   Such information is important in particular in the case of object-oriented languages that represent one class per file such as C++. Indeed the presence of file-internal duplication suggest a refactorization of the class, which could result in a number of private methods.

2. **Outsider.** The fourth line concerning the file F definitively presents a file that does not contain duplicated code.

3. **Clusters of related files.** The shape of the matrix gives us that two main clusters of files having duplication relationships exist. The files I, H, K, M, C and G form a first cluster, whereas the files B, E, J, L, A and D from a second cluster.

4. **Granularity of the duplication.** In the first group, a file is primary composed by a big piece of duplicated code e.g., the 79% of the file I are also found in file K, 70% of G are also found into H. In the second group the granularity of the duplicated functionality is smaller. For example, 30%, 47% and 30% of file E are respectively found in file B, L and A.

5. **Cross cutting overviews.** As shown in 10.5.3 the overview presented three main groups of files depending on their percentage of duplication. However, these groups were just high level views of the duplication and they do not represent groups based on *duplication relationship*.

   The file per file matrix solves this problem. For example, the file L that belongs to the group of the files heavily duplicated with an overall duplication of 75% (see Figure 10.6) belongs in reality to the second cluster of files and is the result of multiple duplications (45%, 62%, 24%, 47% and 26%).

10.6 Ongoing Work

We are currently assessing the amount of duplicated code that we might lose in case of renaming [RD01b]. For that we are performing empirical analysis comparing duplication identified using our simple line matching algorithm and parametrized matched approaches.

Our approach to minimize renaming is to allow holes inside a sequence of duplicated lines. Two criterion are important to consider: first the density of holes allowed in a sequence of lines and second the number of consecutive holes in a sequence. We are currently analyzing the impact of these two parameters on the resulting identified code.

10.7 Future Tracks

Reporting Duplicated Code. One of the major remark that we can do while reading the duplicated code identification literature is the poorness of the way the information is presented. Most of the time the identified duplication is displayed in overall percentage of the global application under study. While such an information is relevant it contains a qualitative information. What is definitively needed is a means to sort duplicated fragments in terms of, for example, the occurrence or the number of files impacted. However, such an approach is not trivial to implement due to the fact that code segments can overlap, be contained, can contains holes and skipped lines.

10.8 Contributions

Implementations. Under our supervision, Matthias Rieger developed during his PhD a tool called Duploc which is freely available at: http://www.iam.unibe.ch/∼pure/.

DUPLOC supports graphical representation of duplication and a mural information system developed by P. Malorgio during a project under our supervision.

Bibliography.

[Mal99] P. Malorgio An Information Mural Visualization for Duploc University of Berne, 1999, http://www.iam.unibe.ch/∼scg/Archive/Projects/malorgio.pdf.

[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)*, pages 109–118. IEEE, September 1999.

[RD98] M. Rieger and S. Ducasse. Visual Detection of Duplicated Code In Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543, 75-76, Springer-Verlag, 1998.

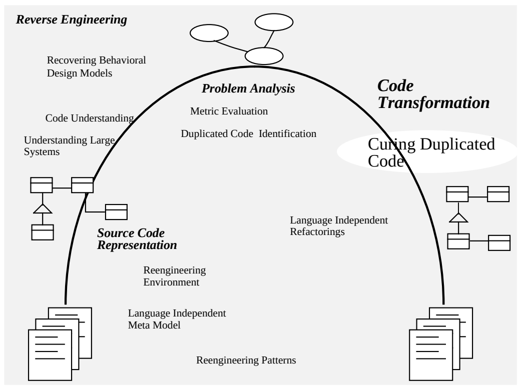
[Rie01] M. Rieger *Duplicated Code Identification (temporary title)* PhD Thesis of the University of Berne.

[RD01b] M. Rieger and S. Ducasse Language Independent Duplicated Code Identification. Working Paper.

Chapter 11

Supporting Object-Oriented Duplicated Code Cure

In the previous chapter we showed how duplicated code can be identified in any language. Now we address the problem of how to handle such a duplicated code in the particular context of object-oriented programming languages. Our approach is based on the categorization of duplicated code in specific situations that we call *scenario* and the interaction with the reengineer to steer refactorings. Hence it is at the border between the problem analysis and code transformation areas as shown by the following Figure.


Reverse Engineering
Recovering Behavioral Design Models
Code Understanding
Understanding Large Systems
Problem Analysis
Metric Evaluation
Duplicated Code Identification
Code Transformation
Curing Duplicated Code
Source Code Representation
Language Independent Refactorings
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

11.1 State of the Art

Techniques already exist that propose automatic reorganization of class hierarchies [Cas91] [Cas92] [Moo96]. These techniques are proven to be scientifically sound, however we believe that they are not applicable on real industrial settings that have to be maintained by *human*. Indeed, for example, [BMD+99] introduces automatically Strategy design patterns with artificially generated names and [Moo96] splits and recomputes hierarchies resulting in classes containing methods with randomly created names.

Refactorings [Opd92, Rob99, FBB+99] enable the developers to transform code in a behavior preserving way. Even if they are extremely important functionality to improve code, up to now there is no tool that supports the analysis of duplicated code and allows the application of refactorings. In parallel to our research [BMD+00] proposes an analysis of the duplicated code whose intention is to support refactoring application.

From the qualitative evaluation of duplicated code we made [GKN01] we found that eliminating non trivial duplication cannot be made automatically, subtle design decisions and code reorganization have to be taken into account. In case of partial method duplication, the tradeoff between generalization of the code and ease to use it is also the design appreciation that have to be in the hand of the developer.

We would like to stress the following limits of a fully automated approach:

- Class explosion. We want to limit introduction of new classes that do not carry symbolic information for the programmer. While automatically generated classes can be an interesting approach to optimize hierarchies in the perspective of deployment and compiled applications, the code that is further developed should not include classes or methods that have not been consciously written.
- Do not add to the burden! Legacy code tends to suffer from a lack of cleaning. Moreover, any system not carefully maintained tends to decay. That is why new symbols without human controlled semantics should not be added into the code.
- A fast edit compile loop. The approach should be integrated into the incremental edit-compilation loop. Techniques requiring complete split and reorganization of hierarchies are thus out of question.

11.2 Steering Refactorings

Our approach is based on the categorization of duplicated code into specific situations that we call *scenarios* and the interaction with the reengineer to steer refactorings. SUPREMO, the prototype we built, analyzes the duplication, classifies it into scenarios, qualifies it in term of certain attributes of the duplication (e.g., partial or complete, length of duplication, or/and density). Then the information is presented to the developer who can see the duplication in the context of the system (see Figures 11.4 and11.5) and change some parameter of the duplication identification like the length or the density of the duplication.

Depending on the relationship between classes containing the methods where the duplicated code is found, we define different scenarios that characterize the situation and allow the definition of possible cures. The list of scenarios is: In the Same Method, In the Same Class, With a Sibling Class, With a Superclass, With an Ancestor, In Common Hierarchy, With a First Cousin, In Common Hierarchy, and Between Unrelated Classes. Each of these scenarios is linked with a set of possible refactorings that would lead to reduce the duplication. [GKN01] describes scenarios in details. We present the With a Sibling Class scenario with an example.

11.2.1 Duplication With a Sibling Class Scenario

By sibling classes we refer to all classes with the same direct superclass and with the same hierarchical level (see Figure 11.1). The highlighted rectangles represent the classes in which the methods containing duplicated code were found.

The following refactorings may help to cure this scenario: *Pull Up Method*, *Parameterization*, *Extract Method*, *Substitute Algorithm*, *Form Template Method*, *Replace Subclass with Field*, and *Extract Superclass*.

We illustrate the *Form Template Method* refactoring with the example shown in Figure 11.2. It shows partial duplication between two sibling classes, which is a good candidate for the creation of

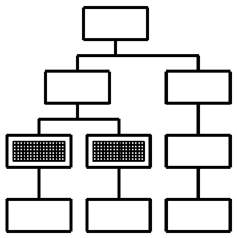


Figure 11.1: Duplication between sibling classes.

a template method: both methods in subclasses perform similar steps in the same order, yet the steps are different. Hook methods are created then the methods are moved up in the superclass as shown by Figure 11.3.

11.3 SUPREMO

SUPREMO allows one to select, identify duplicated code and apply refactorings. It supports the specification of parameters to filter duplicated information: the length of the duplication and its density, i.e., number of lines duplicated over total number of lines of a code segment . Such parameters depend on the language used, and the organization (coding style). In the empirical validation we performed we found that 0.6% of density was a good compromise between too much noise and missing template like duplication that could be generalized [GKN01].

SUPREMO is composed by two main tools: the *code segment analyzer* and the *global viewer*. SUPREMO manipulates *duplication* elements. A duplication element is composed by two duplicated code segments that represent analyzed method codes. A duplication element contains the following information : the quality of the duplication (partial or complete), the classes that contained the duplicated code segments, the scenario in which it belongs.
To create a duplication element, the code segment analyzer performs the following tasks:

1. Different duplicated code segments belonging to the same method are regrouped. Indeed, DUPLOC been purely text-based, information about the object-oriented structure has to be reconstructed.

2. Duplication information is reduced. All transitive duplication elements are removed to keep only a canonical representation, e.g., if A is copied with B and B with C, DUPLOC reports that A is also duplicated with C.

3. The object-oriented location of the duplicated segment code are then used to derive the scenario in which the duplication element belongs to.

The Figure 11.4 presents the code segment analyzer user interface that displays a duplicated occurrence.

11.3.1 Code Segment Analyzer

The code segment analyzer is the tool that analyses the duplication information given by DUPLOC and proposes to the reengineer scenario (See Figure 11.4).

```
MSEImporter>>import
     "self importMyself"

     | class |
     class := [
        self startUIProgressFeedback: 'Import (by querying Smalltalk
         repository)' maxProgressMeasure: smalltalkClasses size.
     classCounter := 0.
     smalltalkClasses
        collect:
          [:clss |
            classCounter := classCounter + 1.
            self nextUIProgressFeedback: clss name
               progressMeasure: classCounter.
   self perform: classCreatorMessage with: clss class with: true.
   self perform: classCreatorMessage with: clss with: false]]
    valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
    self optimize.
    ^class
```

```
 MSEVisualWorksImporter>>import
     "self importMyself"

     | classEntities |
     classEntities := [
        self startUIProgressFeedback: 'Import (by parsing
         method-sources)' maxProgressMeasure: smalltalkClasses size.
     classCounter := 0.
     smalltalkClasses
        collect:
          [:clss |
            classCounter := classCounter + 1.
            self nextUIProgressFeedback: clss name
            progressMeasure: classCounter.
    self buildEntitiesFromSmalltalkClass: clss ]]
    valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
    self optimize.
    ^classEntities
```

Figure 11.2: Duplication between sibling classes where the application of the refactoring Form Template Method would help removing the duplication.

