Vrije Universiteit Brussel
Faculteit Wetenschappen – Department Informatica

A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems

Koen De Hondt

December 1998

Promotor: Prof. Dr. Theo D'Hondt
Co-promotor: Dr. Patrick Steyaert

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Vrije Universiteit Brussel
Faculty of Sciences – Computer Science Department

A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems

Ph.D. Dissertation

Koen De Hondt

December 1998

Supervisor: Prof. Dr. Theo D'Hondt
Co-supervisor: Dr. Patrick Steyaert

Abstract

Coping with evolution is one of the major issues in software development today. Software evolution is necessary to keep software from ageing in the rapidly changing world we live in. Software evolution is a continuous process that requires models of the software, in particular models of the software architecture. In absence of appropriate and reliable software documentation, reverse engineering is often the only way to acquire a mental model of the software.

The thesis of this research is that the software development environment should provide support for recording models that result from reverse engineering efforts, so that the models become online software documentation that can be exploited in subsequent software development activities.

This dissertation proposes *software classification* as an approach to architectural recovery in evolving object-oriented systems. The results of recovery are tangible entities in the software development environment. Software classification has two aspects: the software classification model and the software classification technique.
The *software classification model* is very simple – classifications are containers of items and items can be classified in multiple classifications –, but apparently it is a powerful model to organise software entities in a flexible and uniform manner. The *software classification technique* is the act of classification. Several software classification strategies are conceivable. In this work, four classification strategies are used in the applications of software classification: manual classification, virtual classification, classification with advanced navigation tools, and automatic classification through method tagging.
Five applications of software classification are discussed: expressing multiple views on software, recovery of collaboration contracts, recovery of reuse contracts, recovery of architectural components, and management of changes.

The viability of the proposed concepts, methods and tools is demonstrated by applying them on parts of a large and complex object-oriented software system developed in industry.

Keywords: software classification, collaboration contracts, reuse contracts, software evolution, reverse engineering, architectural recovery, object-oriented software development environments.

Contents

Contents **i**

List of Figures **viii**

List of Tables **xii**

Acknowledgements **xv**

1 Introduction **1**
- 1.1 Reverse Engineering . . . 3
- 1.2 Software Architecture . . . 4
- 1.3 Architectural Recovery . . . 4
- 1.4 Novel Approach to Architectural Recovery . . . 5
- 1.5 Larger Context . . . 7
- 1.6 Structure of the Dissertation . . . 7

2 Software Evolution **9**
- 2.1 The Economic Relevance of Software Evolution . . . 9
- 2.2 Software Evolution at a Technical Level . . . 10
- 2.3 Software Change Impact Analysis . . . 11
- 2.4 Software Evolution vs Software Maintenance . . . 14
- 2.5 Legacy Systems . . . 15
- 2.6 Software Evolution Problems . . . 15
  - 2.6.1 Version Proliferation . . . 16
  - 2.6.2 Poor Effort Estimation . . . 16
  - 2.6.3 Poor Insight into the Effect of Changes . . . 17
  - 2.6.4 Architectural Drift . . . 17
  - 2.6.5 Overfeaturing . . . 18
- 2.7 The Importance of Class Collaboration . . . 18
- 2.8 Collaboration Contracts . . . 19
  - 2.8.1 Not a Class Diagram, not a Sequence Diagram . . . 22
  - 2.8.2 Classes Play a Role in Many Collaboration Contracts . . . 22
  - 2.8.3 A Method May Play a Role in Many Collaboration Contracts . . . 23
  - 2.8.4 Classes are the Sum of Roles . . . 23

2.9 Evolution Model: Reuse Contracts . . . 24
2.9.1 Reuse of Collaboration Contracts . . . 24
2.9.2 Evolution of Collaboration Contracts . . . 27
2.9.3 Impact Analysis . . . 27
2.9.4 The Reuse Contract Model Attacks Evolution Problems . . . 27
2.9.5 The Reuse Contract Model as-is is not the Solution . . . 28
2.10 Summary . . . 29

3 Case Study: Evolution in Framework-Based Development 31
3.1 Context . . . 31
3.1.1 Software . . . 31
3.1.2 Development Style . . . 32
3.1.3 Process Model . . . 33
3.1.4 Framework Engineering . . . 34
3.1.5 Application Engineering . . . 34
3.1.6 Iterative and Incremental Software Development . . . 35
3.1.7 Evolutionary Software Development . . . 35
3.1.8 Management of Changes . . . 36
3.1.9 Software Development Environment . . . 37
3.2 Software Evolution Problems Revisited . . . 38
3.2.1 Version Proliferation . . . 38
3.2.2 Poor Effort Estimation and Poor Insight into the Effect of Changes . 39
3.2.3 Architectural Drift . . . 39
3.2.4 Overfeaturing . . . 40
3.3 Role of the Case Study in this Work . . . 41
3.3.1 Industry as Laboratory . . . 41
3.3.2 Requirements Provider . . . 41
3.3.3 Platform for Experiments, Evaluation and Feedback . . . 42
3.4 Summary . . . 42

4 Multiple Views on Software 43
4.1 Architectural Views . . . 43
4.1.1 Software Layers View . . . 43
4.1.2 Modules View . . . 44
4.1.3 Customisations View . . . 45
4.1.4 Features View . . . 46
4.2 Other Views . . . 47
4.2.1 Ownership View . . . 47
4.2.2 Traceability View . . . 48
4.3 Conclusion . . . 49

5 Requirements for the Recovery Process 51
5.1 Take Multiple Views on the Software into Account 51
5.2 The Recovery Process Should be Incremental 51
5.3 Motivate the Software Engineer 52
5.4 Keep the Model Simple 52
5.5 Keep the Recovery Process Lightweight 52
5.6 Integrate the Model and the Recovery Process in the SDE 53
5.7 Integrate Existing Software Entities in the Model 53
5.8 Make the Results of Recovery Tangible in the SDE 53
5.9 Cope with an Obscure Software System 54
5.10 Summary 54

6 Software Classification 57
6.1 Participant Model 58
6.1.1 A Participant is a View on a Class 58
6.1.2 Participant 59
6.1.3 Acquaintance Relationship 60
6.1.4 Method 60
6.1.5 Method Invocation 62
6.2 Software Classification Model 63
6.2.1 Classifications and Items 63
6.2.2 Classifications as Items 63
6.2.3 Classes as Items 63
6.2.4 Participants as Items 65
6.2.5 Classifications Have a Structure 65
6.2.6 Classifications Have a Classification Policy 66
6.2.7 Collaboration Contracts as Classifications 67
6.2.8 Reuse Contracts as Classifications 67
6.2.9 Virtual Classifications 68
6.2.10 The Classification Repository 68
6.3 Classification Strategies 69
6.3.1 Manual Classification 70
6.3.2 Virtual Classification 70
6.3.3 Classification with Advanced Navigation Tools 71
6.3.4 Automatic Classification Through Method Tagging 71
6.4 Summary 72

7 Recovery of Collaboration Contracts 75
7.1 Problem and Overview of the Proposed Solution 75
7.2 How does Source Code Map on Collaboration Contracts? 77
7.3 Issues in Recovering Collaboration Contracts 78
7.3.1 A Collaboration Contract is a Unit of Reuse 78
7.3.2 Where to Start and Where to Stop 79
7.3.3 Level of Detail 79

7.3.4 Coding Practice and Coding Conventions 79
7.3.5 Tool Support 80
7.4 Incremental Recovery Based on Software Classification 80
7.5 Identifying Concerns 83
7.5.1 Top-Down Identification 84
7.5.2 Bottom-up Identification 84
7.5.3 Extra Clues in Smalltalk 85
7.6 Identifying Key Classes 85
7.6.1 Problem Domain Concepts 86
7.6.2 Abstract Classes 86
7.6.3 Classes with Many Subclasses 87
7.6.4 Classes that Participate in Design Patterns 88
7.6.5 Classes or Instances Returned by Factory Methods 88
7.6.6 Abstract and Concrete Factory Classes 89
7.7 Identifying Key Collaborations 89
7.7.1 Template Methods 89
7.7.2 Method Invocations Between Key Classes 90
7.7.3 Frequently Overridden Methods 90
7.7.4 Invocations of Methods with the Receiver as Argument 90
7.7.5 Collaborations in Design Patterns 92
7.8 Identifying Acquaintances 93
7.8.1 Source Code Level Acquaintances 93
7.8.2 Categorisation of Source Code Level Messages 95
7.8.3 Collaboration Contract Acquaintances 97
7.8.4 Acquaintance Names 99
7.9 Extraction From the Source Code 101
7.9.1 Extraction of the Participant Interfaces 102
7.9.2 Extraction of the Interaction Structure 102
7.9.3 Extraction of Acquaintance Relations 102
7.9.4 Traceability of the Corresponding Flattened Class 103
7.10 Determining Acquaintance Classes In Smalltalk 103
7.10.1 A Lightweight Approach 104
7.10.2 Collecting the Required Interface 105
7.10.3 Computing the Conforming Classes 105
7.10.4 Determining the Best Match 106
7.10.5 Enhancing the Search for Conforming Classes 107
7.10.6 Problems with Metalevel Code 107
7.11 Summary 108

8 Recovery of Reuse Contracts 111
8.1 How Does Source Code Map on Reuser Clauses? 111
8.2 A 5-step Approach to Reverse Engineering 113
8.3 Step 1: Setting up the Initial Collaboration Contract 114
8.3.1 A Collaboration Contract for a Set of Classes 114

8.3.2 Example 114
8.4 Step 2: Setting up the Derived Collaboration Contract 116
8.4.1 A Collaboration Contract for a Set of Subclasses 116
8.4.2 Traceability of Initial Participants 116
8.4.3 Example 117
8.5 Step 3: Determination of the Extraction Reuser Clause 117
8.6 Step 4: Decomposition of the Extraction Reuser Clause 118
8.6.1 Decomposition Issues 119
8.6.2 Renaming 120
8.6.3 Context Extension and Context Cancellation 120
8.6.4 Context Refinement and Context Coarsening 120
8.6.5 Participant Concretisation and Participant Abstraction 120
8.6.6 Participant Extension and Participant Cancellation 121
8.6.7 Participant Refinement, Participant Specialisation, and Participant Coarsening 121
8.6.8 Participant Redefinition 121
8.6.9 Example 122
8.7 Step 5: Setting up the Reuse Contract 122
8.8 Summary 124

9 Classification with Advanced Navigation Tools 125
9.1 The Classification Browser 125
9.1.1 A View on the Classification Repository 125
9.1.2 Brief Overview of the Functionality 126
9.1.3 Default Top-Level Classifications 129
9.1.4 Browsing the Interaction Structure 130
9.1.5 Browsing Acquaintance Relationships 132
9.2 Application in Architectural Recovery 135
9.2.1 Recovery of Collaboration Contracts 135
9.2.2 Interoperability with UML 140
9.3 Application in Software Evolution 140
9.3.1 Recovery of Reuse Contracts 140
9.3.2 Keeping Collaboration Contracts in Sync 142
9.4 Experiment: Recovery of Design Documentation 143
9.4.1 The Provided Design Documentation 143
9.4.2 Preparing the Browser Context 144
9.4.3 Identifying Concerns 144
9.4.4 Identifying Key Classes 144
9.4.5 Identifying Key Collaborations 145
9.4.6 Identifying Acquaintance Relationships 146
9.4.7 Determining Acquainted Participants 147
9.4.8 Setting up the Collaboration Contract 147
9.4.9 Results 148
9.5 Summary 150

10 Automatic Classification Through Method Tagging 151
10.1 Tagging Methods in "whats'On" . . . 151
10.2 Processing the Method Tags . . . 153
10.3 Steady Increase in Classified Methods . . . 154
10.4 Application in Architectural Recovery: Recovery of Architectural Components . . . 157
10.5 Applications in Software Evolution . . . 158
10.5.1 Management of Changes . . . 158
10.5.2 Methods and Classes in Multiple Classifications . . . 161
10.6 Summary . . . 164
10.7 Considerations for Introduction of the Technique in a SDE . . . 165

11 How the Proposed Solution Meets the Requirements 167
11.1 Take Multiple Views on the Software into Account . . . 167
11.2 The Recovery Process Should be Incremental . . . 168
11.3 Motivate the Software Engineer . . . 168
11.4 Keep the Model Simple . . . 169
11.5 Keep the Recovery Process Lightweight . . . 170
11.6 Integrate the Model and the Recovery Process in the SDE . . . 171
11.7 Integrate Existing Software Entities in the Model . . . 171
11.8 Make the Results of Recovery Tangible in the SDE . . . 171
11.9 Cope with an Obscure Software System . . . 172

12 Related Work 173
12.1 Reuse Contracts . . . 173
12.2 Models . . . 173
12.3 Classification of Reusable Components . . . 174
12.4 Determining Acquaintance Classes . . . 176
12.4.1 Type Inference and Type Systems . . . 176
12.4.2 Runtime Statistical Typing . . . 176
12.5 Reverse Engineering . . . 177
12.5.1 Recovering Design Patterns . . . 177
12.5.2 Reflexion Models . . . 177
12.6 Program Visualisation . . . 179

13 Conclusion 181
13.1 Summary . . . 181
13.2 Contributions . . . 186
13.3 Future Work . . . 187
13.3.1 Architectural Components as Participants . . . 187
13.3.2 Generic Collaboration Contracts . . . 188
13.3.3 Keeping Classifications In Sync . . . 190
13.3.4 Enforcing Collaboration Contracts . . . 190
13.3.5 Expressing Architectural Constraints . . . 192