We briefly present its main constituents:

- In the top pane the duplicated code segment is qualified. The following information is presented: the scenario in which the duplication occurs, the quality of the duplication (complete or partial), the number of impacted classes (i.e., the classes in which the same duplication occurs).
- Global information is displayed such as the minimal number of matched lines considered for a duplicated segment or the density of the duplication. By changing the values of these global parameters the reengineer can reduce the presented duplication.
- Simple navigation between the duplicated code segments is provided.
- The two code segments composing a duplicated code segment are displayed with their respective duplicated lines in bold and class and method name. The refactoring browser can be invoked from this pane.

```
MSEAbstractImporter>>import
    "self importMyself"

    | classEntities |
    classEntities := [
       self startUIProgressFeedback: self myComment
            maxProgressMeasure: smalltalkClasses size.
    classCounter := 0.
    smalltalkClasses
       collect:
         [:clss |
           classCounter := classCounter + 1.
           self nextUIProgressFeedback: clss name
           progressMeasure: classCounter.
   self buildEntitiesFrom: clss ]]
   valueNowOrOnUnwindDo: [ self terminateUIProgressFeedback].
   self optimize.
   ^classEntities
```

Figure 11.3: Template method resulting from the application of the refactoring Form Template Method.

11.3.2 The global viewer

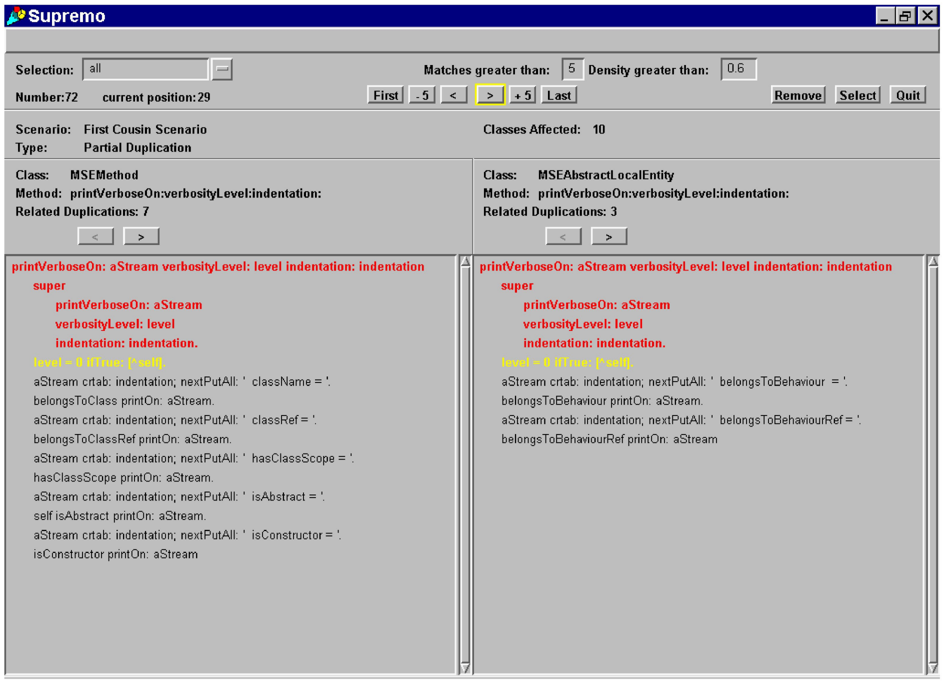
While the code segment analyzer presents information relative to an actual duplication in a local manner, i.e., two methods containing a certain amount of duplicated lines, the global viewer presents the information at the application level.

The global viewer acts relatively to the selected duplication element analyzed in the code segment analyzer. It presents how the duplicated code segment currently analyzed by the code segment analyzer is duplicated in all the application. The global viewer presents the inheritance tree of all classes in the analyzed application. The color of the nodes conveys the following semantics:

- Green means that the class does not contain duplicated code that belongs to the currently analyzed code segment.
- Yellow means that the class is one of the two classes containing the currently analyzed code segment (long arrows in the Figure).
- Blue means that the class is impacted by the duplication element, i.e., it contains a piece of code that has a duplication relationship with the current duplication element (short arrows in the Figure).

The Figure 11.5 shows a duplication with a first cousin scenario. The current duplication has an impact of 10 (see Figure 11.4). The two classes pointed by the big arrows are those containing the methods with the current duplication. Both are colored yellow. The eight other classes are colored blue. They are pointed to by the small arrows in the Figure 11.5. In the presented example, one cannot envisage refactoring the current duplication without considering the involved duplications in the other eight classes. The hierarchical presentation definitively helps to find out which class is the common ancestor of the all 10 classes.

The global viewer provides a useful information that is used during the refactoring application or cure of the duplication.



Selection: all

Matches greater than: 5 Density greater than: 0.6

Number:72 current position:29

Scenario: First Cousin Scenario
Type: Partial Duplication

Classes Affected: 10

Class: MSEMethod
Method: printVerboseOn:verbosityLevel:indentation:
Related Duplications: 7

```
printVerboseOn: aStream verbosityLevel: level indentation: indentation
    super
        printVerboseOn: aStream
        verbosityLevel: level
        indentation: indentation.
    level = 0 ifTrue: [^self].
    aStream crtab: indentation; nextPutAll: ' className = '.
    belongsToClass printOn: aStream.
    aStream crtab: indentation; nextPutAll: ' classRef = '.
    belongsToClassRef printOn: aStream.
    aStream crtab: indentation; nextPutAll: ' hasClassScope = '.
    hasClassScope printOn: aStream.
    aStream crtab: indentation; nextPutAll: ' isAbstract = '.
    self isAbstract printOn: aStream.
    aStream crtab: indentation; nextPutAll: ' isConstructor = '.
    isConstructor printOn: aStream
```

Class: MSEAbstractLocalEntity
Method: printVerboseOn:verbosityLevel:indentation:
Related Duplications: 3

```
printVerboseOn: aStream verbosityLevel: level indentation: indentation
    super
        printVerboseOn: aStream
        verbosityLevel: level
        indentation: indentation.
    level = 0 ifTrue: [^self].
    aStream crtab: indentation; nextPutAll: ' belongsToBehaviour = '.
    belongsToBehaviour printOn: aStream.
    aStream crtab: indentation; nextPutAll: ' belongsToBehaviourRef = '.
    belongsToBehaviourRef printOn: aStream
```

Figure 11.4: The code segment analyzer showing two occurrences of a duplicated code.

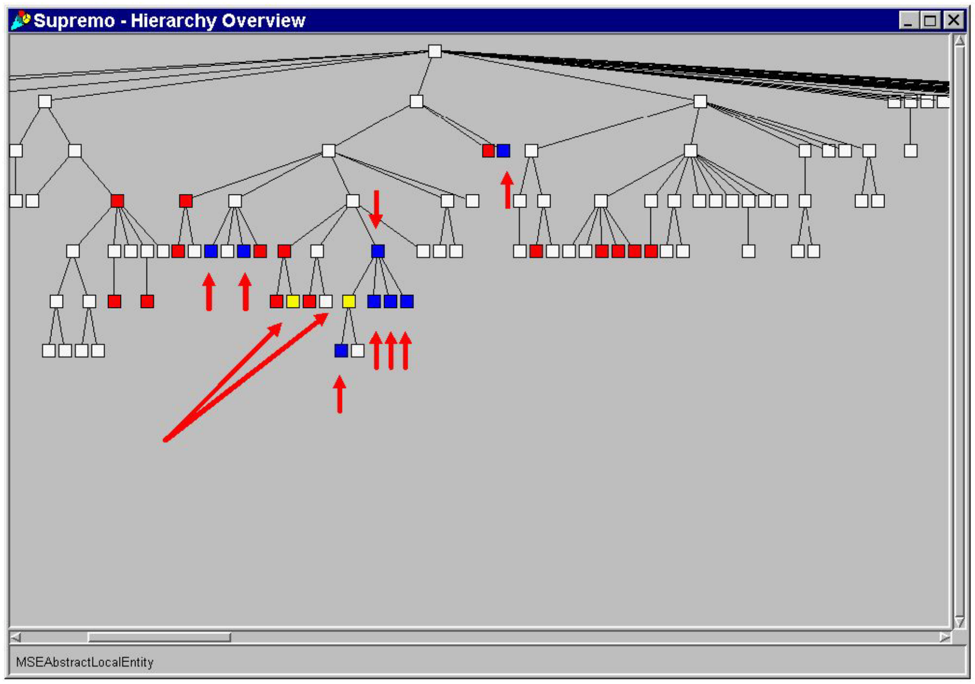


Figure 11.5: Duplication in context: the colored nodes of the inheritance tree represent class containing duplication: the currently identified clones shown in the segment analyzer are in yellow (long arrows) while the other duplicated occurrences are in blue (short arrows).

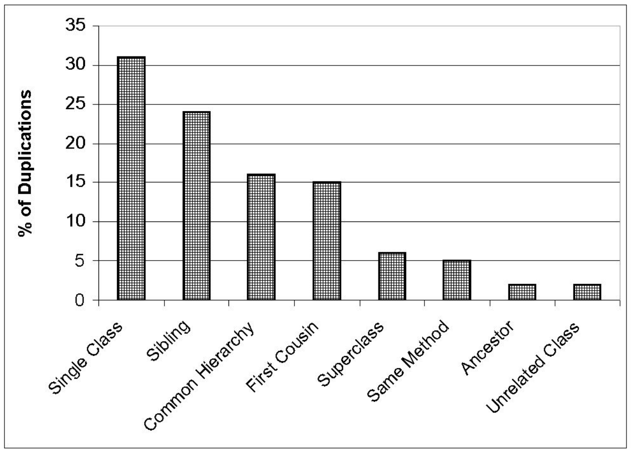


Figure 11.6: Distribution of the scenario is the case studies.