13.3.6 Software Classification Strategies 193
13.3.7 Classifications in Other Development Tools 194
13.3.8 Classifications in non-Smalltalk Environments 194

A Reuse Contracts 197
A.1 The Reuse Contract Model 197
A.2 Collaboration Contracts 198
A.2.1 Collaboration Contract 198
A.2.2 Well-Formedness 199
A.3 Reuse Contracts 200
A.3.1 Reuse Contract 200
A.3.2 Reuse Contract Type 200
A.4 Reuser Clauses 200
A.5 Participant Reuser Clauses 201
A.5.1 Participant Extension 202
A.5.2 Participant Cancellation 203
A.5.3 Participant Refinement 204
A.5.4 Participant Coarsening 205
A.5.5 Participant Specialisation 206
A.5.6 Participant Concretisation 208
A.5.7 Participant Abstraction 209
A.6 Context Reuser Clauses 210
A.6.1 Context Extension 210
A.6.2 Context Cancellation 211
A.6.3 Context Refinement 212
A.6.4 Context Coarsening 213
A.7 Combined Reuser Clauses 214

B Definition of Accessor Methods 217
B.1 Direct Accessor 217
B.2 Direct Accessor with Lazy Initialisation 217
B.3 Direct Accessor with Default 217
B.4 Indirect Accessor 218
B.5 Design-Specific Accessor 218
B.6 Developer Dependent Accessor 218
B.7 Collection Accessor 219

C Example Generated Rose Script 221

Bibliography 230

Vita 231

List of Figures

2.1 Dependency graph of software life cycle objects 12
2.2 Version proliferation 16
2.3 Collaboration contract diagram (separated static and interaction structure) 21
2.4 Collaboration contract diagram (combined static and interaction structure) 21
2.5 Classes play a role in several collaboration contracts 23
2.6 Methods play a role in several collaboration contracts 24
2.7 Combined collaboration contract - classes are the sum of roles 25
2.8 Adapted collaboration contract 26
2.9 Reuse contract diagram 26

3.1 The broadcast planning application (week view) 32
3.2 Making, logging and tracking changes in a request/defect system 36
3.3 Merging software changes into the software baseline 39

4.1 Software layers 44
4.2 Software modules 44
4.3 How modules relate to layers 45
4.4 The broadcast planning framework and its customisations 46
4.5 Combined customisations/layers/modules view (for two customisations) . . 47
4.6 Ownership view for two developers 48
4.7 Combined ownership/layer/module view for two developers 48
4.8 Traceability view 49

6.1 Participant 59
6.2 Acquaintance relationship 60
6.3 Implementation stereotypes 61
6.4 Method 61
6.5 Method signature 62
6.6 Method invocation 62
6.7 Classifications and items 63
6.8 Classifications as items 64
6.9 Classes as items 64
6.10 Participants as items 65
6.11 A classification has a structure 66

6.12 A classification has a classification policy 66
6.13 Collaboration contracts as classifications 67
6.14 Reuse contracts as classifications 68
6.15 Virtual classifications 69
6.16 Classification repository 69

7.1 Incremental recovery of collaboration contracts 81
7.2 Example of top-down identification of concerns 85
7.3 Collaboration contracts often exist in parallel class hierarchies 87
7.4 The receiver as argument is a clue for finding a collaboration contract . . . 91

8.1 5-step approach to reverse engineering reuse contracts 115
8.2 The initial collaboration contract describes part of the Collection class . . . 116
8.3 The derived collaboration contract describes part of the Set and Object classes 117
8.4 Decomposition of the change between initial and derived collaboration contracts 123

9.1 A Classification Browser is a view on the classification repository 126
9.2 Classification Browser 127
9.3 Default top-level classifications (some expanded) 129
9.4 Senders and implementers facilities in the Classification Browser 131
9.5 Acquaintance facilities in the Classification Browser 133
9.6 Source code level acquaintances ordered by stereotype in the browser 134
9.7 Context classifications created during incremental recovery 136
9.8 Classes as participants in the target classification 137
9.9 Classified methods in the recovered participants 138
9.10 Classified acquaintances in the recovered participants 139
9.11 UML class diagram generated from a collaboration contract 141
9.12 UML collaboration diagram generated from a collaboration contract 141
9.13 The provided class diagram 144
9.14 Two classifications, one with classes, one with participants 145
9.15 Classified and unclassified methods 146
9.16 Classified and unclassified acquaintances names 147
9.17 Determining acquainted participants 148
9.18 Static structure of the recovered collaboration contract 149
9.19 Interaction structure of the recovered collaboration contract 149

10.1 Method tagging dialog for incremental classification 152
10.2 Example tag with classification information 153
10.3 Processing method tags to produce classifications 153
10.4 Classifications generated from method tags 154
10.5 Number of incrementally classified methods over a two-month period 155
10.6 Number of incrementally classified methods per developer and per module . 156
10.7 Recovered framework customisations and modules 157

10.8 Making, logging and tracking changes with tag-based classifications 159
10.9 Changes corresponding to specifications and bug fixes in the Classification Browser 160
10.10 Exploring method changes in context 161
10.11 Number of classified classes in two different modules 162
10.12 Classes classified in two different modules 163

13.1 Instantiation of a generic collaboration contract 189
13.2 Collaboration contract for layered architecture 193
13.3 Reuser clause to be interpreted as negative information 193

List of Tables

7.1 Mapping from language concepts onto collaboration contract concepts . . . 78
7.2 Mapping from source code to acquaintances 99
7.3 Acquaintance names 101
7.4 Classes of Smalltalk literals 105
7.5 Best match for an acquaintance class 107

8.1 Mapping from reuse concepts onto reuse contract concepts 113

12.1 Elemental classes and compound classes in enumerative classification 174

A.1 Basic contract types 201

Acknowledgements

First and foremost, I thank my wife Nele for her love and support during the seemingly endless writing phase of this work.

I thank Prof. Dr. Theo D'Hondt and Dr. Patrick Steyaert for supervising this thesis. Patrick and I have been colleagues for eight years now, and the co-operation is still as fruitful as in the early days when Patrick, Wim Codenie, Marc Van Limberghen and I conceived Agora. I thank Patrick for long discussions on software classification and several other aspects of this work. He has been instrumental in the implementation of the Classification Browser as well.

I thank the members of the jury: Adele Goldberg (Neometron, Inc.), Oscar Nierstrasz (Software Composition Group, University of Berne), Viviane Jonckers (System and Software Engineering Lab, Vrije Univesiteit Brussel), and Robert Meersman (Systems Technology and Applications Research Laboratory, Vrije Universiteit Brussel). They gave a lot of feedback and raised many interesting questions. Adele Goldberg was generous with advice, on small as well as on big issues. Oscar Nierstrasz was very helpful to improve the dissertation. Having these widely known researchers as members of the jury is a real asset to this work.

I thank Wouter Roose, Wim Codenie and Wilfried Verachtert at MediaGeniX with whom I worked closely. I enjoyed working with them very much. I thank all other employees at MediaGeniX for the nice working atmosphere and for the prompt technical support when I needed it.

I thank other members of the Reuse Contract research group for reading drafts of this dissertation: Carine Lucas, Kim Mens, Tom Mens, and Roel Wuyts. I also thank Tom for help with UML and Rose scripts.

I thank Serge Demeyer of the Software Composition Group at the University of Berne for his support, and for useful comments on drafts of this document. When Serge got his PhD, he gave me a copy of his dissertation and wrote on the first page: 'Let this be an example.' I hope my dissertation lives up to his.

I thank Stéfane Ducasse of the Software Composition Group for reading drafts and commenting on it over e-mail.

I like to thank my other colleagues at the Programming Technology Lab for our co-operation over the years: Jan De Laet, Wolfgang De Meuter, Kris De Volder, Tom Lenaerts, Tom Tourwé, Werner Van Belle, and Mark Willems.

Part of the research reported on in this document was supported by the Flemish Institute for the stimulation of industrial research and technology transfer (IWT), project 950153.

Chapter 1

Introduction

The subject of this dissertation is architectural recovery in evolving object-oriented systems. The title reflects the subdomain of software engineering to which this research makes a contribution: the cross-section of object-oriented systems, software evolution, software architecture, and reverse engineering.

This research is driven by five observations.

One, models are crucial for evolution. A software engineer needs models of the software in order to evolve it correctly. Without models, software engineers can only solve the problem to the best of their ability, which may result in code duplication, reinvention of designs, and breaches of the software architecture. Models should provide answers to questions raised when a software system is subject to evolution: What is the software architecture? What is the design? What can be reused? Is redesign necessary? What is the effort that will be required? What is the code that needs to be modified? What are the constraints? What are the interactions with other parts? What is the impact on the rest of the software? What needs to be re-tested?

Two, software engineers have a hard time evolving object-oriented systems, because current object-oriented analysis and design methodologies badly support iterative and evolutionary development. The models provided by the methodologies are not resilient to change. Consequently, software documentation including such models is never up-to-date.

Three, knowing that the models in the software documentation are not up-to-date, software engineers resort to reverse engineering to get a mental model of the target software. Most popular software development environments for object-oriented languages are not equipped with specialised reverse engineering tools. Therefore, software engineers reverse engineer manually by browsing the source code. Reverse engineering involves a huge intellectual effort. Reading source code of object-oriented programs requires many context switches and a good concentration to keep a mental record of the recovered software architecture, the interrelationships between the architectural components, and the rationale behind the software design.

Four, reverse engineering is a recurrent development activity. Since evolution is a continuous process, reverse engineering recurs for each software evolution task. It also tends to recur for the same part of the source code. The same part of the source code may be reverse engineered more than once by different software engineers, as well as by the same engineer. There are two reasons for this recurrence. First, the models obtained through reverse engineering are not recorded and shared. Second, software engineers have limited memory capacity.

Five, models obtained through reverse engineering are not recorded. In general, software engineers do not record the models in the software documentation because of time constraints and because the development environment and the documentation environment are not integrated. Software development environments do not provide any means for recording the results of a reverse engineering session either. Therefore, the models remain in the heads of the software engineers. They cannot be shared among the other members of the development team, except through informal communication, which is error-prone. The inability to share models of the software is not only a technical problem. It is also a major managerial problem, because it decreases productivity.

Although the solution to this problem seems obvious – let the developer record what he[1] has learned by examining the source code –, this is not easily achieved. First, the lack of proper notations and models for evolution is a major impediment. The prevalent notations and models for describing, building and reasoning about software deal with evolution in an implicit manner only. Second, software development environments do not provide any means to represent recovered abstractions (models) that are not part of the programming language. Third, reverse engineering is not acknowledged as a recurrent development activity. Therefore, many software development environments do not provide support for recording recovered abstractions of the software whatsoever. The link between the extracted abstractions and the source code cannot be made explicit and is therefore hard to maintain.

The thesis of this research is that the software development environment should provide support for recording models that result from reverse engineering efforts, so that the models become online software documentation that can be exploited in subsequent software development activities.

This dissertation introduces *software classification*, a model and strategies for incremental recovery of the architecture of evolving object-oriented systems. The model is generic and it supports different levels of granularity, so that it can be used to describe different abstractions of the source code. The strategies for recovery are techniques to reverse engineer the abstractions and to record them in the software development environment as

[1]In the remainder of this document, wherever ''he" is used, ''he or she" is intended.

tangible software entities.

1.1 Reverse Engineering

This work is concerned with recovery. Recovery is a subarea of reverse engineering.

Chikofsky and Cross define reverse engineering as follows [CC90]: “Reverse engineering is the process of analysing a subject system to identify the system’s components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction.”

To make a clear distinction between normal engineering activities and reverse engineering activities, normal engineering is referred to by the term *forward engineering*. Reverse engineering is not to be confused with *re-engineering*. Reverse engineering is a process of examination and extracting design artefacts only, while re-engineering involves alteration of the software (being forward engineering) as well.

The goal of reverse engineering is to gain a better comprehension of the software. In their widely accepted paper [CC90], Chikofsky and Cross identify the six key objectives of reverse engineering: cope with complexity, generate alternate views, recover lost information, detect side effects, synthesise higher abstractions, facilitate reuse.

There are two important subareas of reverse engineering: redocumentation and design recovery. Redocumentation is the process of recovering software documentation that existed or should have existed. It is achieved by examining the software system itself, regardless of any other available information about the system. The results are pretty-printed source code, diagrams extracted from the source code, and cross-references [CC90].

Design recovery is concerned with the identification of meaningful higher level abstractions. According to Biggerstaff [Big89]: ”Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.” Design recovery requires input from the software engineers, usually problem domain and application experts, to correlate their knowledge with information extracted from the source code and any available documentation.

Design recovery is often a major hurdle and requires proper tool support [Cas96]. It is therefore not surprising that the development of reverse engineering methodologies and the development of supporting tools go hand in hand.

1.2 Software Architecture

This work focuses on *architectural recovery*. Shaw and Garlan define software architecture as follows. The software architecture is the overall system structure of a software system. It describes the organisation of the software as a composition of components. It lays down the global control structures and the protocols for communication, synchronisation and data access. It defines how functionality is assigned to design elements and how design elements are composed. It also addresses the physical distribution, the scaling and the performance, the dimensions of evolution and the selection among design alternatives [SG96]. There are several kinds of software architectures. A kind of architecture is known as an *architectural style*. Examples of architectural styles are: client-server, pipe and filter, blackboard, layered, and framework. The software architecture is also called the *structure* of the software by Ossher [Oss87] and Murphy [Mur96].