11.4 Lessons Learnt

Duplication refactoring is to a large extent depending on the application, the language used, and the organization context. We performed some empirical validation to assess the relevance of the scenario on different case studies as shown in the table 11.1. Note that the Java and C++ case have been chosen to validate the language independent aspect of the approach. The case studies are: (1) software that was developed by our group. It was used for the qualitative evaluation of the identified duplication, (2) software that we expect not containing significant duplication, (3) an industrial application.

Application	# Classes	# Methods	LOC
CODECRAWLER 2.912	82	1552	9745
DUPLOC 2.14g	269	4768	36526
MOOSE 1.45	254	4592	36566
VisualWorks 3.0	883	30689	278347
Refactoring Browser 3.5	238	5693	39825
PDP 2.6	62	4147	45289
MAF	269	6076	55251
Jpeg (C++)	14	145	2954
Swing (Java)	30	387	6101

Table 11.1: The applications used for our empirical validation.

At the moment is difficult to draw general conclusions, however the diagram 11.6 illustrates the distribution we found. Analyzing it shows that in the case studies we took (i.e., we knew that really few trivial duplicated)

The use of SUPREMO shows an unexpected result. The global view that has been originally developed by G. Golomingi as a view to help him debugging SUPREMO, proved to be an important tool to help developer analyzing duplication.

Scenarios	CodeCrawler 2.912	Duploc 2.14g	Moose 1.45	VisualWorks 3.0	RefactoringBrowser 3.5	PDPApp 2.6	MAF
With an Ancestor Scenario	1	1	1	18	2	0	1
In Common Hierarchy Scenario	1	2	26	15	6	0	0
With a First Cousin Cousin Scenario	11	1	14	17	2	0	4
In the Same Class Scenario	11	25	18	202	26	7	33
In the Same Method Scenario	0	6	2	24	3	0	2
With a Sibling Class Scenario	9	26	8	27	17	0	3
With a Superclass Scenario	3	4	3	42	4	6	5
Between Unrelated Classes Scenario	2	1	0	44	2	0	24
Total	38	66	72	389	62	72	13

Table 11.2: Number of Duplications in Analyzed applications.

11.5 Future Tracks

The analysis performed by SUPREMO is limited by the constraint to stay language independent. As SUPREMO is based on the FAMIX model, it does not manipulate AST. Proposing finer analysis for one dedicated language is definitively interesting because it would allow us to more precisely characterize the nature of the duplication.

The integration of tools like DUPLOC and SUPREMO with the Refactoring Browser is really promising. The current integration is not complete and the process for analyzing duplicated code could be enhanced from a developer point of view. It would be interesting to see how code duplication detection could be integrated in a development environment. In particular, improving the overview functionality of SUPREMO to provide more semantical information about the duplication is definitively appealing.

11.6 Contributions

Implementations. Under our supervision, a tools called SUPREMO have been developed by G. Golomingi during his master [GKN01]. SUPREMO analyses the duplication of object-oriented code. It provides a conceptual framework to understand the duplication and refactoring it.

[GKN01] G. Golomingi Koni-N Sapu A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems Master Thesis of the University of Berne, 2001 http://www.iam.-unibe.ch/~scg/Archive/Diploma/golomingi.pdf.

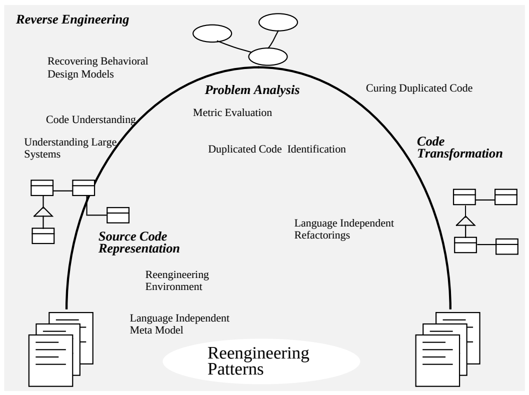
[DRG99] , S. Ducasse, M. Rieger and G. Golomingi Tool Support for Refactoring Duplicated OO Code In Proceedings of the *ECOOP'99 Workshop on Experiences in Object-Oriented Re-Engineering*, 1999.

[DRG01] S. Ducasse, M. Rieger and G. Golomingi A Scenario based Approach for Steering Duplicated Code Refactorings, Working paper.

Chapter 12

Patterns: Transferring Unit of Expertise

The results we presented in the previous chapter cover different aspects of a reengineering effort. What was missing is a means to identify, record and communicate object-oriented reengineering in a more abstract way. We worked on reengineering patterns which do not belongs to the areas we identified in the first chapter as they represent a kind of meta-information.


Reverse Engineering
Recovering Behavioral Design Models
Problem Analysis
Curing Duplicated Code
Code Understanding
Metric Evaluation
Understanding Large Systems
Duplicated Code Identification
Code Transformation
Language Independent Refactorings
Source Code Representation
Reengineering Environment
Language Independent Meta Model
Reengineering Patterns

In this chapter, we elaborate why reengineering patterns are important to record unit of knowledge in the context of reengineering. We discuss the role of reengineering patterns and contrast them with related kinds of patterns. We then highlight the form of reengineering patterns we elaborated and present one simple pattern for type-check elimination as an example. Note that this example is not formatted and edited as it is in [DDN02].

12.1 The Need for Reengineering Patterns

Reengineering projects, despite their diversity, often encounter some typical problems again and again. These can be problems at different levels, e.g., process, reverse engineering or reengineering, and due to different practices [FY00]. However, it is unlikely that one methodology or process will be appropriate for all projects and organizations [SP98], just as no one tool or technique can be expected to solve all the technical problems encountered in a reengineering project.

To allow reengineering projects to benefit from the experience gained in previous efforts, it is therefore essential to have a way to record and communicate expertise at different levels: from knowledge about how to approach a system to be reengineered, to knowledge about improving code by eliminating legacy problems. Moreover, an appropriate form is required for transferring expertise.

This form should be small enough to be easily consulted and navigated, and stable enough as to be useful for many reengineering projects.

12.2 Reengineering Patterns.

In the object-oriented community Design Patterns [GHJV95], [BMR$^+$96] have been adopted as an effective way of communicating expertise about software design. A design pattern describes a solution for a recurring design problem in a form which facilitates the reuse of a proven design solution. In addition to the technical description of the solution, an important element of a design pattern is its discussion of the advantages and disadvantages of applying the pattern.

Following the Design Pattern community's intent, we used a pattern form to transfer expertise in the area of reengineering. Reengineering patterns are stable units of expertise which can be consulted in any reengineering effort: they describe a process without proposing a complete methodology. A more thorough discussion of the advantages of the pattern form as a vehicle for reengineering expertise can be found in [SP98], which discusses patterns closely related to ours.

12.2.1 Three kinds of patterns

We distinguish three different sort of reengineering patterns. A detailed description of the forms we used can be found in [DDN02].

Process Patterns. Process reengineering patterns are patterns that focus on migration strategies like wrapping legacy code, and issues concerning risk minimisation. Examples of process reengineering patterns are Wrapper, Bridge to the new Town, LittleChicken [BS95], Buggiest First.

Reverse Engineering Patterns. Reverse engineering patterns focus on how to help reengineers to acquire a gradual understanding of a legacy system. They are based on code information but also on organization and human relationships. Reverse engineering pattern examples are: Read all the code in one hour, Refactor to Understand, Tie Code and Question [DDN00c], [DDN00d], [DDN00b].

Reengineering Patterns. A *reengineering pattern* connects an observable problem in the code to a reengineering goal: it describes the process of going from the existing *legacy* solution causing or aggravating the problem to a new *refactored* solution which meets the reengineering goal. It thus gives a method appropriate for a specific problem, rather than proposing a general methodology, and makes reference to the appropriate tools or techniques for obtaining the refactored solution. Examples of reengineering patterns are Move Behavior Close to Data, Decouple Chain of Classes, Conditionals into Polymorphism, Conditionals into Registration, Cure God Class.

12.3 State of the Art on Reengineering Patterns and Comparison

Four main groups or people qwork actively on Reengineering Patterns: Alan O'Callaghan from De Montfort University Stevens of University of Edinburgh [SP98], the team of Ralph Johnson at the University of Illinois, while focusing more deeply on refactoring produces some reengineering patterns [FO94], [FO95], [FY96], [FY00] and our group the SCG of the University of Bern. Some isolated work like [Kel00] also presents some reengineering patterns.

Our reengineering patterns are close to the Systems Reengineering Patterns [SP98] or the patterns written by O'Callaghan [O'C98]. The main difference is that our reengineering patterns are low level and focus in particular on object-oriented legacy systems. Note that our patterns cannot be used to evaluate whether or not an application should be reengineered in the first place; this difficult task has been tackled by [STS97] and [RSW98]. In [BS95] a methodology is proposed to help in the migration of legacy systems (principally legacy database systems) to new platforms.

Reengineering patterns differ from Design Patterns [GHJV95] in their emphasis on the *process* of moving from an existing *legacy* solution to a new *refactored* solution. Whereas a design pattern presents a solution for a recurring design problem, a reengineering pattern presents a refactored solution for a recurring legacy solution which is no longer appropriate, and describes how to move from the legacy solution to the refactored solution. The mark of a good reengineering pattern is (a) the clarity with which it exposes the advantages, the cost and the consequences of the target solution with respect to the existing solution, and not how elegant the target solution is, (b) the description of the change process: how to get from one state of the system to another.

We also contrast reengineering patterns with AntiPatterns [BMMM98]. AntiPatterns, as exposed by Brown et al., are presented as "bad" solutions to design and management issues in software projects. Many of the problems discussed are managerial concerns that are outside the direct control of developers. Moreover, the emphasis in AntiPatterns is on prevention: how to avoid making the mistakes which lead to the antipatterns. Consequently, AntiPatterns may be of interest when starting a project or during development but are no longer helpful when we are confronted with a legacy system. In approaching legacy systems we prefer to withhold judgment and use the term "legacy solution" or "legacy pattern" for a solution which at the time, and under the constraints given, seemed appropriate. In reengineering it is too late for prevention, and reengineering patterns therefore concentrate on the cure: how to detect problems and move to more appropriate solutions.

Finally, our reengineering patterns are different from code refactorings [Opd92], [Rob99], [TB99a], [FBB$^+$99]. A reengineering pattern describes a process which starts with the detection of the symptoms and ends with the refactoring of the code to arrive at the new solution. A refactoring is only the last stage of this process, and addresses the technical issue of automatically or semi-automatically modifying the code to implement the new solution. Reengineering patterns also include other elements which are not part of refactorings: they emphasize the context of the symptoms, by taking into account the constraints that reengineers are facing, and include a discussion of the impact of the changes that the refactored solution may introduce.

A reengineering pattern may describe a solution that would not be ideal if one is designing a system from scratch, but is a good solution under the current constraints of the legacy system. For example, if the constraint is that changes must be kept local some solutions are clearly not applicable even if they seem at first hand to be the best solutions.

12.4 An Example: Transform Conditionals to Registration

Intent. Increase flexibility between classes providing services and classes using them by transforming conditionals into a registration mechanism.

Problem

How can you reduce the coupling between tools providing services and tool users so that the addition or removal of tools does not lead to change the code of the tool users (as shown by the Figure 12.1)?

Symptoms

- Everytimes you remove certain functionalities from your system or a tool, you have to remove one case in some conditional statements, else certain parts (tool users) would still reflect the presence of the removed tools leading to fragile systems.
- Everytimes you add new functionality (i.e., for example importing different file formats like Flash, HTML, gif, JPEG), you have to add a new case in all the tool users that could use this new functionality.

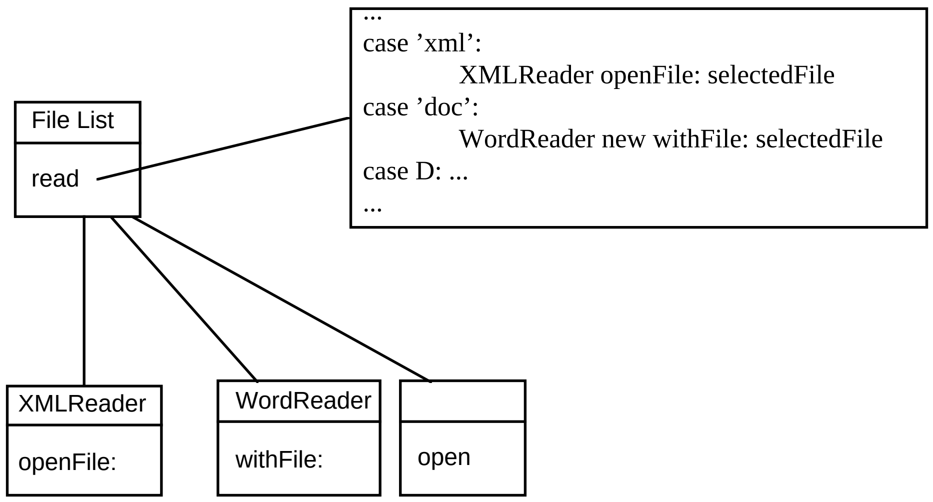


Figure 12.1: Tool Users use conditionals to determine which Tool should be invoked.

Solution

Replace conditional statements linking a set of classes providing services, (the tools), and the classes that used them , (the tools users), by making the tools register themselves to a registry mechanism and the tool users invoking the tool via the registry mechanism (see Figure 12.2).

Detection

Look for conditionals that dispatch on different values. The value identifies the tool(s) that have to be used.

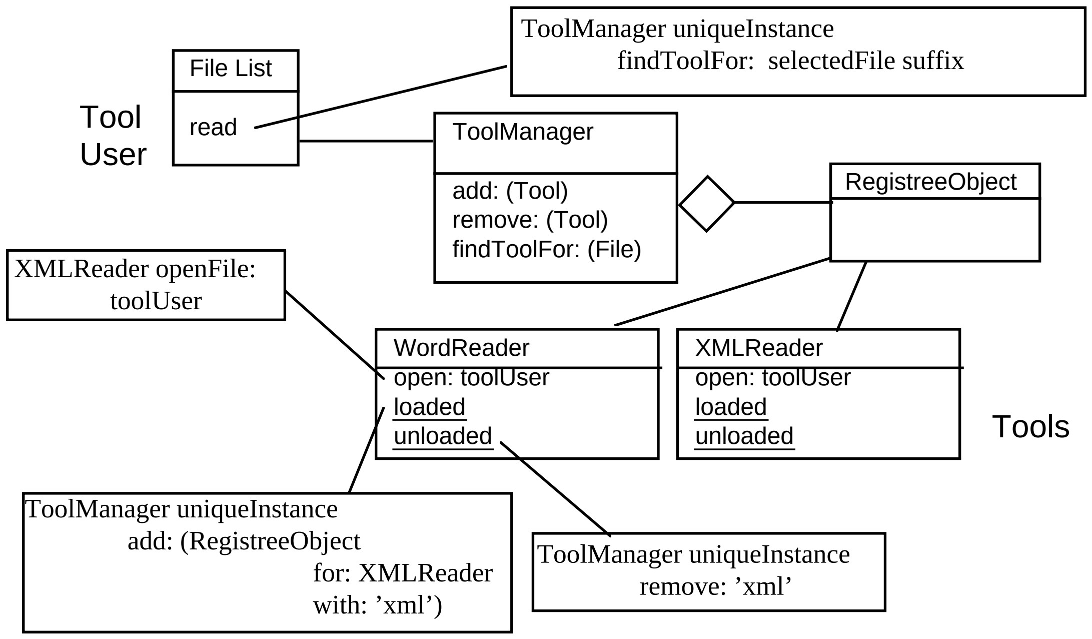


Figure 12.2: Transforming conditionals in tool users by introducing a registration mechanism and defining a clear protocol for communication between the tools and the tool users and for tool registration.

Steps

1. Define a class representing registree objects, i.e., an object representing the necessary information for registering a tool. Although, the internal structure of this class depends on the purpose of the registration, a registree object should provide the necessary information so the tool manager can identify it, create instance of the represented tool and invoke methods.

2. Define a class (a tool manager) that manages the registree objects and that will be queried by the tool user to check the presence of the tools. This class is certainly a singleton as the registrees representing the tools available should not be lost if a new instance of the registree manager is created.

3. For each case of the conditional, define a registree object associated with a given tool. The creation of this object and its registration into the tool manager must be made automatically when the tool it refers to is loaded. In a similar manner the registree object should be unregistrered as soon as its associated tool is not available anymore.

4. Define a method or a similar mechanism that is invoked by the tool user when it needs to invoke the tool. To support the following step, a common protocol should be defined to which each registree objects (or tool should conform to depending on the mechanism used) to invoke a given tool. To pass information from the tool user to the tool, the current tool user can be passed as argument when the tool is invoked.

5. Transform the complete conditional expression into a query to the tool manager object. This query should return a tool associated to the query and invoke it to access the wished functionality.

6. If the tool user class defined methods for the activation of the tools, such methods are now been moved into the tool. These methods should be removed from the tool user class.

Example

The following example is extracted from Squeak. We slightly modified the original code to improve the example readibility. In Squeak, the `FileList` is a tool that allows one to load different kinds of files in the system like Smalltalk code, JPEG images, MIDI files, HTML.... Depending on the suffix of the selected file, the `FileList` proposes different actions to the user. We show in the example the loading of the different file depending on their format.

Before

The `FileList` implementation creates different menus items representing the different possibilities depending on the suffix of the files. The dynamic part of the menu is defined in the method `menusForFileEnding:` that requires a suffix as argument and returns a menu item containing the label of the menu item and the name of the corresponding method that should be invoked on the `FileList` object.

```
FileList>>menusForFileEnding: suffix

   (suffix = ’jpg’) ifTrue:
      [ˆMenuItem label:’open image in a window’.
          selector: #openImageInWindow].
   (suffix = ’morph’) ifTrue:
      [ˆMenuItem label: ’load as morph’.
         selector: #openMorphFromFile].
   (suffix = ’mid’) ifTrue:
      [ˆMenuItem label: ’play midi file’.
         selector: #playMidiFile].
   (suffix = ’st’) ifTrue:
      [ˆMenuItem label: ’fileIn’.
         selector: #fileInSelection].
   (suffix = ’swf’) ifTrue:
      [ˆMenuItem label: ’open as Flash’.
         selector: #openAsFlash].
   (suffix = ’3ds’) ifTrue:
      [ˆMenuItem label: ’Open 3DS file’.
         selector: #open3DSFile].
   (suffix = ’wrl’) ifTrue:
      [ˆMenuItem label: ’open in Wonderland’.
         selector: #openVRMLFile].
   (suffix = ’html’) ifTrue:
      [ˆMenuItem label: ’open in html browser’.
         selector: #openInBrowser].
   (suffix = ’*’) ifTrue:
      [ˆMenuItem label: ’generate HTML’.
         selector:#renderFile].
```

The methods whose selectors are associated in the menu are implemented in the `FileList` class. We give two examples here. First the method checks if the tool it needs is available, if not it produces a beep, else the corresponding tool is created then used to treat the selected file.

```
FileList>>openInBrowser
   Smalltalk at: #Scamper ifAbsent: [ˆ self beep].
   Scamper openOnUrl: (directory url , fileName encodeForHTTP)

FileList>>openVRMLFile
   | scene |
   Smalltalk at: #Wonderland ifAbsent: [ˆ self beep].
   scene := Wonderland new.
   scene makeActorFromVRML: self fullName.
```

After

The solution is then to let every tool the responsibility to register themselves and let the `FileList` query the repository of available tools to find which tool can be invoked.