The definition of software architecture covers many issues, too many to cover all at once in the context of recovery. Therefore, in this work, the concept of software architecture is restricted to static structure (the composition of components) and the interaction structure (the protocols for communication). Consequently, architectural recovery will be restricted to the recovery of architectural components and the recovery of the interaction structure of these components. In this work, the term *architectural element* (also known as *architectural feature* [HYR96]) is used in a broad sense to refer to a design element, the composition of design elements, or the interaction between design elements.

1.3 Architectural Recovery

The definition of software architecture given in the previous section determines what the targets of the recovery activities are: the static structure and the interaction structure. Other aspects of the software architecture are not discussed in this dissertation.

In object-oriented systems, the static structure of a software architecture refers to the composition of classes in large architectural building blocks. Typical examples are software layers in a layered architecture, pipes and filters in a pipe-and-filter architecture, or framework and customisations in a framework architecture. In the remainder of this document, the term *architectural component* will be used to refer to large architectural building blocks, that is, software entities that consist of (whole or partial) classes. From the literature, it is not obvious what is a good model to describe architectural components. Therefore, this work will propose its own model. It will be demonstrated that software systems do not have one predominant software architecture. Often they have multiple architectures. The proposed model takes multiple architectures into account.

In an object-oriented system, the interaction structure of architectural components always boils down to the interaction between objects. The interaction between them is defined by their classes, which collaborate to achieve global behaviour. This work has its roots in the Reuse Contracts [Luc97] research. Therefore, this work adopts the collaboration contract model to describe the interaction structure of architectural components. It focuses on collaboration contracts for classes, however. Recovery of collaboration contracts for architectural components can be considered a generalisation of the proposed approach and is therefore deferred to future work (see Section 13.3.1).

The subject of this work is architectural recovery in *evolving* object-oriented systems. Therefore, this work is also concerned with the evolution of object-oriented systems. The reuse contract model is adopted to describe the evolution of collaboration contracts. The recovery of reuse contracts gives insight into the reuse and evolution of a software system. Since the reuse contract model provides a conflict detection scheme, the recovery of reuse contracts is essential to tap into the model's potential for impact analysis. This work only treats the recovery of reuse contracts, it does not discuss conflict detection, nor impact analysis.

In summary, this work has three foci of attention with respect to architectural recovery. First, the recovery of architectural components like software layers and framework customisations. Second, the recovery of collaboration contracts that describe the interaction structure of a set of classes. Third, the recovery of reuse contracts that describe the evolution of collaboration contracts.

1.4 Novel Approach to Architectural Recovery

The new approach to architectural recovery in object-oriented software systems proposed in this work is *software classification.* Software classification is a model, as well as a technique.

The *software classification model* defines the necessary concepts to describe and organise architectural components. The model is very simple: classifications are containers of items, and items can be classified in multiple classifications. Collaboration contracts and reuse contracts are embedded in the software classification model. Despite its simplicity, it is a very flexible model to organise software entities.

The *software classification technique* is a way to carry out a classification. The power of software classification comes from the application of the technique in architectural recovery. Several *software classification strategies* can be conceived. In this work, four strategies will be presented: manual classification, virtual classification, classification with advanced navigation tools, and automatic classification through method tagging. The choice of classification strategy depends on the goal of the recovery activity.
Manual classification is the most basic classification strategy. It can be used to create classifications of architectural elements to the software engineer's liking. The creation and recovery of multiple views on software is an application of manual classification.
Virtual classification is a classification strategy to draw architectural elements of the soft-

ware development environment into the software classification model. A direct application of this classification strategy is drawing existing Smalltalk categories into the classification model.
Classification with advanced navigation tools is a classification strategy to recover architectural elements that describe interaction structures or dependency relationships. This classification strategy is the basis of an important application of software classification: the recovery of collaboration contracts and reuse contracts.
Automatic classification through method tagging is a classification strategy to recover architectural components based on classification information tagged onto methods during forward engineering. Classifications are computed based on the method tags provided. This classification strategy has an interesting side effect that can be exploited: the method tags can be used to log and monitor the changes made to a software system. Automatic classification based on method tagging thus also has an application in the management of changes.

A recovery process requires proper tool support. Therefore, this dissertation supplements the software classification model and technique with tools that have the software classification model as their foundation, and that support the different software classification strategies.
An important characteristic of software classification as an approach to architectural recovery is the availability of the recovery results in the software development environment. The recovered software entities can be consulted in the development tools, and they can be used in subsequent development activities.

The viability of the proposed methods and tools is demonstrated by applying them on parts of a large and complex object-oriented software system developed in industry: a broadcast planning system for television and radio stations, called "whats'On". It serves as an excellent case study for this work. It is large and it is subject to constant evolution, so that good models of the software are crucial to evolve it correctly and effectively. The original software engineers are still around, so that the results of validation experiments can be compared against their knowledge of the software system. All source code since the conception of the software is available in the repository of a versioning system, so that the evolution of the software can be traced.

This research is conducted according to the industry-as-laboratory approach. Therefore, the case study plays a crucial role in several aspects of this research. The case study is a medium for dialogue with software engineers "in the field", and it is an ideal context to acquire knowledge about the production of "real world" software. It is an excellent platform for collecting requirements, conducting experiments, evaluating ideas and prototypes, and getting early feedback.
Not only the case study is crucial for this research; the commitment of the software provider's management to this research is crucial too. The investigation of the models, methods, and tools proposed in this dissertation, in particular the classification strategy *automatic classification through method tagging*, requires changes to the software develop-

ment environment and the software development process. Without the commitment of the software provider's management, some parts of this research are impossible to conduct.

1.5 Larger Context

This work is part of a research effort aiming at a fully-fledged methodology based on reuse contracts, backed up with a set of tools to enforce that methodology. This dissertation is the second one in this area. While Carine Lucas' dissertation [Luc97] established the foundation of a disciplined reuse methodology, this dissertation discusses tools and the underlying methods that can be built on top of that foundation. Since collaboration contracts are not yet used in forward engineering, recovery of collaboration contracts is an important step towards a development environment in which collaboration contracts play an active role in software development. The recovery of reuse contracts is a crucial step towards a development environment in which reuse contracts can be exploited to perform impact analysis and effort estimation.

1.6 Structure of the Dissertation

Chapter 2 discusses issues in software evolution. An overview of collaboration contracts and reuse contracts is given as they are the target abstractions of the recovery processes discussed in Chapter 7 and Chapter 8.

Chapter 3 presents the case study or the context in which a large part of this research took place and discusses the role of the case study in this work.

One role of the case study is that of requirements provider. Chapter 4 discusses the requirement of multiple views on software. Chapter 5 discusses other requirements for the recovery process, the model underlying the process, and the tools that support the process.

Chapter 6 discusses software classification. It presents the software classification model and introduces software classification strategies. The model and the strategies will be used during recovery activities, as discussed in Chapter 7, Chapter 9, and Chapter 10.

Chapter 7 discusses recovery of collaboration contracts based on incremental software classification. Chapter 8 proposes a 5-step approach to recovery of reuse contracts.

Chapters 9 and 10 discuss two software classification strategies in detail, including their applications: classification with advanced navigation tools, and automatic classification through method tagging.

Chapter 11 discusses how the proposed classification model, the classification strategies, the methods for recovery, and the tool support meet the requirements stated in Chapter

4 and Chapter 5.

Chapter 12 discusses related work.

Finally, Chapter 13 summarises, draws conclusions, and outlines future work.

For reference, Appendix A recapitulates the essential (formal) definitions and terminology concerning collaboration contracts and reuse contracts. Since detailed explanation of the concepts can be found in Lucas' dissertation [Luc97], no elaborate explanation is given. However, the differences between the original definitions and the definitions employed in this document are highlighted.

Chapter 2

Software Evolution

This chapter explains why evolution is important in modern software development and discusses problems that may occur when software is subject to evolution. The Reuse Contract model [Luc97], that has been conceived to address software evolution issues, is adopted in this work. This chapter presents collaboration contracts and reuse contracts as the abstractions that will be the target of the recovery processes discussed in subsequent chapters.

2.1 The Economic Relevance of Software Evolution

"A program that is used in a real world environment necessarily must change or become progressively less useful in that environment."

Lehman's first law of evolution [Leh84], [Leh97]

In a world where economic factors such as market share, turnover and profit define a company's success, the ability to respond rapidly to new market opportunities and trends is of major importance to a company. New opportunities present themselves continuously because people's taste for food, clothing, housing, toys, communication, transportation, leisure activities, travel, music, books, television, etc changes constantly. The inability to respond rapidly to new opportunities may cause a decline in market share; in highly competitive markets, it may even break a company.

Besides the incessant evolution in the market place, companies face the constant evolution in hardware, software, protocols, treaties, laws, regulations, standards, customs, etc, which may significantly affect their products and their product lines. Television broadcasters, for instance, see a change in their product line due to technological evolution: the introduction of interactive television and electronic programme guides. On top of having to solve the technological problems that induce the 'year 2000' problem, banks in the European Community face a change in their products due to new treaties and laws: they have to change all their products to cope with the upcoming change in currency.

Meanwhile, on the verge of the twenty-first century, software becomes the fundamental technology of the Information Age. Software pervades our lives. It is found in our houses, in many appliances, in our means of transportation, in the means of communication, etc. Without software, one would be unable to do many things one takes for granted nowadays. Software also pervades the way we do business. For some companies software is even their source of life. Their business depends on software so much, that they simply cannot exist without it.

Software not only drives the production process and structures the work flow within a company, it also allows companies to exploit technology they would never have dreamt of a few years ago. For instance, now that virtually everybody has access to the information superhighway, companies have clearly acknowledged that publishing their products on the World-Wide-Web should be an integral part of their advertising policy and their efforts concerning the dissemination of product information. Rather than being a cost reducer, software becomes a money generator.

Apparently, software is vital to our society and it is an important asset of our companies. So when a company wants to tap new markets successfully, or when it has to conform to new rules and regulations, or when it wants to catch up with technological advances, it is essential that the company's software evolves in parallel with its changing objectives and activities. Keeping the software in sync with the company's activities and its way of doing business is today's major challenge for software providers, be it the company's IT department or an external software provider. Software providers bear a huge responsibility to keep the software up-to-date with evolution that is both under and out of control of the company.

2.2 Software Evolution at a Technical Level

While software providers have to cope with changing requirements imposed by their clients, they also struggle with their own problems concerning the delivered software product and the employed process model to make that product, on a technical as well as on a managerial level.

Software evolution is not only required to meet new or extra functional requirements. Often it is also required to meet non-functional requirements, such as reusability, flexibility, adaptability, low complexity, maintainability, etc. Typically, this form of software evolution is not visible to the user of the software, but only benefits the software engineers in their future work. Software evolution is called for in any of the following situations [CHSV97]:

New insights in the domain. When software is being customised for different customers, several customisations may include concepts that may in fact be general concepts that should reside in the core of the software. Moving these concepts to the kernel benefits the reusability and the complexity of the software and it avoids code duplication.

Complexity of classes is too great. As object-oriented software evolves classes tend to become more complex. To reduce the complexity, the software engineer has several techniques at his disposal to redesign the software, ranging from introduction of new abstractions, such as abstract classes and decomposed methods, to refactoring [Opd92], [OJ93] and the application of design patterns [GHJV94].

New design insights. Some design issues are neglected or even forgotten in the initial design phase. These issues often turn into problems during later development. The software often has to be redesigned to improve algorithms and overall performance.

Although the economic relevance of software evolution is clear, the importance of software evolution is still underestimated by the software engineering community, as can be inferred from the fact that state-of-the-art object-oriented software engineering methodologies do not properly address software evolution issues [GR95].

2.3 Software Change Impact Analysis

A very important issue in software evolution is insight into the effects of a software change on the rest of the software. Making changes blindly may lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system.

Bohner and Arnold give the following definition of impact analysis in *Software Change Impact Analysis*, a collection of tutorial papers, trade articles, and software research literature that captures current impact analysis techniques and technical results [BA96b]:

> *''Software change impact analysis, or impact analysis for short, estimates what will be affected in software and related documentation if a proposed software change is made. Impact analysis information can be used for planning changes, making changes, accommodating certain types of software changes, and tracing through the effects of changes. Impact analysis provides visibility into the potential effects of changes before changes are implemented. This can make it easier to perform changes more accurately."*

The term *software life cycle object* (SLO) is used to denote any part of the software, such as the requirements, the analysis and design documentation, the source code and the test suites, that can be affected by a change. A change to the software may affect only one piece of the software, e.g. introduce undesirable behaviour in some place, or it may affect many other parts of the software system. The effect of a change may even affect yet other parts of the software. This propagation of changes is known as the *ripple effect*.

To identify the impact of a software change, the software engineer must have knowledge of the dependency relationships between the software life-cycle objects. Such dependency relationships can be represented by graphs in which each node is an SLO and each directed edge indicates that the SLO represented by the target node will be affected by a change in the SLO represented by the source node. Immediate impacts are found by collecting all target nodes of outgoing edges of the node that represents the changed SLO. In Figure 2.1, the immediate impacts of a change in SLO1 are all light-grey nodes. Ripple effects are found by computing the transitive closure, a fundamental impact analysis technique. The transitive closure of affected nodes is the set of all nodes reachable from the given node. In Figure 2.1, the transitive closure of the black node is the set of all grey nodes (light grey as well as dark grey).