Step1

The solution is to first create the class `ToolRegistree` representing the registration of a given tool. Here we store the suffix files, the menu label and the action to be performed when the tools will be invoked.

```
Object subclass: #ToolRegistree
   instanceVariableNames: ’fileSuffix menuLabelName blockToOpen ’
```

Step 2

Then the class `ToolsManager` is defined. It defines a structure to hold the registered tools and defines behavior to add, remove and find registered tool.

```
Object subclass: #ToolsManager
   instanceVariableNames: ’registrees ’

ToolsManager>>initialize
   registree := OrderedCollection new.

ToolsManager>>addRegistree: aRegistree
   registrees add: aRegistree

ToolsManager>>removeRegistree: aBlock

   (registrees select: aBlock)
      do: [:each| registrees remove: each]

ToolsManager>>findToolFor: aSuffix
```

```
   "return a registree of a tool being able to treat file of format
    aSuffix"

   ˆ registrees
         detect: [:each| each suffix = aSuffix]
         ifNone: [nil]
```

Note that the `findToolFor:` method could take a block to select which of the registree objects satisfying it and that it could return a list of registree representing all the tools currently able to treat a given file format.

Step 3

Then the tools should register themselves when they are loaded in memory. Here we present two registrations, showing that a registree object is created for each tool. As the tools need some information from the `FileList` object like the filename or the directory, the action that has to be performed take as parameter the instance of the `FileList` object that invokes it (`[:fileList |` in the code below). In Squeak, when a class specifies a class (static) `initialize` method, this method is invoked once the class is loaded in memory. We then specialize the class methods `initialize` on the class `Scamper` and `Wonderland` to invoke the class methods `toolRegistration` defined below:

```
Scamper class>>toolRegistration

   ToolsManager uniqueInstance
      addRegistree:
      (ToolsRegistry
            forFileSuffix: 'html'
            openingBlock:
               [:fileList |
               self openOnUrl:
                  (fileList directory url ,
                     fileList fileName encodeForHTTP)]
            menuLabelName: 'open in html browser')

Wonderland class>>toolRegistration

   ToolsManager uniqueInstance
      addRegistree:
      (ToolsRegistry
            forFileSuffix: 'wrl'
            openingBlock:
               [:fileList |
               | scene |
               scene := self new.
               scene makeActorFromVRML: fileList fullName]
            menuLabelName: 'open in Wonderland')
```

In Squeak, when a class is removed from the system, it receives the message `removeFromSystem`. Here we then specialize this method on every tool so that they unregister themselves.

```
Scamper class>>removeFromSystem

   super removeFromSystem.
   ToolsManager uniqueInstance
      removeRegistree: [:registree| registree forFileSuffix = ’html’]

Wonderland class>>removeFromSystem

   super removeFromSystem.
   ToolsManager uniqueInstance
      removeRegistree: [:registree| registree forFileSuffix = ’wrl’]
```

Step 4

The `FileList` object now has to use the `ToolsManager` to identify the right registree object depending on the suffix of the selected file. Then if a tool is available for the suffix, it creates a menu item specifying that the `FileList` has to be passed as argument of the action block associated with the tool. In the case where there is no tool a special menu is created whose action is to do nothing.

```
FileList>>itemsForFileEnding: suffix

   registree := ToolManager uniqueInstance
               findToolFor: suffix ifAbsent: [nil].
   ˆ registree isNil
      ifFalse: [Menu label: (registree menuLabelName)
               actionBlock: (registree openingBlock)
               withParameter: self]
      ifTrue: [ErrorMenu new
               label: ’no tool available for the suffix ’, suffix]
```

Tradeoffs

Pros

- By applying Transform Conditionals into Registration you obtain a system which is dynamic, letting the responsibility to each tool to declare its presence. The proposed solution allow the transparent addition of new tools without implying any modification from the tool users.

- The actions that you moved from the tool user class to the associated tool may be much simpler because you do not have to test anymore that the right tool is available. The registration mechanism ensures you that the action can be performed.

- The interaction protocol between every tool and the tool user is now normalized.

Cons

- You have to define two new classes, one for the object representing tool registration and the object managing the registered tools.

- You may be forced to define a method in the tool for the action been invoked by the tool user. This way you are linking the tool with the tool user class in the tool class whereas the legay solution was linking them in the tool user. In Smalltalk, as shown by the example such a method can replaced by the definition of a block limiting the link between both classes. In Java, an inner class can be used instead of defining a new method.

- If not already existing new protocols for loading and removing tools have to be put in place and follow all the tools. For example, the tool programmer must have the possibility to specify actions at load time and at unload time.

Difficulties

- While transforming one case of the case statement into a registree object, you will have to define an action associated with the tools via the registree object. To ensure a clear separation and full dynamic registration, this action should be defined on the tool and not anymore on the tool user. However, as the tool may need some information from the tool user, the tool user should be passed to the tool as parameter whne the action is invoked. This changes the protocol between the tool and the tool user from a single invocation on the tool user to a method invocation to the tool with an extra parameter. This also implies that in some cases the tool user class have to define new public or friend methods to allow the tools to access the tool user right information.

- If each single conditional branch is associated only with a single tool, only one registree object is needed. However, if the same tool can be called in different ways we will have to create multiple registree objects. You need to identify the right criteria, usually the expression used in the conditional give a good discrimator. Creating multiple registree objects may lead to multiple registration aspects and registree classes depending on the level of felxibility of the implementation language.

When the legacy solution is the solution.

If all the tools are always available and you will never add or remove at run-time a new tool, a conditional is perfect.

Related Patterns

Transform Conditionals into Registration is related to Transform Client Type Checks by the fact that they both eliminate conditional expressions that discriminate to select which method should be invoked on which object. The main difference between these two patterns relies in their use of the flexibility they provide. Indeed, both allow one to add new tools (service providers) without having to change clients. However, Transform Conditionals into Registration provides an architecture that supports the dynamic use of the available service providers while Transform Client Type Checks only provide code flexibility without infrastructure support. In this sense, Transform Conditionals into Registration can be seen as a generalization of Transform Client Type Checks.

12.5 Contributions

Bibliography.

[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.

[DDT99] 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.

[DDN01a] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering: A Pattern-Based Approach. Morgan Kaufman Publisher, 2002.

[DNR98a] S. Ducasse, R. Nebbe, and T. Richner. Two reengineering patterns: Eliminating type checking. In P. Dyson, editor, *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999*, Konstanz, Germany, July 1998. UVK Universitätsverlag Konstanz GmbH.

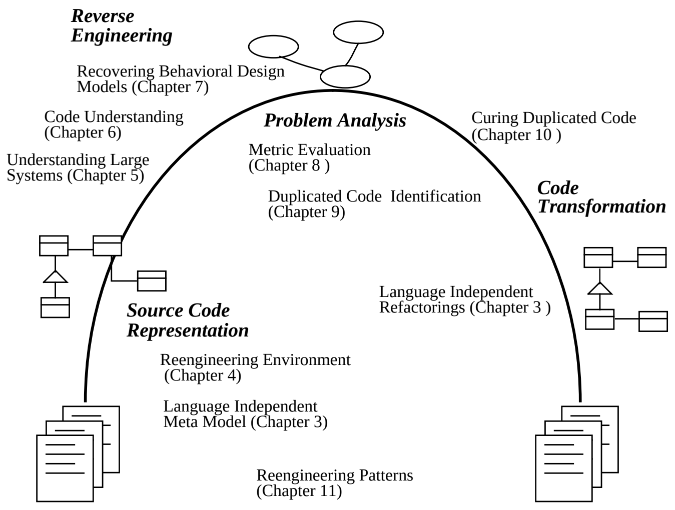
[DRN99] S. Ducasse, T. Richner, and R. Nebbe. Type-check elimination: Two object-oriented reengineering patterns. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of WCRE'99 (6th Working Conference on Reverse Engineering)*, pages 157–168. IEEE Computer Society Press, Oct. 1999.

Chapter 13

Conclusion

”Programs, like people, get old. We can’t prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products.”[Par94]

In this chapter we step back and evaluate our research. Then we present the possible future axes of research.


Reverse Engineering
Recovering Behavioral Design Models (Chapter 7)
Code Understanding (Chapter 6)
Understanding Large Systems (Chapter 5)
Problem Analysis
Metric Evaluation (Chapter 8)
Duplicated Code Identification (Chapter 9)
Curing Duplicated Code (Chapter 10)
Code Transformation
Language Independent Refactorings (Chapter 3)
Source Code Representation
Reengineering Environment (Chapter 4)
Language Independent Meta Model (Chapter 3)
Reengineering Patterns (Chapter 11)

13.1 Evaluation

The research we presented covers the fundamental areas of a reengineering effort we identified in the first chapter: *Source code representation*, *reverse engineering*, *problem analysis* and *code transformation*. We proposed a way to represent, store, exchange and manipulate source code of application developed in the different languages. We defined new approaches to understand a system at various levels of granularity and from different angles. We evaluated how metrics can assess code quality and proposed a simple and efficient approach to identify duplicated code. We proposed ways to cure duplicated code based on refactorings steering and evaluated how language independant refactorings could be expressed.

Our research, while anchored on concrete industrial problems, did not deal with the problems of assessing the impact or the cost of new requirements in terms of changes. This is principally due to the fact that we never had access to the requirements nor cost information of a legacy system, else we could have taken such information into account.

Now we evaluate the work we presented and identify potential research axes. Note that we try to not repeat the future axes we presented in the document, so we invite the interested reader to read the detailed description at the end of each chapter.

13.1.1 FAMIX and MOOSE

FAMIX answered our need to deal with applications written in different languages. Its extensibility allows us to customize it to the needs of the different languages. Moreover the approach proves to be scalable which is an important characteristics in the context of reengineering. However dealing with multiple languages are intrinsic limits of our approach in its current form. Indeed, our choice to analyze several languages, while well-suited for dealing with reverse engineering activities or simple computation like basic metrics does not support well complex analysis like program slicing, data flow analysis which are based on abstract syntax tree representation. The intention behind FAMIX was to support simple analysis in a language independant basis and to offer abstract syntax representation when needed. The question to know if AST representation can be language independant is now open and can be the topic of future research.

With the same limits induce by the code representation of languages that do not offer reflective capabilities such Smalltalk [FJ89] [Riv96b] or CLOS [KdRB91], the other limit of our approach is the lack of causal connection between the source code and its representation. Hence, every time the model is modified the source code elements it represents have to be regenerated or vice versa.

13.1.2 Reverse engineering large applications

The work we made around CODECRAWLER was successful because of its intuitive simplicity. As we mentioned in the Chapter 6 the approach could be improved to take into account (1) aggregation or collaboration relationships, (2) groups of entities, (3) new graph layouts, and (4) other ways to add semantical information in the graph.

Possible Research Axes. Research in the direction of design extraction and architectural recovery are areas on which we did not work because of lack of time and that are still important for reverse engineering. From the experiments we made with Rational Rose, it is obvious that filtering and representing code entities as boxes and arrows is not sufficient. Research efforts trying to support the recovery of design patterns did not prove to be really successful. Thus alternative approaches to support the inference of design information via propagation of annotations could be an interesting research topic.

Visualization of programs is clearly a field where new approaches can be successful. Visualization and information compression [HIBM97] are definitively an aid for program understanding. Cliches identification [RS97] and program compression [Bal97] are also interesting approaches that help understanding programs. [DeB97] presents an approach to use domain information as a base to steer the reverse engineering process. Evaluating such an approach could be interesting because high-level information would be confronted with source code information.

13.1.3 Supporting Code Understanding

We consider the work we made on class blueprints as a first step in the direction of support for class understanding. Class blueprints allows us to define a visual vocabulary that identify characteristic attributes of classes. However, “normal” classes show common blueprints which are not specific enough. We think that the next steps are to focus on these normal classes. This could be done using visulization techniques, analysis, heuristics, or/and IDE support.

13.1.4 Use of Dynamic Information

Our experience on iteratively extracting views by combining static and dynamic information reduces the amount of information generated by dynamic information and in the same time enriches static information with dynamic aspects of the system. What we think is missing is a methodology to guide the reengineer based on a set of well-known views. The intrinsic limits of this approach is the quality of the scenario used to collect dynamic information.

Possible Research Axes. Based on the combination of static and dynamic information, other approaches such as the recovery of objects belonging to a common collaboration and playing a given role are interesting areas that could improve the reverse engineering of applications.

Using dynamic information to qualify more precisely the relations that exist between classes could bring useful information when extracting design information. Code coverage information and code interpretation mixed with program visualization could be used to support run-time understanding of the application [DPKV94].

13.1.5 Metrics

We found metrics useful as a simple way to influence the perception of code entities. We learned that metrics coupled with visualization result in a powerful combination. In the context of the evaluation of metrics as a tool to assess the quality of a system, it would be interesting to continue our work with different metrics. However, we expect the same results, i.e., the unreliability of the metrics to detect badly design code entity. Moreover coupling and cohesion metrics are not really well-defined yet.

From the experiments we made while assessing more complex metrics such as coupling and cohesion metrics we learned that simplicity in metrics is necessary because it gives the reengineer the control of his interpretation.

Possible Research Axes. The experience we made on understanding refactorings based on metrics even if it was successful was extremely hard to perform due to the amount of code (even if the heuristic focus was good) to browse. It is why we think that metrics can be applied to understand evolution of a system but not at the level of granularity we used them. We believe that metrics are a good means to reduce the amount of information which extremely important when dealing with several versions of a legacy system. Hence we believe that simple metrics can be successfully used to support the understanding of software evolution coupled with visualization.

13.1.6 Detection of Code Duplication and Cure

The results we got with language independent code duplication identification are some of the most remarkable. This is clearly the case where language independence paid off more than expected. From a technical point of view this is also the work where we see the impact of the size of legacy systems on a simple idea. Indeed the implementation has to take account the size of the application. The only drawback of the approach is also its power: by being textual the identification does not provide language specific information which have to be recreated when necessary.

For the cure of duplicated code, it is difficult to draw generalizable conclusion due to the empirical aspect of the validation. We can say that in code where we knew that there was few duplication the duplication was localized to classes and their neighbors and the duplication elimination is not easy. We believe is that the proposed approach is the right one, i.e., supporting the reengineer by providing semi-automatic analysis and elements to support his decisions.

Using the prototype was successful even if we did not perform its scientific validation. The prototype helped us to quickly assess the duplication. By using the prototype we learned that curing non trivial duplicated code necessitates not only to have the knowledge of the application, the trade-offs between the different language constructs but also to have the following sources of information: the different duplication locations and their relationships.

13.1.7 Language Independent Refactorings

We validated our meta-model via the specification of language independent refactorings. Working on a language independent basis helps us to compare the different refactoring semantics. However we think that language refactorings are definitively an area where language independence is not an advantage. The algorithms get more complex to deal with issues due to multiple languages pecularities. We believe that much more efficient code transformation can be made on a per language basis.

Possible Research Axes. As excellent research has been done with for example the Refactoring Browser and on combining refactorings, it is difficult to do scientific research on refactorings anymore. Researchers have been working on using refactorings to introduce [1] Design Patterns. One of the topics where we believe refactoring research could be interesting is how to support the transformation of classes into components.

The identification of components in object-oriented code is an area where a lot of techniques can be applied [Kos99]. Looking at research efforts to identify objects into procedural code is a first start. We started working on the transformation of white box components into black-box ones. It necessitates specific analysis, tool support, and dedicated refactorings.

13.1.8 Reengineering Patterns

The work we made on the *reengineering patterns* is essential from the reengineering knowledge point of view because it records essential pratices in a “transferable” form. However, we do not consider reengineering patterns as a scientific contribution by itself. It is more a practical and empirical contribution for the reengineers of object-oriented applications. Our intention is also not to cover every aspects of reengineering but to identify the most important patterns. We hope other persons will complement our work with other reengineering patterns.

[1] An amusing note is that Wendorf in [Wen01] reports on how to remove misused Design Patterns during a reengineering effort too.

13.2 Potential Research Axes in Reengineering

Now we present all the future research axes that we are not direct extension of the presented work.

Supporting Team Reengineering. Most of the time reengineering an application is not a single person effort. Sharing information in the team is crucial. We are convinced that reengineering environments can benefit from collaborative environments. Representation of analysis and team work are challenging, for example, multiple users may want to share and edit different views of the same piece of code.

Code Analysis. Lot of techniques could be investigated like program slicing [GL91], [LH96], concept analysis [LS97], [SR97], clustering algorithms [AL99], data mining, learning systems like C4.5. [Wig97], [WBF97] presents a methodology for using clustering algorithms for identifying objects in legacy code. [BMW96] presents some discussions on the use of two processes to identify modules in Cobol code. [AL99] presents experiments using clustering algorithms as a remodularization method.

New Development Environments. When Smalltalk-80 came out it offered one of the first integrated development environment which was simply revolutionary at that time. All the Smalltalk code could be browsed, senders or implementers of a given message could be searched, instance variable access could be identified. Nowadays, most of the development environments provide the same functionality. In the meantime, the original Smalltalk environment did not really evolve. The Refactoring Browser is one of the few exception, it offers a powerful refactoring engine. Except for this work we think that there was nearly no significant advance in IDE to support the understanding of code. We think that the code browsers could be enhanced the following ways:

- Like a doctor that uses radiography to understand human body internals, we would like to have ways to change our perception of the code by applying some heuristics like all the methods accessing attributes x and y.

- The navigation of code should be improved to let developer follow program flow without losing track of the intent of the navigation. Navigation support like web browsing is only a first step in that direction.

- Usually a developer tends to be flooded by too much irrelevant information. First showing only certain classes, like the other classes involved in collaborations, or only the part invoked by the class under development, would be an interesting enhancement to investigate.

- Like when walking in a forest we use paths created by other persons or animals, we would like environments that reflect their own use. When developing a class, it often occurs that only a single aspect of a collaborator class is used. It would be interesting to see if the environment could customized itself by using the way it is used to present the most browsed code in a particular way.

- Using dynamic information of program execution like in some tools coverage [Til00] to add information or present browsed information into the code is another way to explore.

Evolution. Understanding the evolution of an application is an important research axe as evolution can help predict the future of the applications. Because visualization can compress information and that multiple versions of a huge system produce a huge amount of information, understanding evolution using visualization techniques are promissing. For example, we would like to understand what is the last subsystem that grows, did some classes changed from one subsystem to another one, what are the subsystems that ave been created since the beginning of the application, how subsystem structure changed,....

Identifying divergences or commonalities in presence of branches is also definitively important in particular during the development of frameworks.

Bibliography

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

[AT98] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of WCRE'98*, pages 30–40, 1998.

[Bae99] H. Baer. FAMIX C++ language plug-in 1.0. Technical report, University of Berne, September 1999.

[Bak92] B. S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak95] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995.

[Bal97] F. Balmas. Toward a framework for conceptual and formal outlines of program. In *Proceedings of WCRE'97*, pages 226–235, 1997.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24 of *ACM SIGPLAN Notices*, pages 1–6, 1989.

[BC00] Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). Technical report, Bell Canada, 2000.

[BDW99] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions of Software Engineering*, 25(1):91–121, 1999.

[BE96] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

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

[BFJR98] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP'98*, LNCS 1445, pages 396–417. Springer-Verlag, 1998.

[BG97] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *Proceedings of WCRE'97*, pages 2–12, 1997.

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

[BJ94] K. Beck and R. Johnson. Patterns generate architectures. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, LNCS 821, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag.

[BMB96] L. C. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. *IEEE Transactions of Software Engineering*, 22(1):68–85, 1996.

[BMD$^+$99] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings Sixth Working Conference on Reverse Engineering*, pages 326–336. IEEE Computer Society, 1999.

[BMD$^+$00] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In F. Balmas and K. Kontogiannis, editors, *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107. IEEE Computer Society, 2000.

[BMMM98] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. Antipatterns, 1998.

[BMR$^+$96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, 1996.

[BMW96] E. Burd, M. Munro, and Wezeman. Extracting reusable modules from legacy code: Considering the issues of module granularity. In *Proceedings of WCRE'96*, pages 189–198, 1996.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 - w3c recommendation 10-february-1998. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bro96] K. Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters thesis, North Carolina State University, 1996.

[BS95] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan Kaufman, 1995.

[BYM+98] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[Cas91] E. Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*. Ph.D. thesis, Centre Universitaire d'Informatique, University of Geneva, 1991.