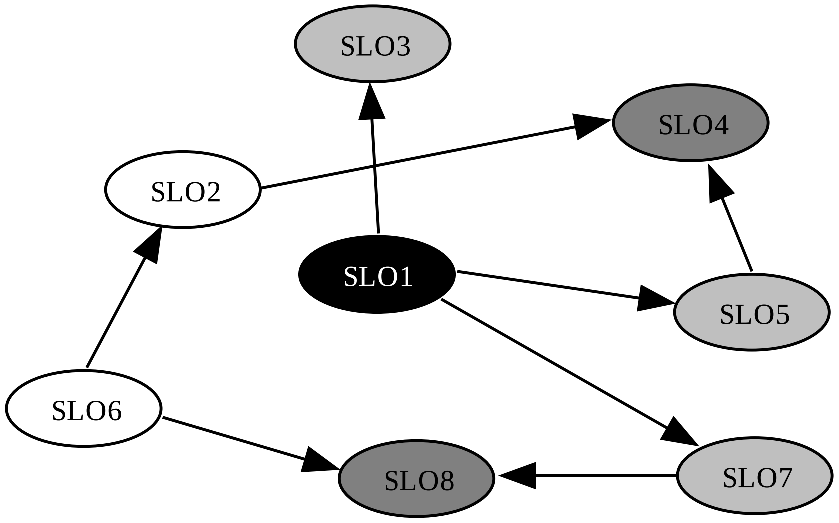


Figure 2.1: Dependency graph of software life cycle objects

The fact that two SLOs depend on each other does not necessarily mean that a change in one gives rise to a problem in another. So using a brute force transitive closure algorithm may not be the best option in all situations, as it returns all SLOs that can possibly be affected by a change. Sometimes, propagated changes may have no real consequences. Therefore, impact analysis methods use search algorithms that take more factors into account than just the plain dependency relationships, such as predetermined semantics of objects and relationships, heuristics that suggest paths to avoid, stochastic probabilities that determine the likelihood of the impact, or a combination of all these.

Although Bohner and Arnold state that impact analysis techniques are being used in source code analysis, metrics and CASE tools, documentation management systems, traceability systems, software project libraries and software repositories [BA96a], impact analysis is rarely encountered in widely used software development environments. In particular, there appears to be no commercial object-oriented development environment that features a mature impact analysis tool. Smalltalk environments are able to indicate where source code needs adaptation after removal of a class, but this hardly qualifies as a mature impact analysis feature. In many file-based systems, the only way to assess the impact of changes is recompilation and interpretation of the compiler errors. Impact analysis is thus predominantly a manual activity. It is labour intensive and ad-hoc, as can be concluded from the examples of impact analysis techniques currently in use [BA96a]:

- browsing (including cross-referencing);
- program slicing;
- using traceability relationships to identify software artefacts associated by a change;
- using configuration management systems to track and find changes;
- consulting software documentation to determine the scope of a change.

Automating impact analysis appears to be a major technical challenge. That is probably the reason why there do not exist many reports on impact analysis tools.

Ajila reports on the WHAT-IF tool [Aji95]. The WHAT-IF impact analysis tool has not been integrated in a development environment; Ajila presents a prototype that is able to determine the impacts of a change in the design documentation (HOOD = Hierarchical Object-Oriented Design) or in the implementation (ADA). Relationships between design entities and implementation entities, on the same level as well as in between levels, are extracted from the HOOD and ADA specifications, and fed into a knowledge base that can be queried. A query has the form WHAT-IF-ADA(object-name, change-type) or WHAT-IF-HOOD(object-name, change-type), where object-name is an identifier and change-type is one of the following: parameter change type, type definition change, and assignment statements change type for ADA, and object mode change type and operation constraint change type for HOOD. The result of the query is a report of the object's visibility and the impacts and propagation of the change. The latter is no more than a list of affected objects (modules, variables, etc). Although Ajila states that the results are encouraging, it appears that the WHAT-IF impact analysis tool supports a limited list of change types. On top of that, it does not give an indication of how impacts can be eliminated.

Collofello and Orn report on the ASU software maintenance environment [CO88] for compiler-error-free Pascal code, that features the Ripple Effect Analyser. The ASU tool facilitates detection of direct and indirect ripple effects. A direct ripple effect is defined as the effect of a change to one variable on the definition of another variable. An indirect ripple effect refers to the effect on yet other variable definitions. The authors state that semantic knowledge on top of syntactic knowledge is required to determine which code sections are truly affected by a change. However, they do not explain how semantic knowledge is derived from the code and used by the ripple effect analyser. User-supplied semantic filters can be used to stop ripple effects from propagating through undesired modules.

2.4 Software Evolution vs Software Maintenance

''Programs, like people, get old. We can't prevent ageing, 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." [Par94]

The IEEE Standard Glossary of Software Engineering Terminology states [IEE91]:

''Software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a change in environment."

Software maintenance has undergone major changes in interpretation and importance during the 45 years of modern computing. In the early days, writing new software was the major activity and therefore little software maintenance was involved. In the late sixties and early seventies, software maintenance started to be recognised as being important to keep software from aging. In the eighties, it became clear that the old architectures from the previous two decades restricted new development. The terms "legacy software" and "legacy systems" (see Section 2.5) were introduced. Studies show that software maintenance eats up a large part of the total life cycle costs, making software maintenance a major commercial and economic factor. In sharp contrast to the situation of the early days, where revolutionary change was typical, evolutionary change is characteristic of software development in the nineties, where the business needs of many organisations are covered in software, but where enhancement and evolution are necessary to accommodate business change [Ben97].

Four types of software maintenance can be identified [LST78], [Ben97]:

Corrective maintenance. Corrective maintenance is concerned with the identification and removal of software defects to keep the software operational. Corrective maintenance consumes less than 10% of all resources [Art87].

Adaptive maintenance. Adaptive maintenance accommodates changes in the processing environment, such as new versions of the operating system, new hardware to run the software on, database changes, but also changes in the real world, such as new or changed rules, laws and regulations that affect the software.

Perfective (or evolutive) maintenance. Perfective maintenance concerns functional changes that result from new and changing user requirements, performance optimization with respect to execution time and storage requirements, and activities such as restructuring and redocumenting the software. Perfective maintenance accounts for 60 to 70 percent of the total maintenance cost [BA96c].

Preventive maintenance. Planned changes made to software to make it more maintainable.

Corrective maintenance is the only maintenance activity that focuses on fixing software defects. Corrective, adaptive, perfective and preventive maintenance are actually software evolution activities.

At this point in time, software maintenance and software evolution appear to refer to the same activity in the software life cycle. They share the same economic, organisational and technical concerns. The only difference between the two is their flavour: software maintenance has a negative connotation, implying that something is wrong with a software product [Art87], while software evolution has a positive ring to it, suggesting a bright future with many possibilities.

2.5 Legacy Systems

A legacy system is considered a very old software system that has been subject to many modifications. It is usually very large, it is supported by a team of software developers and it is typically based on older technology including out-of-date programming languages [Ben97].

This conventional view on legacy systems can be generalized. New software systems using the latest technology may become legacy systems as soon as they are delivered to the customer: if new software is unable to evolve, it is legacy software from the start. Therefore, software can be considered legacy software when it cannot evolve, regardless of its age, the underlying technology, or the number of changes it has already been subjected to.

2.6 Software Evolution Problems

In his *Principles of Software Engineering* [Dav94], Davis states:

> *"Any system that undergoes continuous change will grow in complexity and will become more and more disorganised".*

This problem has also been acknowledged by Lehman, who wrote it down as his second law of evolution [Leh84], [Leh97]:

> *''As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure."*

The continuous increase in complexity may even turn the software into legacy software when the complexity reaches a point at which evolution becomes too expensive or even unfeasible. Controlling the complexity of a software system is therefore essential for the system's future.

Many causes give rise to increased software complexity and therefore hinder its control, including the following problems that occur during software development.

2.6.1 Version Proliferation

Version proliferation occurs when two or more versions of the software exist and when they are changed separately without feedback from the one to the other, giving rise to multiple versions of the software existing in parallel (V1' and V1" in Figure 2.2). Version proliferation is typically the result of copy and paste reuse, a kind of reuse whereby a copy of the original software is adapted. The adapted copy and the original are not related anymore, which may give rise to severe maintenance problems. After all, it is difficult to maintain the different versions, since bug fixes need to be made to all versions. Note that version proliferation may occur at several levels, ranging from copies of entire modules or subsystems to duplicate methods.

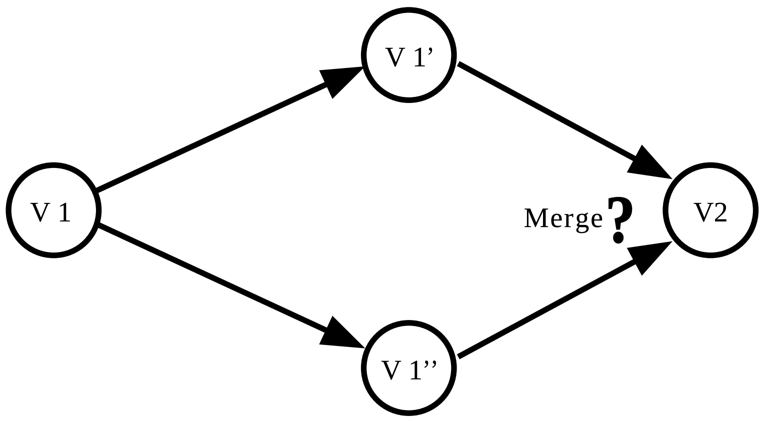


Figure 2.2: Version proliferation

When version proliferation is identified, software engineers try to solve it by merging the different versions, possibly by refactoring [OJ93] the software. When the dependencies between the affected parts and the rest of the software are unknown, merging two (or more) versions of a piece of software may not be easy. Moreover, merging different versions may even be impossible due to conflicting implementations. Merging is usually done by carefully browsing and comparing the different versions of the software to detect the commonalities and the differences and to identify the dependencies between the versions and the rest of the software. Therefore, solving version proliferation is in general a manual activity that tends to be very labour-intensive.
Version proliferation may also result from the *not-invented-here syndrome* [SF97]. It is an I-can-do-it-better/faster-myself way of thinking, which is a major inhibitor to reuse, a source of code duplication, and a waste of resources.

2.6.2 Poor Effort Estimation

When a software system needs to evolve according to new or changed specifications, software engineers (and their managers) like to know how much work (and time) this evolution will take. In particular, a software engineer likes to assess the number of classes that needs to be changed in order to meet the specifications. The number of classes gives an estimate of the required effort, but it is seldom a good estimate. Dependencies between the affected classes may significantly increase the work involved, because a change to a class may propagate through a large part of the software.

2.6.3 Poor Insight into the Effect of Changes

Effort estimation relies on a good assessment of the effect of changes. An impact analysis tool to identify such effects would help a lot to get a better effort estimation. Impact analysis tools are seldom found in object-oriented software development environments, however. In absence of impact analysis tools, a good understanding of the software is essential to make a fair assessment of the required work. Understanding the dependencies between classes is crucial to perform an impact analysis manually.

2.6.4 Architectural Drift

Defining a mature software architecture is one achievement, actually enforcing its use is another. Most software development environments are not equipped with tools to express designs and architectures. Therefore, it is not surprising that software development environments are unable to force software engineers to create software that conforms to a desired architecture, or indicate when the architecture is being violated. Consequently, the prevention of architectural drift relies entirely on the software engineers' good will to follow the software architecture.

In spite of all good will, a software architecture is often breached. Architectural drift typically occurs because the software engineer is ignorant of an imposed software architecture due to little or no documentation on the subject. Because of the non-integration of the development environment with the documentation environment, a software engineer is compelled to look into the source code instead of the documentation, which leads to an incomplete mental picture of the software. In a working environment where it is known that the documentation is incomplete or not up-to-date, the software engineer may not even bother to have a look at the documentation. The result is that the software engineer's work drifts away from the intended software architecture.

The situation described above gets worse under *deadline pressure*. Since time is a scarce resource in such a situation, normal working rules are bypassed, often resulting in software that deviates from the target software architecture. Ruling out deadline pressure is almost impossible considering the stresses and strains of modern business, so uncontrolled software evolution due to deadline pressure is unavoidable and must be reckoned with in the development process. The problem is, however, that after delivery of the software and certainly after a missed deadline, there is usually no time to reduce the extra complexity, because other work is piling up. So software that is subject to uncontrolled evolution may very well remain in a state that is unfit for evolution, which is a dead end street called "legacy software" (see Section 2.5).

Cookbooks have been proposed as a solution to architectural drift, but usually cookbooks are too constraining. They provide only a limited number of predefined ways to reuse software. Consequently, undocumented opportunities for reuse are not spotted, which may give rise to design violations and code duplication. Cookbooks have to be kept up-to-date with the framework that they serve. Keeping cookbooks up-to-date is similar to keeping software documentation up-to-date: it is an activity that seldom gets enough attention.

Current OOA/OOD methods and notations do not provide a solution to architectural drift either, because the notations are unable to express how software can be reused in a disciplined fashion, i.e. without violating its architectural design.

2.6.5 Overfeaturing

In the production of off-the-shelf software, there is a tendency to add as much features as possible. When the software provider indulges the users with all extra or enhanced features they request, the software product results in a product full of features useful for some users, but useless, and often unwanted, for other users. The tendency to overfeaturing has been clearly seen in the evolution of word processors, by increasingly resembling desktop publishing applications, and in the all-in-one Internet applications that combine several stand-alone applications. Applications with a lot of whistles and bells are typically large and come with a considerable collection of user adjustable options. The abundance of features, and the dependencies that come along with them, makes evolution of the software harder.

2.7 The Importance of Class Collaboration

An object-oriented program is defined as a set of interacting objects. Objects interact by sending messages to each other. This definition clearly emphasises the importance of object interaction, and therefore class collaboration in class-based object-oriented programming.

The exploitation of class collaboration is essential for building reusable and adaptable object-oriented software. Reusability and adaptability largely result from the combination of class collaboration (delegation) and inheritance. By distributing behaviour among different classes, the different classes can be reused separately and classes can be composed in new and interesting ways.

The distribution of behaviour among different classes promotes fine-grained reuse, adaptation and composition, but it introduces dependencies between classes that would otherwise be encapsulated in one class. It introduces acquaintance relationships and invocation dependencies between classes. So when a software engineer wants to reuse a class, he is actually reusing the acquaintance relationships and the invocation dependencies the class is involved in.

In order to reuse or adapt a class properly, the software engineer should be aware of those dependencies. The object-oriented programming paradigm does not provide the necessary concepts to capture those dependencies, however. So although information on class collaboration is crucial to reuse or adapt a class (or a set of classes), there is no way to capture how classes collaborate to achieve some behaviour. The resulting lack of insight into the collaboration between classes is a major source of the software comprehension problems experienced by software engineers.

Contracts [HHG90], frameworks [JF88], design patterns [GHJV94], adaptive programming [Lie96], separation of concerns [HL95], component-oriented programming [Ude94] are well-known models and techniques that address this problem. While class, inheritance and polymorphism are still the fundamental concepts, the paradigm has been enhanced with concepts that build upon these concepts. Collaborations between classes, reuse and evolution are the central themes in these enhancements.

Although some of the newly introduced concepts have been successfully used in the design and implementation of object-oriented software systems, they have not found their way into popular software development environments. It is not clear why. Maybe because the suggested models are too cumbersome, maybe because the models simply do not mix well with the way of working imposed by development environments, or perhaps because there is no formal model for these concepts, which makes integrating them into a development environment difficult, if not impossible.

2.8 Collaboration Contracts

The Reuse Contract Model [Luc97] has been conceived to provide a means to document software, software reuse and software evolution, and to provide support for change propagation. The model has two main concepts: collaboration contracts and reuse contracts. An overview of collaboration contracts and reuse contracts is given in this and the next section. Appendix A includes all formal definitions.

Collaboration contracts[1] [Luc97] were conceived as structured documentation for the description of class collaboration. A collaboration contract is a formal description of software that is still close to the source code. This quality makes collaboration contracts very suitable to be used in a software development environment. The balance between formality and closeness to the source code is crucial for this work, as it is concerned with the creation of abstractions of object interactions found in the source code.

[1]Avoid misinterpretation based on earlier work on Reuse Contracts! Note that the term collaboration contract used here corresponds to the term reuse contract, as defined in Lucas' work [Luc97]. The term reuse contract used in this document refers to reuse of a collaboration contract. See Appendix A for the reasons why the terminology has been changed.

A collaboration contract formally describes collaboration between participants. Participants are software entities that have an interface and that invoke each other's operations. This means that participants can represent classes, conglomerates of classes that define a software component, or even large architectural building blocks, such as software layers, modules, etc. In the context of this work, participants represent classes and the operations are methods.

A collaboration contract has two aspects: the static structure and the interaction structure. The static structure refers to the participants and the acquaintance relations between them. The interaction structure refers to the messages (method invocations) sent between the participants. In the current state of the model, a collaboration contract is no more than a named set of participants. The static structure and the interaction structure are modelled by participant descriptions. This is reflected in the following definition, which is a combination of several definitions from the Reuse Contract model [Luc97] (also summarised in Appendix A).

Definition 1 (Collaboration contract)

> A **collaboration contract** consists of a name and a set of participants. Each **participant** has a unique name within the collaboration contract, an **interface** holding methods and an **acquaintance clause** holding acquaintance relationships. A **method** has a **method signature**, an annotation *abstract* or *concrete*, and a **specialisation clause**. A specialisation clause is a set of **method invocations**, each associating an acquaintance name with a method signature. An **acquaintance relationship** is an association between an acquaintance name and a participant name.

Collaboration contracts must be *well-formed*. A well-formed collaboration contract only references existing methods, acquaintances and participants.

A collaboration contract has a graphical representation. There are two variations. The first variation separates the static structure diagram and the interaction structure diagram. The second variation combines the two diagrams.
The collaboration contract depicted in Figure 2.3 and Figure 2.4 describes how participant `controller` (a `PlannerController`) and participant `command` (a `PlannerCommand`) collaborate. When the `controller` receives the message `executeCommand:`, it sends a message `execute` to participant `command`, with which it is acquainted. The `PlannerController` refers to the `PlanningCommand` with the acquaintance name `theCommand`. The notation {`executeCommand: invokes`} `execute` on the association link denotes that method `executeCommand:` sends the message `execute`. It reflects that a method invocation `theCommand.execute` is part of the specialisation clause of method `executeCommand:`[2].

[2]The notation with the braces is chosen to use a diagram notation that is in accordance with the UML. It is a UML *constraint* [OTI97].

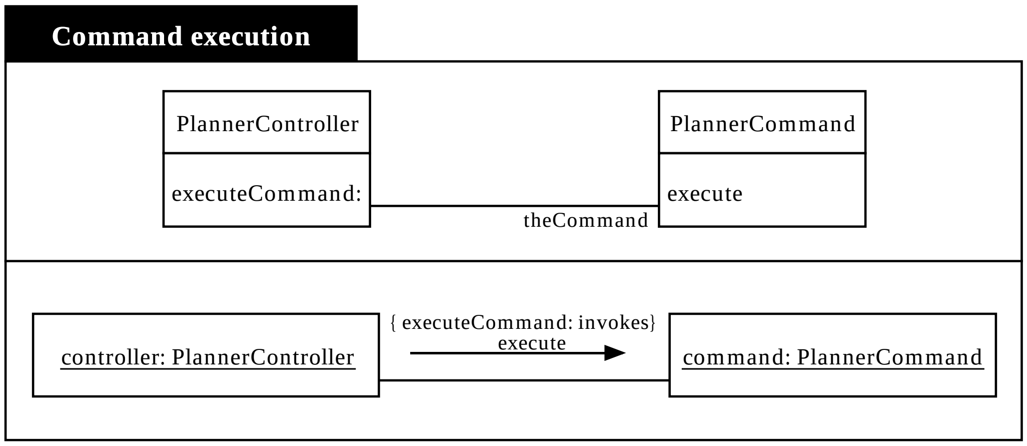


Figure 2.3: Collaboration contract diagram (separated static and interaction structure)

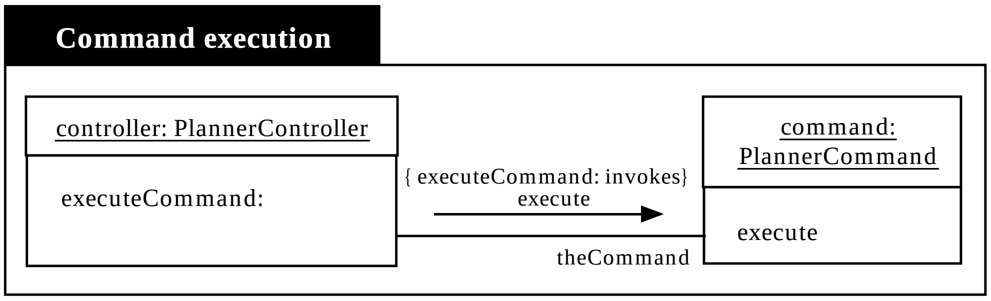


Figure 2.4: Collaboration contract diagram (combined static and interaction structure)

2.8.1 Not a Class Diagram, not a Sequence Diagram

A collaboration contract differs from a class diagram in three respects:

1. A collaboration contract holds participants, not classes. Each participant defines the role its corresponding class plays in a collaboration with other participants (including itself).
2. A collaboration contract includes the interaction structure of its participants. The interaction structure states the method invocations from one participant to another. A class diagram does not include an interaction structure.
3. A collaboration contract does not include inheritance relationships.

Besides the limitations mentioned above, in the current state of the Reuse Contract model a collaboration contract also does not state specifics about acquaintance relationships. Multiplicity and information on the kind of acquaintance relationship (aggregation, association, etc) are not recorded in a collaboration contract.

A collaboration contract is not a sequence diagram. A method invocation in the specialisation clause of a method denotes that the method can be invoked; it does not mean that it will be invoked, and it certainly does not say when it will be invoked. In its current state, control-flow aspects are thus neglected by a collaboration contract. This contrasts with UML collaboration diagrams or sequence diagrams, which state the order in which methods are invoked.
Another difference between a collaboration contract and a sequence diagram is that the former states the method that performs a message send. The latter only states that the message is sent from one object to another; no specifics about the sending method are given.

2.8.2 Classes Play a Role in Many Collaboration Contracts

Usually, the behaviour of a class has many aspects. Each aspect corresponds to a role the class plays in a collaboration with other classes.

In the example of the `PlannerController` and the `PlannerCommand`, both classes may also be involved in a collaboration contract that relates to undoing. Assume that the `PlannerController` keeps a history of the commands that it has already executed (this was not specified in the collaboration contract of Figure 2.3). When the `controller` receives an `undo` message, it fetches the last command from the command history by sending `commandForUndo` to itself, and sends participant `command` an `unexecute` message. Figure 2.5 shows the collaboration contract.
The collaboration contract does not state any specifics about the method `commandForUndo`, as it does not contribute to the undoing of commands. It is only concerned with the com-

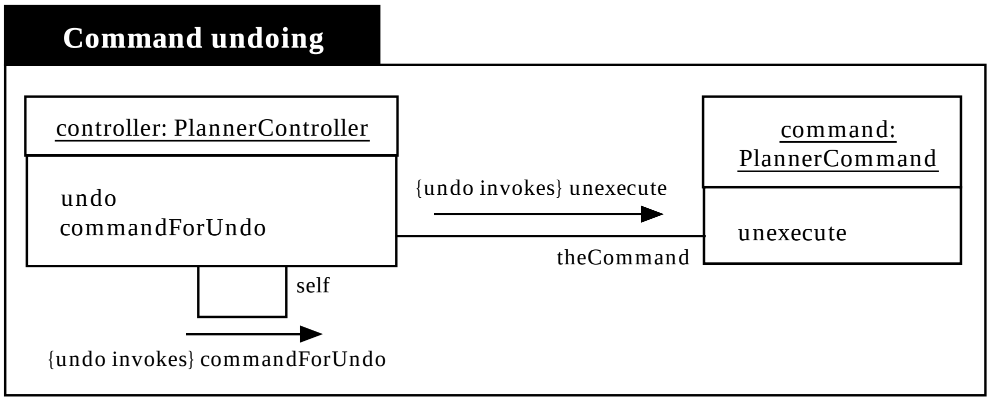


Figure 2.5: Classes play a role in several collaboration contracts

mand history.

The fact that a class may participate in several collaboration contracts implies that participants in collaboration contracts do not correspond to whole classes, but instead are views on classes that carry but a partial interface of their corresponding class. Furthermore, it implies that a collaboration contract is a view on the global interaction structure in which the collaboration contract's participants (and thus their corresponding classes) are involved.

2.8.3 A Method May Play a Role in Many Collaboration Contracts

Since classes may play a role in many collaboration contracts, a method may play a role in more than one collaboration contract too. This typically happens when a method addresses several aspects of a class' behaviour.

In the example, method `executeCommand:` in class `PlannerController` is part of more than one collaboration contract. It plays a role in 'Command execution' (see Figure 2.3), and it plays a role in setting up a history of executed commands to be able to undo commands. Figure 2.6 shows the collaboration contract.
In the collaboration contract of Figure 2.3, the method `executeCommand:` has a specialisation clause that contains method invocations related to executing (`execute`). The specialisation clause in this collaboration contract includes method invocations related to maintaining a command history (`addToHistory:`).

2.8.4 Classes are the Sum of Roles

As shown in Section 2.8.2, a participant can occur in several collaboration contracts, each describing another collaboration in which the corresponding class plays a role.