[Cas92] E. Casais. An incremental class reorganization approach. In O. L. Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 114–132, Utrecht, The Netherlands, 1992. Springer-Verlag.

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

[CC90] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[CCdC92] G. Canfora, A. Cimitile, and U. de Carlini. A logic-based approach to reverse engineering tools production. *Transactions on Software Engineering*, 18(12):1053–1064, December 1992.

[CCdCDL98] G. Canfora, A. Cimitile, U. de Carlini, and A. De Lucia. An extensible system for source code analysis. *Transactions on Software Engineering*, 24(9):721–740, September 1998.

[CEK+00] J. Czeranski, T. Eisenbarth, H. M. Kienle, R. Koschke, E. Plödereder, D. Simon, Y. Zhang, J.-F. Girard, and M. Würthner. Data exchange in bauhaus. In *Proceedings WCRE'00*. IEEE Computer Society Press, Nov. 2000.

[CGK98] Y.-F. Chen, E. R. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Transactions on Software Engineering*, 24(9):682–693, September 1998.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, 1995.

[Chi00] S. Chiba. Load-time structural reflection in java. In *ECOOP 2000 – Object-Oriented Programming, LNCS 1850*, LNCS 1850, pages 313–336. Springer Verlag, 2000.

[CHRY96] M. P. Chase, D. R. Harris, S. N. Roberts, and A. S. Yeh. Analysis and presentation of recovered software architectures. In *Proceedings of WCRE'96*, pages 153–162, 1996.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CM93] M. Consens and A. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[CMR92] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

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

[CRW98] R. Clayton, S. Rugaber, and L. Wills. Incremental migration strategies: Data flow analysis for wrapping. In *Proceedings of WCRE'98*, pages 69–79. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[DD98] S. Demeyer and S. Ducasse. Do metrics support framework development? In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, 1998.

[DD99a] S. Demeyer and S. Ducasse. Metrics, do they really help? In J. Malenfant, editor, *Proceedings LMO'99 (Languages et Modèles à Objets)*, pages 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.

[DDHL96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On automatic class insertion with overloading. In *Proceedings of OOPSLA'96*, pages 251–267, 1996.

[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)*, pages 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*, pages 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.

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering: A Pattern-Based Approach*. Morgan Kaufman, 2002.

[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, pages 630–645. Springer-Verlag, Oct. 1999.

[DeB97] J.-M. DeBaud. Dare: Domain-augmented reengineering. In *Proceedings of WCRE'97*, pages 164–173, 1997.

[DeH98] K. DeHondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, 1998.

[Dem98] S. Demeyer. Analysis of overriden methods to infer hot spots. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, 1998.

[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.

[DPHKV93] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 326–337, 1993.

[DPKV94] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[DPLVW98] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[DPS99] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

[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)*, pages 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.

[DT00] S. Ducasse and S. Tichelaar. FAMIX Smalltalk language plug-in, 2000.

[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.

[EDL98] L. Etzkorn, C. Davis, and W. Li. A practical look at the lack of cohesion in methods metric. *Journal of Object-Oriented Programming*, 11(5):27–34, September 1998.

[Ern99] E. Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.

[ESJ92] S. G. Eick, J. L. Steffen, and E. E. S. Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[FBB$^+$99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 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)*, pages 95–104, 2001.

[FH79] Fjeldstad and Hamlen. Application program maintenance study- report to our respondents. Technical report, IBM Corporation, DP Marketing Group, 1979.

[FHK$^+$97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, nov 1997.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 327–336, 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[FNP98a] F. Fioravanti, P. Nesi, and S. Perli. Assessment of system evolution through characterization. In *ICSE'98 Proceedings (International Conference on Software Engineering)*. IEEE Computer Society, 1998.

[FNP98b] F. Fioravanti, P. Nesi, and S. Perli. A tool for process and product assessment of C++ applications. In *CSMR'98 Proceedings (Euromicro Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, 1998.

[FO94] B. Foote and W. F. Opdyke. Evolve aggregations from inheritance hierarchies: A consolidation pattern to support evolution and reuse. Department of Computer Science University of Illinois at Urbana-Champaign Urbana, 1994.

[FO95] B. Foote and W. F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[FP97] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[FR98] R. Fanta and V. Rajlich. Reengineering object-oriented code. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[Fre00] M. Freidig. XMI for FAMIX. Informatikprojekt, University of Berne, 2000.

[FS97] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, Oct. 1997.

[FTAM96] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. A cliché-based environment to support architectural reverse engineering. In *Proceedings ICSM '96*. IEEE, Nov. 1996.

[FY96] B. Foote and J. Yoder. Evolution, architecture, and metamorphosis. In J. M. Vlissides, J. O. Coplien, and N. L. Nerth, editors, *Pattern Languages of Program Design 2*, pages 295–318. Addison-Wesley, 1996.

[FY00] B. Foote and J. Yoder. Big ball of mud. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, pages 653–692. Addison-Wesley, 2000.

[Gar01] F. Garau. Concrete type inference in squeak. Master's thesis, Universidad de Buenos Aires, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[GKN01] G. Golomingi Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Berne, June 2001.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *Transactions on Software Engineering*, 17(18):751–761, August 1991.

[GR95] A. Goldberg and K. S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison-Wesley, Reading, Mass., 1995.

[Gra98] R. Grass. Software maintenance: Less is not more. *IEEE Software*, jul/aug 1998.

[Gri81] S. Grier. A tool that detects plagiarism in pascal programs. *SIGSCE Bulletin*, 13(1), 1981.

[Gxl01] Gxl (graph exchange language), 2001. http://www.gupro.de/GXL/.

[Hal77] M. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[Hau95] J. Haungs. A technical overview of visualworks 2.0. *The Smalltalk Report*, pages 9–14, January 1995.

[HEH$^+$96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. In *Automated Software Engineering, Vol. 3 Nos 1/2, June 1996*. 1996.

[Hel95] J. Helfman. Dotplot Patterns: A Literal Look at Pattern Languages. *TAPOS*, 2(1):31–41, 1995.

[Hen97] S. Henninger. Case-based knowlegde management tools for software development. *Journal of Automated Software Engineering*, 4(3), 1997.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90*, volume 25, pages 169–180, 1990.

[HIBM97] T. Hendrix, J. C. II, L. Barowski, and K. Mathias. Tool Support for Reverse Engineering Multi-lingual Software. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 136 – 1143. IEEE Computer Society, 1997.

[HM95] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95)*, 1995.

[HM96] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite; a measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.

[HMC97] P. Haynes, T. Menzies, and R. Cohen. *Software Visualization*, chapter Visualisations of Large Object-Oriented Systems. World-Scientific, 1997.

[Hol98a] R. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of WCRE'98*, pages 210–219. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[Hol98b] R. C. Holt. An introduction to TA: The Tuple-Attribute language. Technical report, University of Waterloo, Nov. 1998.

[How95] T. Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, 1995.

[HP96] R. C. Holt and J. Pak. Gase: Visualizing software evolution-in-the-large. In *Proceedings of WCRE'96*, pages 163–167, 1996.

[HS96] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[HWS00] R. Holt, A. Winter, and A. Schurr. Gxl: Toward a standard exchange format. In *Proceedings of WCRE'2000*, pages 162–172, 2000.

[HYR96] D. Harris, A. Yeh, and H. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1-2):109–139, 1996. model capture.

[Jan88] H. Jankowitz. Detecting plagiarism in student pascal programs. *Computer Journal*, 1(31):1–8, 1988.

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

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse*. Addison-Wesley/ACM Press, 1997.

[JO93] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, 1993.

[Joh92] R. E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 63–76, 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Joh93] J. H. Johnson. Identifying Redundancy in Source Code using Fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[Joh94] J. H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintence (ICSM)*, pages 120–126, 1994.

[Joh95] J. H. Johnson. Using Textual Redundancy to Understand Change. In *Proceedings of CASCON 95*, page CD ROM, 1995.

[JSB97] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing Interactions in Program Executions. In *Proceedings of ICSE 97*, pages 360–370, 1997.

[Kaz96] R. Kazman. Tool support for architecture analysis and design, 1996. Proceedings of Workshop (ISAW-2) joint Sigsoft.

[KC98] R. Kazman and S. J. Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[KC99] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 1999.

[KDM$^+$96] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Journal of Automated Software Engineering*, 3:77–108, 1996.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kel00] W. Keller. The bridge to the new town. In *Proceedings of Europlop'2000*, 2000.

[KG88] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 191–205, Nov. 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[KM96] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proceedings of ICSE-18*, pages 366–375. IEEE, 1996.

[KN] E. Koutsofios and S. C. North. *Drawing graphs with dot*. AT & T Bell Laboratories, Murray Hill, NJ.

[Kon97] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44–54. IEEE Computer Society, 1997.

[Kos99] R. Koschke. An incremental semi-automatic method for component recovery. In *WCRE'99 (Sixth Working Conference on Reverse Engineering)*, pages 256–267. IEEE, 1999.

[KP96] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedingsof WCRE '96*. IEEE, 1996.

[KSRP99] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse engineering of design components. In *Proceedings of ICSE'99*, 1999.

[Lan99] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Berne, Oct. 1999.

[Lau98] S. Lauesen. Real life object-oriented systems. *IEEE Software*, pages 76–83, 1998.

[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*, pages 108–124, 1996.

[Let98] T. C. Lethbridge. Requirements and proposal for a Software Information Exchange Format (SIEF) standard. Technical report, University of Ottawa, Nov. 1998.

[LH96] L. Larsen and M. Harrold. Slicing object-oriented software. In *Proceedings ICSE '96*, pages 495–505. IEEE, 1996.

[LHS97] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings OOPSLA '97*, ACM SIGPLAN, pages 304–317, 1997.

[LHS99] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LLB+98] B. Laguë, C. Leduc, A. L. Bon, E. Merlo, and M. Dagenais. An analysis framework for understanding layered software architectures. In *Proceedings of International Workshop on Program Comprehension (IWPC '98)*, 1998.

[LN95a] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[LN95b] D. Lange and Y. Nakamura. Program explorer: A program visualizer for C++. In *Proceedings of Usenix Conference on Object-Oriented Technologies*, pages 39–54, 1995.