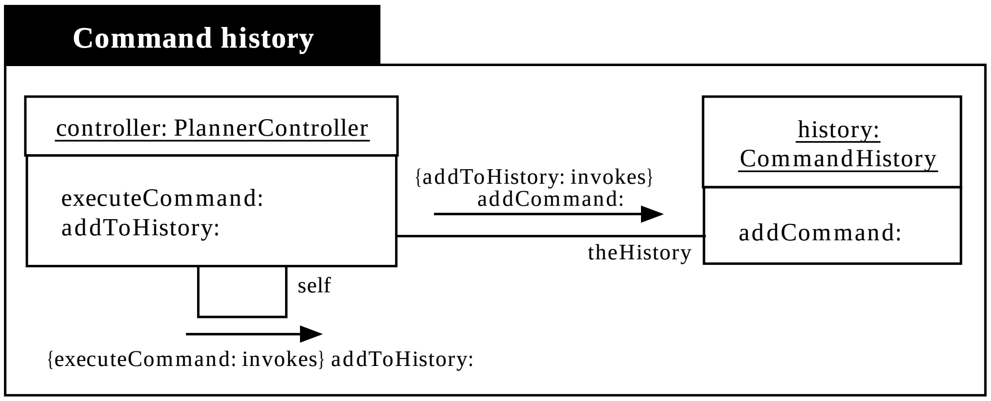


Figure 2.6: Methods play a role in several collaboration contracts

In principle, when *all* collaborations of a class with other classes are described by collaboration contracts, the union of the interfaces of the corresponding participants in the different collaboration contracts should be the complete interface of that class. If the union is not equal to the interface of the class, either some collaborations of that class with other classes and with itself are not captured by the collaboration contracts, or either the class has methods that do not play a role in a collaboration with other classes or with itself (which probably means that those methods are obsolete and unused).
Note the stress on *in principle*. In practice, not all collaboration contracts are assembled. Usually the most important ones are enough to get a good understanding of how classes collaborate.

In the example, when all collaboration contracts are put together, the resulting collaboration contract, with the united interfaces of the participants, looks like depicted in Figure 2.7. For completeness, a method `lastCommand` for `CommandHistory` and its invocation by `commandForUndo` are added.

2.9 Evolution Model: Reuse Contracts

In general, a *reuse contract* is a contract between a provider and a reuser. It comprises a *provider clause*, that states what is actually provided, a *reuser clause*, that states what is actually reused, and a *contract type*, that states how the contents of the provider clause is reused.

2.9.1 Reuse of Collaboration Contracts

For object-oriented systems, the provider clause of a reuse contract contains a collaboration contract (see Appendix A for details). A reuser clause describes the changes made to the provider clause, that is, to the collaboration contract. The description depends on the contract type, as the contract type lays down how the reuser clause should be interpreted

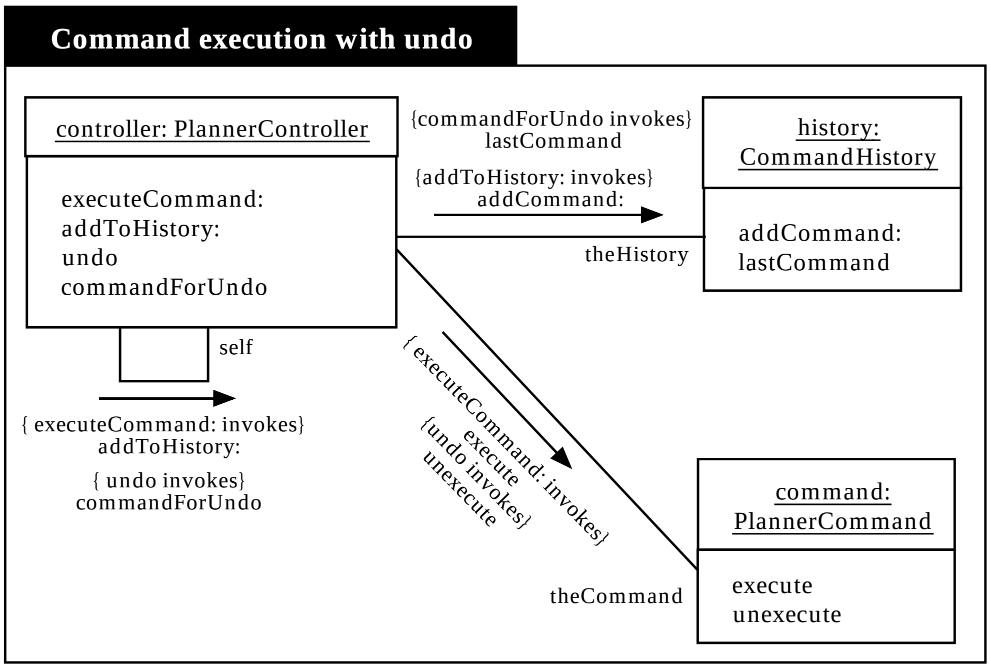

Command execution with undo
controller: PlannerController
executeCommand:
addToHistory:
undo
commandForUndo
self
{ executeCommand: invokes} addToHistory:
{ undo invokes} commandForUndo
{commandForUndo invokes} lastCommand
{addToHistory: invokes} addCommand:
theHistory
history: CommandHistory
addCommand:
lastCommand
{ executeCommand: invokes} execute
{undo invokes} unexecute
theCommand
command: PlannerCommand
execute
unexecute

Figure 2.7: Combined collaboration contract - classes are the sum of roles

and how it should be applied on the provider clause to obtain the resulting (or derived) collaboration contract. There exist several contract types, each defining a different kind of reuse (see Table A.1 on page 201): addition or removal of participants, methods, acquaintance relationships, and method invocations, and changes in the abstractness annotation of a method. These *basic reuser clauses* can be grouped in *combined reuser clauses* to describe any kind of adaptation of collaboration contracts.

For example, assume that the collaboration contract for `PlannerCommand` and `Planner-Controller` in Figure 2.3 is reused (adapted) for a subclass `ResizeCommand` in order to specify what execution means for `ResizeCommands`. Figure 2.8 shows the collaboration contract. The interface of participant `command` is extended with two methods: `resizeTo:` and `newSize`. The `execute` method invokes both of them. In the original collaboration contract, `command`'s execute method performs no method invocations.

The reuse contract that describes how the original collaboration contract is reused to produce this collaboration contract is shown in Figure 2.9.

The provider clause contains the original reuse contract. The contract type is a combined contract type: context refinement, participant extension and participant refinement. In accordance with the combined contract type, the reuse contract has a combined reuser clause. The first reuser clause is a context refinement that describes the addition of the acquaintance relationship named `self` on participant `command`. The second reuser clause is a participant extension that describes the addition of the two extra methods in the

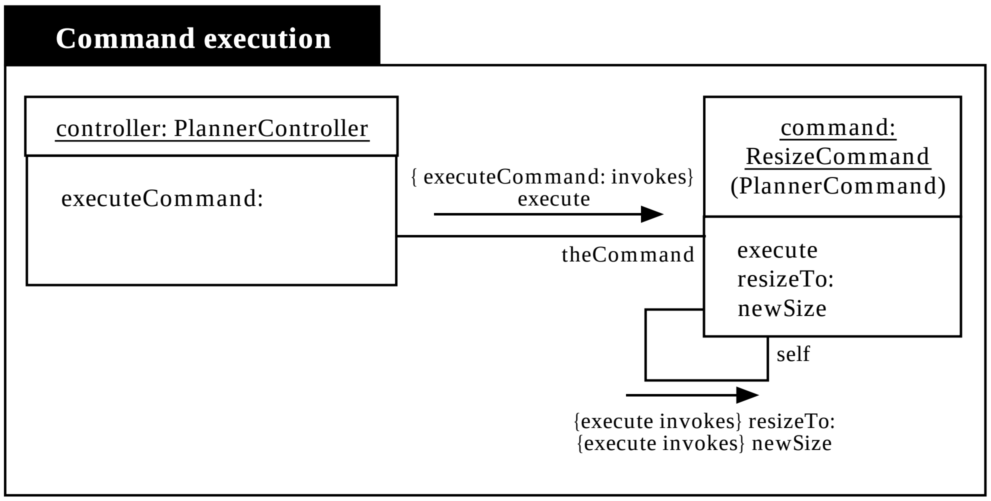

Command execution
controller: PlannerController
executeCommand:
{ executeCommand: invokes} execute
theCommand
command: ResizeCommand (PlannerCommand)
execute
resizeTo:
newSize
self
{execute invokes} resizeTo:
{execute invokes} newSize

Figure 2.8: Adapted collaboration contract

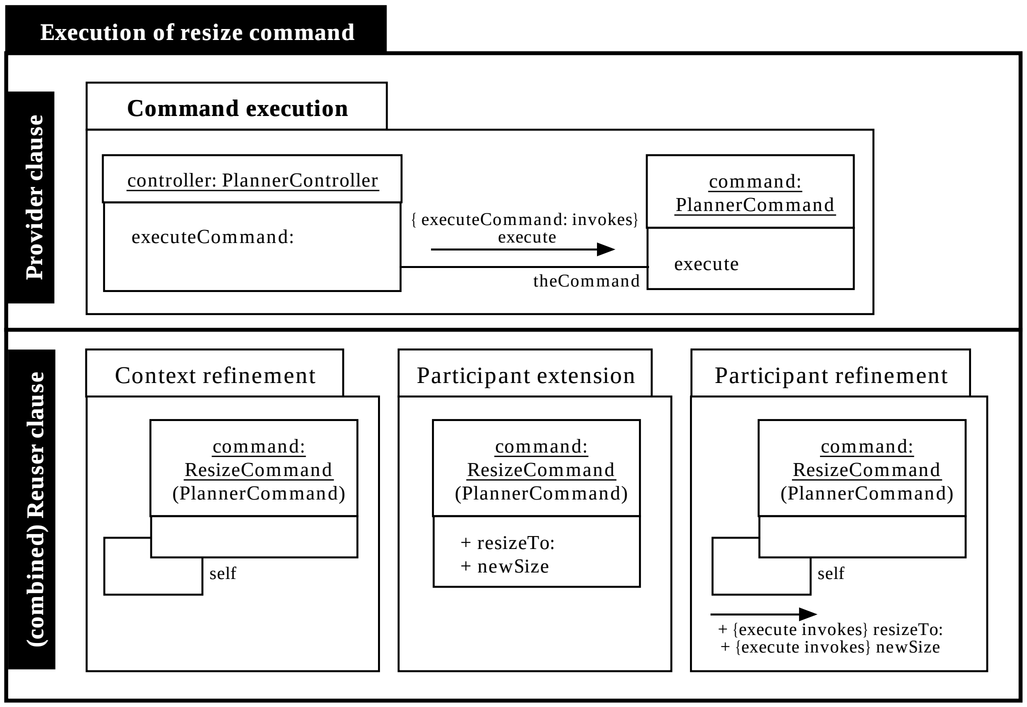

Execution of resize command
Provider clause
Command execution
controller: PlannerController
executeCommand:
{ executeCommand: invokes} execute
theCommand
command: PlannerCommand
execute
(combined) Reuser clause
Context refinement
command: ResizeCommand (PlannerCommand)
self
Participant extension
command: ResizeCommand (PlannerCommand)
+ resizeTo:
+ newSize
Participant refinement
command: ResizeCommand (PlannerCommand)
self
+ {execute invokes} resizeTo:
+ {execute invokes} newSize

Figure 2.9: Reuse contract diagram

interface of participant `command`. The third reuser clause is a participant refinement that describes the addition of two extra method invocations, performed by the `execute` method.

2.9.2 Evolution of Collaboration Contracts

The reuse contract model employs the same technique for documenting software evolution as for documenting reuse. When part of a software system is subject to changes, these changes can be seen as changes to the providers of the reuse contracts that describe the software system. This means that reuse contracts can be set up to describe the evolution of a software system. An evolution reuse contract has an existing collaboration contract as provider clause and its reuser clause describes the evolution changes.

2.9.3 Impact Analysis

The impact of software changes can be analysed by correlating the reuse contracts that describe reuse with the reuse contracts that describe the evolution. The Reuse Contract model states how the contract type – reuser clause pairs, one describing reuse and another describing evolution, should be compared to determine the impact of the changes. The model comes with a table of impacts that may arise during evolution [Luc97].

2.9.4 The Reuse Contract Model Attacks Evolution Problems

The Reuse Contract Model addresses the evolution problems stated in Section 2.6 as follows.

Version proliferation. The problem of version proliferation will not be solved by reuse contracts. However, reuse contracts are helpful when different versions of software are merged, but this is only possible when reuse contracts for different versions are available. The development environment can perform some conflict detection to check whether the changes made to produce the different versions do not clash with each other. Checking can be done automatically according to the conflict detection rules laid down by the theoretical reuse contract model. A merge is successful if no conflicts are detected.

Effort estimation. According to the theory, reuse contracts can be used to estimate the effort needed to make a change to the software. Collaboration contracts state the (invocation) dependencies among classes. A development environment is able to indicate the *minimal* set of classes that must be looked at and the *minimal* set of methods that should be examined when a planned change is actually made. The software engineer gets a rough estimate of the minimal amount of work it will take to make the change.

Insight into the effect of changes. Insight into the effect of changes is one of the key features of the reuse contract model. Conflict detection can be used to provide answers to 'what if' questions. A proposed change to the software can be looked at as a change that should be merged with the software. The conflict detection process is able to return a list of conflicts (effects) that would be introduced when the proposed change would actually be made. That way the software engineer gets insight into the effects of a change under consideration.

Architectural drift. Architectural drift seems to be a problem that must be solved by imposing the strict discipline on the development team not to deviate from the software architecture. However, when collaboration contracts are available in the software development environment, they can be employed to ensure that the software architecture described by the collaboration contracts is not violated by the developer. Checking violations can be done in a Draconian way, each time a change to the source code is made, but that would probably be too constraining for flexible software development. An alternative is to postpone checking of architectural violations until the software engineer has finished his work. The batch of applied changes can then be checked at once.