[LPM+97] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of ICSM (International Conference on Software Maintenance)*, 1997.

[LRP95] J. Lamping, R. Rao, and P. Pirolli. A focus + context technique based on hyperbolic geometry for visualising larges hierarchies. In *Proceedings of CHI'95*, 1995.

[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.

[LS98] C. Lewerentz and F. Simon. A Product Metrics Tool Integrated into a Software Development Environment. In *Object-Oriented Technology Ecoop'98 Workshop Reader*, LNCS 1543, pages 256–257, 1998.

[LS99] P. K. Linos and S. R. Schach. Comprehending multilanguage and multiparadigm software. In *Proceedings of the short papers of ICSM'99*, pages 25–28, August 1999.

[Mad85] N. Madhavji. Compare: A collusion detector for pascal. *Techniques et Sciences Informatiques*, 4(6):489–498, Nov. 1985.

[Mal99] P. Malorgio. An information mural visualization for duploc. Informatikprojekt, University of Berne, July 1999.

[Mal01] P. Malorgio. Heuristics for object-oriented program analysis and understanding (temporary title), 2001. Master Thesis of the University of Berne.

[Man94] U. Manber. Finding similar files in a large file system. In *Proceedings 1994 Winter Usenix Technical Conference*, pages 1–10, 1994.

[Mar97] R. Marinescu. The use of software metrics in the design of object-oriented systems. Diploma thesis, University Politehnica Timisoara - Fakultat fur Informatik, 1997.

[Mar98] R. Marinescu. Using object-oriented metrics for automatic design flaws in large scale systems. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 252–253. Springer-Verlag, 1998.

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

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, pages 244–253, 1996.

[MN96] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *IEEE Computer*, pages 45–51, September 1996.

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

[Moo96] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA '96 Conference*, pages 235–250. ACM Press, 1996.

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

[Mur96] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[Neb99] R. Nebbe. FAMIX Ada language plug-in 2.2. Technical report, University of Berne, August 1999.

[Nes88] P. Nesi. Managing OO project better. *IEEE Software*, July 1988.

[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.

[O'C98] A. O'Callaghan. Pattern for change. In *Proceedings of Europlop'99*, 1998.

[OCN99] M. Ó Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings ICSM'99*. IEEE Computer Society Press, August 1999.

[OJ93] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 66–73. ACM Press, 1993.

[OMG98] Object Management Group. XML Metadata Interchange (XMI). Technical Report ad/98-10-05, Object Management Group, 1998.

[OMG99] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.

[OMG00] Object Management Group. Meta Object Facility (MOF) specification (version 1.3). Technical report, Object Management Group, Mar. 2000.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

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

[Pen95] F.-C. Penz. *Reverse Engineering in Smalltalk*. PhD thesis, Sozial-und Wirstschaftswissenschaftliche Fakultät Universität Wien, 1995.

[PP94] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, jun 1994.

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

[RBFDD98] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk, 1998. Proceedings of the ECOOP'98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP'98 Workshop Reader forthcoming LNCS).

[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.

[RD98] M. Rieger and S. Ducasse. Visual detection of duplicated code. In S. Ducasse and J. Weisbrod, editors, *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report. Forschungszentrum Informatik Karlsruhe, 1998.

[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)*, pages 13–22. IEEE, September 1999.

[RD01a] T. Richner and S. Ducasse. Iterative recovery of collaborations and roles in dynamically typed object-oriented languages. Technical report, University of Berne, 2001.

[RD01b] M. Rieger and S. Ducasse. Language independent duplicated code identification., 2001. Working Paper.

[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.

[Ree96] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[RG98] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98 ACM SIGPLAN Notices*, pages 117–133, Oct. 1998.

[Ric01] T. Richner. *Recovering Behavioral Design Models: A Query-based Approach*. PhD thesis, University of Berne, 2001.

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[Rie01] M. Rieger. *Language Independent Duplicated Code Identification*. PhD thesis, University of Berne, 2001.

[Riv96a] F. Rivard. *Smalltalk et Réflexivité*. PhD thesis, Ecole des Mines de Nantes, 1996.

[Riv96b] F. Rivard. Smalltalk : a Reflective Language. In *Proceedings of REFLECTION'96*, pages 21–38, Apr. 1996.

[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.

[Rob99] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[RPR93] H. Reubenstein, R. Piazza, and S. Roberts. Separating Parsing and Analysis in Reverse Engineering Tools. In *First Working Conference on Reverse Engineering*, pages 117–125, 1993.

[RS93] H. Ritch and H. M. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of WCRE '93*, pages 192–201. IEEE, 1993.

[RS97] A. A. Reeves and J. D. Schlesinger. Jackal: a hierarchical approach to program understanding. In *Proceedings of WCRE'97*, pages 84–12, 1997.

[RSK00] S. Robitaille, R. Schauer, and R. K. Keller. Bridging program comprehension tools by design navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, 2000.

[RSW98] J. Ransom, I. Sommerville, and I. Warren. A Method for Assessing Legacy Systems for Evolution. In *Proceedings of Reengineering Forum'98*, 1998.

[RW98] S. Rugaber and J. White. Restoring a legacy: Lessons learnt. *IEEE Software*, 15(4), July 1998.

[San96] G. Sander. Graph layout for applications in compiler construction. Technical report, Universitaet des Saarlandes, 1996.

[SBM01] M.-A. D. Storey, C. Best, and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[Sch01] A. Schlapbach. Generix XMI support for the MOOSE reengineering environment. Informatikprojekt, University of Berne, Jun 2001.

[SDSK00] G. Saint-Denis, R. Schauer, and R. K. Keller. Selecting a model interchange format. the spool case study. In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, 2000.

[Sef96] M. Sefika. *Design Conformance Management of Software Systems: an Architecture-Oriented Approach*. PhD thesis, University of Illinois, 1996.

[Sel90] P. Selfridge. Integrating code knowledge with a software information system. In *Proceedings of Knowledge-Based Software Engineering Conference*, pages 183–195, 1990.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SGMZ98] B. Schulz, T. Genßler, B. Mohr, and W. Zimmer. On the computer aided introduction of design patterns into object-oriented systems. In J. Chen, M. Li, C. Mingins, and B. Meyer, editors, *Proceedings of the 27th TOOLS conference*, 1998.

[SM95] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.

[SMW96] M.-A. D. Storey, H. A. Müller, and K. Wong. Manipulating and documenting software structures. In *Series on Software Engineering and Knowledge Engineering*, volume 7, pages 244–263. World Scientific Publishing Co., 1996.

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

[SP98] P. Stevens and R. Pooley. System reengineering patterns. In *Proceedings of FSE-6*. ACM-SIGSOFT, 1998.

[Squ01] http://www.squeak.org/, 2001.

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

[SRMK99] R. Schauer, S. Robitaille, F. Martel, and R. K. Keller. Hot spot recovery in object-oriented software with inheritance and composition template methods. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 220–229, 1999.

[SS00] S. E. Sim and M.-A. D. Storey. A strutured demonstration of program comprehension tools, 2000. Proceedings of WCRE 2000.

[Sta84] T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 5(10):494–497, September 1984.

[Sta90] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[Ste01] L. Steiger. Recovering the evolution of object oriented software systems using a flexible query engine. Diploma thesis, University of Berne, June 2001.

[STS97] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, 1997.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2), 1981.

[Sys99] T. Systa. On the relationship between static and dynamic models in reverse engineering java software. In *Proceedings of WCRE'99*, pages 304–313, 1999.

[Sys00a] T. Systa. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[Sys00b] T. Systa. Understanding the behavior of java programs. In *Proceedings of WCRE'2000*, pages 35–44, 2000.

[Tak96] TakeFive Software GmbH. *SNiFF+*, 1996.

[TB99a] L. Tokuda and D. Batory. Automating three modes of evolution for object-oriented software architecture. In *Proceedings COOTS'99*, 1999.

[TB99b] L. Tokuda and D. Batory. Automating three modes of evolution for object-oriented software architecture. In *Proceedings COOTS'99*, 1999.

[TD98] S. Tichelaar and S. Demeyer. An exchange model for reengineering tools. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, July 1998.

[TD99] S. Tichelaar and S. Demeyer. SNiFF+ talks to Rational Rose – interoperability using a common exchange model. In *SNiFF+ User's Conference*, January 1999.

[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.

[TFAM96] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo. Augmenting pattern-based architectural recovery with flow-analysis. In *Proceedings of WCRE'96*, pages 153–162, 1996.

[Tic99] S. Tichelaar. FAMIX Java language plug-in 1.0. Technical report, University of Berne, September 1999.

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

[Til99] M. Tilman. Building highly configurable and adaptive frameworks. ESUG'99 Summer School, http://users.pandora.be/michel.tilman/Publications/, 1999.

[Til00] M. Tilman. Building run-time analysis tools by means of pluggable interpreters, 2000. Presentation at ESUG'00, http://users.pandora.be/michel.tilman/Publications/.

[Top] http://www.topicmaps.org/.

[Way01] D. Way. Whisker: a multi method browser, 2001. http://www.mindspring.com/ dway/smalltalk/whisker.html.

[WBF97] T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *Proceedings of WCRE'97*, pages 24–33. IEEE Computer Society, 1997.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 71–76, 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

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

[Wen01] P. Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial product. In *Proceeding of CSMR'2001 (Conference on Software Maintenance and Reengineering)*, pages 77–84, 2001.

[Wer99] M. M. Werner. *Facilitating Schema Evolution With Automatic Program Transformation*. PhD thesis, Northeastern University, July 1999.

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

[Wig97] T. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 33 – 43. IEEE Computer Society, 1997.

[WMFB$^+$98] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, ACM SIGPLAN, pages 271–283. ACM, 1998.

[WMH93] N. Wilde, P. Matthews, and R. Hutt. Maintaining object-oriented software. *IEEE Software (Special Issue on Making O-O Work)*, 10(1):75–80, January 1993.

[WS95] N. Wilde and M. C. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[WTMS95] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.

[Wuy98] R. Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[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.

[YHC97] A. Yeh, D. Harris, and M. Chase. Manipulating recovered software architecture views. In *Proceedings of ICSE'97*, 1997.