Overfeaturing. Overfeaturing is a problem that cannot be solved by employing reuse contracts. After all, the set of features that should be incorporated into the software is defined by the customers and cannot be described with collaboration contracts or reuse contracts.

2.9.5 The Reuse Contract Model as-is is not the Solution

Although the Reuse Contract Model addresses most of the aforementioned problems, the reuse contract model in itself does not solve them completely. First, integration of the model in a software development environment is essential for successful application of the model. Second, the model does not specify how collaboration contracts and reuse contracts should be set up for a piece of software. Third, the reuse contract model cannot handle a large amount of classes. This scalability problem cannot be solved by the reuse contract model. The following paragraphs discuss these issues in more detail.

Tool Support is Indispensable Reuse contract documentation on paper is helpful to describe a software design, but as such, it cannot play an active role in the development process. Documentation based on reuse contracts must be exploited in the software development environment to realise its full potential. Tool support is required for:

- setting up reuse contract documentation in the form of collaboration contracts and reuse contracts;
- browsing and exploring of collaboration contracts and reuse contracts;
- supporting reuse;
- supporting evolution, including impact analysis.

No Method for Setting up Collaboration Contracts and Reuse Contracts The reuse contract model defines the necessary concepts to document class collaboration, reuse and evolution, but it does not provide any information on how collaboration contracts and reuse contracts should be used for documentation purposes. Since setting up collaboration contracts and reuse contracts is essential for all further use of reuse contracts, it is crucial to have a method, or at least some guidelines, for software documentation with reuse contracts.

Inability to Handle a Large Amount of Classes and Changes The reuse contract model has been conceived to document class collaboration, reuse and evolution. In the course of developing the theory, little or no attention has been paid to scalability issues. The theory has been validated in experiments with a small number of classes. In practice, however, a software system consists of hundreds of classes, and each class plays a role in several collaboration contracts. On top of that, software is subject to a large number of changes. In order to handle the large amount of classes, collaboration contracts, and software changes, some kind of structuring to organise these entities would help.

2.10 Summary

Software evolution is crucial to keep up with the evolution of the society we live in. The inability to evolve may even break the organisation that relies on the software.

Preserving software from ageing is not easy. Poor effort estimation and poor insight into the effects of changes make it hard to evolve software correctly. The development process itself may even give rise to software evolution problems such as version proliferation, architectural drift, and overfeaturing. The Reuse Contract model has been conceived to address software evolution issues. The model has two major concepts: collaboration contracts and reuse contracts. Collaboration contracts describe how participants collaborate. They are close to the source code, which makes them very suitable for integration in a software development environment. Reuse contracts document how collaboration contracts are reused and evolved. The Reuse Contract model comes with a conflict detection scheme that can be used to analyse the impact of changes.

This dissertation adopts the reuse contract model. Collaboration contracts and reuse contracts are the target of the recovery processes described in subsequent chapters.

Chapter 3

Case Study: Evolution in Framework-Based Development

Following the industry-as-laboratory approach [Pot93, Gla94, Gla96, Par95], a large part of the research reported on in this document was conducted and validated at a site in industry. This chapter starts with a description of the context in which this research took place. After a brief description of the software, the process model and the software development environment are presented. Next, the evolution problems discussed in Chapter 2 are reconsidered in that context. A discussion on the role of the case study in this work concludes this chapter.

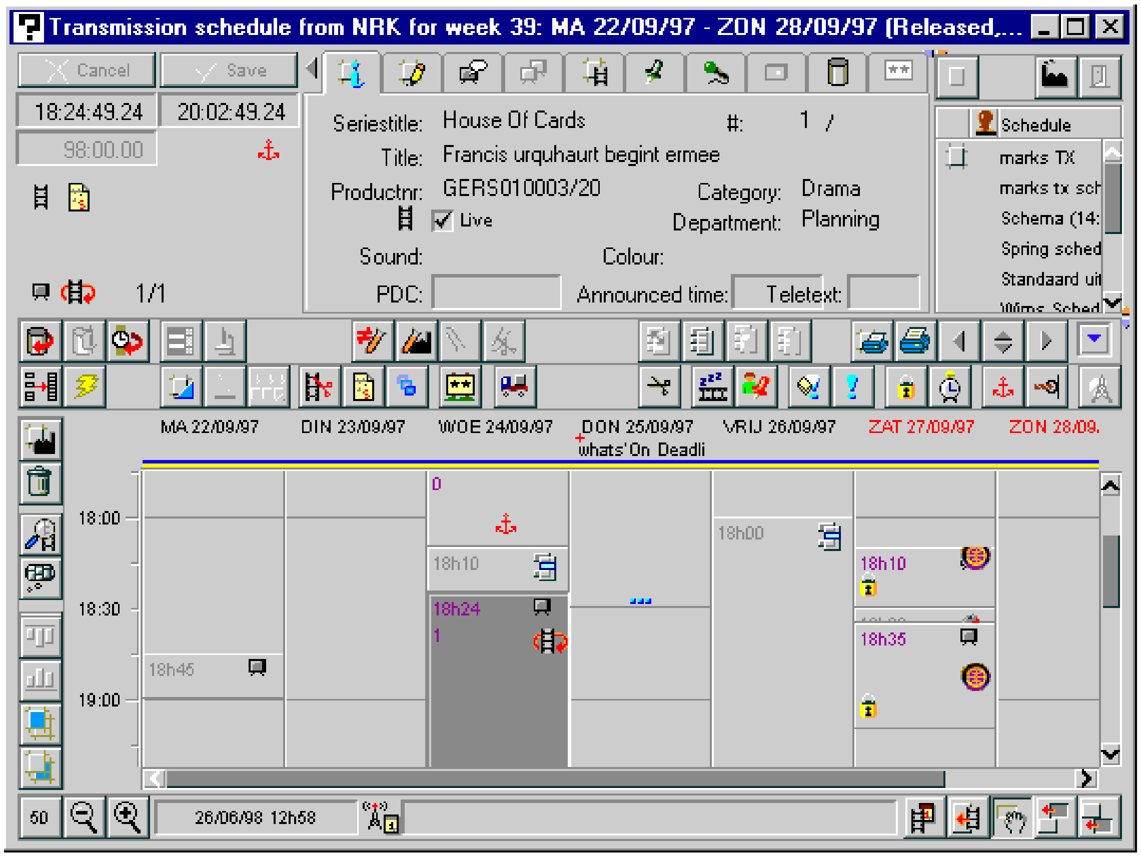
3.1 Context

3.1.1 Software

The subject software is an integrated broadcast management system for television and radio stations. The system is a groupware application. Employees from different departments work together to set up a broadcast planning in a top-down fashion, from a coarse-grained planning of a whole season, down to a fine-grained planning of the different tapes that are used to broadcast a programme at any point in time during that season. Figure 3.1 shows the week planning application.
The current state of the broadcast management software is the result of incrementally building the software for more than six years, and customising it for several television and radio stations across Europe.
A large part of the broadcast management system concerns broadcast planning. Although there are many similarities in the complex broadcast planning processes of different stations, a standard product would satisfy only 70% to 80% of the needs of a particular station. That is why the software provider has formulated the following goal: "Offer a broadcast planning solution that is highly and efficiently customisable to the needs of different television stations and that gives the customer the feeling of a custom system with the qualities of a standard product." [CHSV97]. In order to realise this goal, framework

technology was adopted as the core technology.

The core of the broadcast planning software is a framework that captures the software provider's knowledge of the broadcasting business. The framework is a representation in terms of variations and commonalities of the broadcasting domain at a given point in time. The commonalities describe the parts of the domain that are the same for all customers (television and radio stations). The variations describe the parts of the domain that are different for each station. The framework represents the knowledge at a given point in time because the framework is never finished: the introduction of a new station may result in an update of the domain knowledge and consequently in an update of the variabilities and the commonalities.


Transmission schedule from NRK for week 39: MA 22/09/97 - ZON 28/09/97 (Released,...

18:24:49.24 20:02:49.24
98:00.00

Seriestitle: House Of Cards #: 1 /
Title: Francis urquhaurt begint ermee
Productnr: GERS010003/20 Category: Drama
[✓ Live] Department: Planning
Sound: Colour:
PDC: Announced time: Teletext:

1/1

26/06/98 12h58

Figure 3.1: The broadcast planning application (week view)

3.1.2 Development Style

There are two kinds of software being created today: off-the-shelf software and custom software. The broadcast management software is a combination of these two kinds of software.

Off-the-shelf software includes horizontal software, i.e. software applicable in a wide variety of application domains, such as operating systems, software development environments, and office applications, and vertical software, i.e. domain-specific software, such as desktop publishing applications, and scientific calculation tools. Booch recognises three common characteristics of most off-the-shelf software [Boo94]: the problem domain is usually well defined, improvements are driven by market forces, and over time off-the-shelf software becomes a commodity market. Tackling an off-the-shelf domain requires a significant investment, in development as well as in marketing, support and maintenance. Off-the-shelf software is developed in-house.

Custom software includes strategic, enterprise-specific applications for a particular line of business. Examples are financial software, medical software, and software to control an underground transportation system. Booch describes three common characteristics of custom software [Boo94]: it is generally designed along specific vertical business lines, it is rarely fully custom, and it is very difficult to develop compared to off-the-shelf software because of the unknown. The unknown results from ill-defined requirements. End users often cannot clearly express what they want, let alone understand what is possible. Custom software is often developed at the customer's site, on a project-by-project basis. That is why custom development is also referred to as project development.

The broadcast management software is off-the-shelf software, in the sense that a product (the framework) is offered, and it is custom software, in the sense that the framework is customised according to a station's specific needs. The development style that combines off-the-shelf (product) and custom (project) development is referred to as *framework-based development*.

3.1.3 Process Model

The duality of framework-based development must be reflected in the process model, because product development and project development require different skills. Framework engineers are responsible for the design and the implementation of the framework. They have to be able to create an abstract skeleton of an object-oriented system that is highly customisable. They define the hot spots in which the applications engineers can hook new applications. The application engineers concentrate on the customisation of the framework. They have to understand the framework well, so that they know what hot spots to use when a new application must be mapped to the framework.

The traditional Waterfall process model and its enhanced derivative, the Spiral process model, are not appropriate to support framework-based development. After all, the Waterfall model is based on stable, correct requirements, it does not promote software reuse, it does not promote prototyping, and it takes too long to see results.

As pointed out by Goldberg and Rubin [GR95], there is not one process model for the development of object-oriented programs. Organisations choose a process model that best fits the project at hand. Goldberg and Rubin have identified five types of projects: *First-of-Its-Kind* (building the initial version of a system), *Variation-on-a-Theme* (building a derivative system by refining an existing system), *Legacy Rewrite* (rewriting a legacy system), *Creating Reusable Assets* (producing assets for reuse in later projects), and *System Enhancement or Maintenance* (modifying core utilities or system frameworks).

According to Goldberg and Rubin, it is not likely to find a generic process model in the literature that states how a *Variation-on-a-Theme* project should be carried out, as such a project is driven by the framework and its rules for modification. The structure of the framework dictates the process model. Framework-based development is related to the *Variation-on-a-Theme* type project in that respect.

3.1.4 Framework Engineering

A framework is often defined as a skeleton program defining a reusable software architecture in terms of collaboration contracts between (abstract) classes and a set of variation points, or hot spots [Sch97]. The hot spots define where the framework can be customised. Collaboration contracts define the rules the customisation must obey. This definition is the basis for a model for framework engineering whereby an extensive domain analysis precedes the framework design. In that model, the ultimate goal is to build, through a small number of iterations, a software architecture that can be turned into a custom application by simply filling in the hot spots.

Fundamental to the framework-based development approach is the acknowledgement that for real-world applications only a limited number of frameworks can be customised by just filling in the hot spots. In general, the customisation process is much more complex, sometimes even violating part of the framework architecture. Moreover, the idea of constructing an immutable framework after a limited number of iterations is not realistic. On the one hand, the large up-front investment in the domain analysis and building prototype applications for establishing the framework architecture is in most cases not financially justifiable. This analysis is cost effective only for frameworks that have a relatively well known and stable problem domain, such as generic application frameworks (MacApp, AWT, MFC, etc). On the other hand, framework engineers are confronted with the constant changes in the specifications. As the problem domain evolves, so must the framework. It is simply not possible to conceive a framework that anticipates all future evolutions. Therefore a framework is never finished.

3.1.5 Application Engineering

While the traditional Waterfall process model lays down that the analysis of the system must be performed prior to the design of the system, in framework-based development this is no longer viable. Instead of performing an analysis from scratch, the results of earlier analyses can be reused for subsequent customisations. The software engineers perform a *delta analysis* based on the current state of the framework. In absence of a better (more formal) way to perform delta analysis, application engineers browse through a prototype with the customer and carefully write down where existing functionality covers the customer's needs, where adaptations of existing functionality are necessary, and what functionality must be added. The result is a document that can be used by the framework engineers to adapt the framework towards insights gained through the analysis. New insights typically concern variations that cannot be covered by hot spots in the framework, and commonalities that were not apparent earlier.

3.1.6 Iterative and Incremental Software Development

Essential to the framework-based development approach is the *iterative development* style, which is adopted by many practitioners of object-oriented technology in favour of the traditional process models. Iterative development is based on the belief that the development life cycle is a learning experience in which developers get things wrong before they get them right and they make them bad before they make them well [GR95]. Iteration is necessary to review earlier versions of the software and to correct mistakes based on experience gained since the development of earlier versions, or based on user feedback.

Iterative development usually goes hand in hand with *incremental development*, which allows for software development in a step-by-step manner. Each small step produces tangible results that can be shown to the customer. Together with *prototyping*, incremental and iterative development can be used to test out ideas, to build software incrementally, and to enhance the software based on early customer feedback, so that the risk of making mistakes in requirements and design is minimised.

3.1.7 Evolutionary Software Development

The subject software evolves continuously. Evolution is required when customers submit new specifications, changed specifications, or when they submit bug reports. Evolution is also necessary when the software provider has new insights in the domain or in the design, and when the complexity of the classes is too great.

Software evolution is thus not considered as an occasional activity that may be required in the course of the software's lifetime. It is considered to happen daily, and it must be reckoned with at all stages of the software development lifecycle. This style of software development is referred to as *evolutionary software development.* The amount of changes that is typical for evolutionary software development makes the management of changes an important and a necessary part of the software development process. Change management, estimating the impact of changes, and architectural drift are problems that must be addressed to keep the software system from degrading into total chaos.

3.1.8 Management of Changes

Changes are managed by employing a request/defect system to receive and process requests and defects, and to log the changes to the software system during processing (implementation).

Figure 3.2 shows the request/defect system and its use in the development process. The request/defect database is a repository that holds specifications and bug reports. Specifications come in two forms. New specifications describe desired new functionality. Change specifications describe desired changes of existing functionality. Specifications and bug reports are received from the customers and from the software engineers.

When they are received, they are given a unique identification number. When a development task is assigned to a software engineer, a specification or a bug report is drawn from the request/defect database. The software engineer designs and implements the specification or fixes the reported bug. By doing so, the software engineer makes changes to the software system. When the changes are finished, the software system will include new versions of the classes that have been changed. After finishing the changes, the software engineer manually logs the changes in the request/defect database, so that a processed specification or bug report includes information on the (versions of the) classes that have been changed to implement the specification or to fix the bug. After logging the changes in the request/defect database, the software engineer notifies the code reviewer of the development task that has been performed by supplying the identification number of the specification or the bug report.

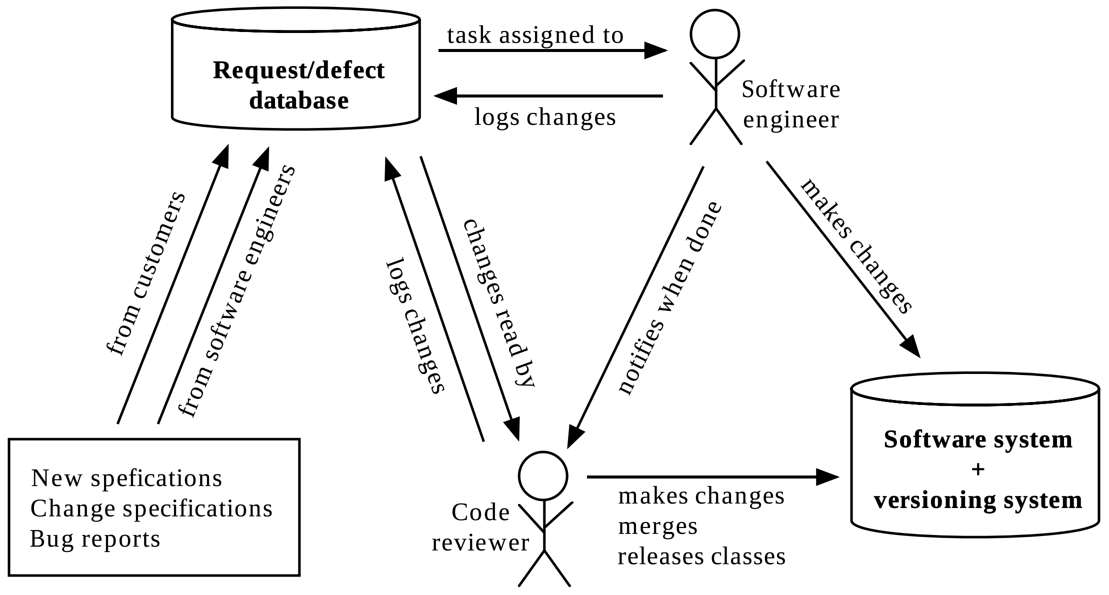


Figure 3.2: Making, logging and tracking changes in a request/defect system

Based on the supplied identification number, the code reviewer retrieves the specification or bug report from the request/defect database. He reviews the changes made by the software engineer. Three cases can be distinguished. One, the changes are not accepted. In that case, the code reviewer asks the software engineer to redo the work by assigning a new development task. Two, the changes are accepted, perhaps with small corrections by the code reviewer. Three, the changes are accepted, but they have to be merged with changes made to the same classes but for another specification or bug fix. In the first two cases, the ultimate versions of the changed classes are released in the versioning system, so that they can be distributed among all members of the development team.

The use of a request/defect system allows the software engineers and the project managers to analyse what changes were made to the software system and to look up why the changes were made. The actual changes are contained in the versioning system, while the mapping between changes and specifications and bug reports is maintained by the request/defect database. Together, the versioning system and the request/defect database log the history of the software system.

3.1.9 Software Development Environment

The software development environment used to produce the broadcast management software is Envy/Developer [OTI95] for VisualWorks\Smalltalk [PD95].
Instead of the classic categories, Envy classifies classes in *applications*, sort of hierarchical categories between which prerequisite and part-of relations can be defined. Nested applications are called *subapplications*. Applications are loaded into the Smalltalk image from a central repository (the *library*). An application can only be loaded if its prerequisite applications are already loaded.
Envy enhances VisualWorks with a versioning system. A versioning system is indispensable to support the evolutionary development style employed to build the subject software. The versioning system logs every change to the software. Methods, classes and applications can be *versioned*. Methods are versioned automatically after each accepted change. Classes and applications are versioned only on demand by the developer. Classes have an *owner* and applications have a *manager*. The owner and the manager are developers. They decide when their versioned classes and applications are *released*, i.e. made accessible to other members of the development team.
Configuration maps define configurations of applications. Simple use of configuration maps includes grouping of applications that should be loaded together. Advanced use includes the configuration of all applications that together form the delivery of a computer application for a particular computer platform.
All changes are recorded in the library, so that software developers have access to all software changes ever made (an important requirement for experimentation with evolution in this work). Envy supports browsing of changes made to methods, classes and applications. Although Envy/Developer provides the building blocks (classes) to build frameworks, it does not actually support the creation, the customisation and the evolution of frameworks. To the best of our knowledge, no commercial software development environment supports

framework development. So the software provider has to make shift with what he has got, and must set up a process model in which the software development environment is exploited to its full potential and in which insufficiently supported development tasks are handled separately in a disciplined way.

3.2 Software Evolution Problems Revisited

This section explains how the evolution problems discussed in Chapter 2 show up in the broadcast management software and the employed development style.

3.2.1 Version Proliferation

Version proliferation is an inherent problem in framework-based development, because customisation of the software is done in projects carried out on-site at the television and radio stations.

Customisation proceeds after installation of the relevant software at the station. The installed software is part of the software baseline. The software baseline is the set of classes that make up the software product as installed at the company site. It contains the framework and all its customisations. The relevant parts for a station are the framework and any customisations that apply to the station's business needs. Version proliferation happens as soon as the software is installed, since the installed version is a copy of the software maintained at the software provider's site.

When the development team in charge of the customisation decides to alter the framework, this version of the framework is no longer synchronised with the original framework at the software provider's site. Reasons for changes to the framework range from long-term vision reasons, such as new insights in the domain, to short-term vision reasons, such as making the customisation task easier. In any case, classes are moved from the framework to the customisation or the other way around, and classes in the framework are changed as necessary.

The scenario above results in two versions of the framework that must be merged in order to obtain an up-to-date version of the framework in the software baseline. The framework in the baseline may have been subject to changes as well, making the integration of the changes made during customisation into the software baseline extra difficult. To make matters even worse, the framework may have been changed in parallel at more than one station. Implementations of the different changes may be in conflict, so that the merging process is far more complicated than merely putting things together.
Figure 3.3 shows a situation in which the framework has been customised at two stations and at the company site as well. The initial version of the framework is V1. Starting from this baseline version, changes are made at two stations, resulting in versions V1' and V1". While customisation for the two stations is going on, the framework in the baseline is also subject to change, typically in response to bug reports and requests for small additions of

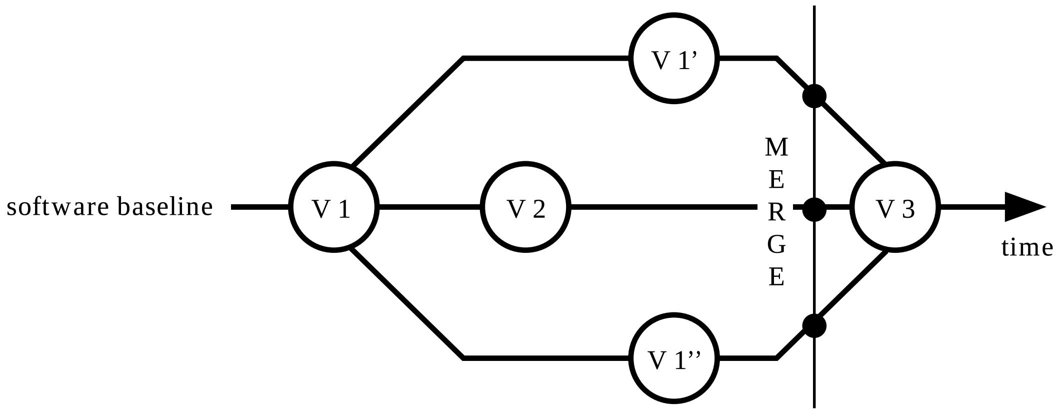


Figure 3.3: Merging software changes into the software baseline

functionality, resulting in version V2. When customisation for the two stations has been completed, the changes made during customisation are merged into the software baseline, so that version V3 of the baseline is compiled from V2, V1' and V1".

3.2.2 Poor Effort Estimation and Poor Insight into the Effect of Changes

Effort estimation is required on both levels of framework-based development. An application engineer likes to assess how much work a customisation will take. Due to the lack of good reuse documentation, assessing the amount of work is difficult, because one cannot determine what can be reused, what can be extended and what must be developed from scratch.

A framework engineer struggles with similar problems. When a proposed change to the framework is to be made, it is hard to assess how the change will affect the rest of the framework and, maybe more important, existing customisations of the framework. Care must be taken that by changing the framework no anomalies are introduced in the customisations, so that the customised applications keep on running.

Software documentation targeted to customisation would be much help for an application engineer to do his job more efficiently than is the case now. Framework engineers need software documentation that describes the dependencies between classes within the framework, as well as between classes from the framework and the classes from the customisations.

3.2.3 Architectural Drift

Although architectural drift, as argued in Section 2.6.4, typically happens because the software engineer is ignorant of the software architecture, it can also be introduced on purpose in framework-based development.

Intended architectural drift may occur during framework customisation. Since for a real-world framework customisation is far more complex than filling in the hot spots of the framework [CHSV97], customisation may require the application engineers to violate the framework architecture when the framework does not support the required customisation. When framework engineers are not available to make the framework more amenable to the desired customisation, the application engineers are compelled to violate the architecture in order to get their work done.

In framework-based development, prevention of architectural drift seems to be served with a good co-operation between framework engineers and application engineers [CHSV97]. Framework engineers have the important advisory role of giving application engineers techniques for increasing the generic aspects of their designs and code, and advice on how to reuse the framework, so that architectural drift during the customisation process is avoided. Application engineers report on their customisation experience to the framework engineers, in particular on the encountered limitations of the framework, so that the framework engineers can use this feedback to improve the framework.

3.2.4 Overfeaturing

Since framework-based development is partly product (or off-the-shelf) development, the tendency towards overfeaturing, as explained in Section 2.6.5, also exists in framework-based development.

The tendency towards overfeaturing is not only driven by customer demand, but also by the software engineers themselves. Application engineers tend to migrate features from the customisations to the framework in order to reduce future customisation efforts. The result is that applications containing features not relevant for a particular customer are still part of the standard package. The decision to include features in the framework should be taken with great care, because overfeaturing makes the framework more expensive, more complex, and less reusable for future customers. A User Advisory Board may serve as a forum for strategic customers to discuss and decide on proposed features.

Evolution of the software may give rise to unintended overfeaturing, as was shown during a case study. The broadcast-planning framework includes a manuscript module. When a television station showed interest in the broadcast planning software, it stated that it did not require the manuscript module. At that point, it became apparent that the ability to handle manuscripts was a feature that should not reside in the framework, but should be part of the customisation. Taking out the manuscript module appeared to be a non-trivial task, because it was not clear where the manuscript module interacted with the rest of the framework. In general, a feature of a product may be tightly integrated with other features, so that isolating its implementation in order to separate it from the framework is very difficult.

3.3 Role of the Case Study in this Work

The case study is an industrial case study that has been a laboratory for investigating the production of continuously evolving software, for conceiving and evaluating ideas, for developing models and methods, for prototyping tools, for experimenting with prototypes and for fine-tuning proposed solutions. Three important roles of the case study can be distinguished.

3.3.1 Industry as Laboratory

The research of this work has been conducted according to the *industry-as-laboratory* approach. The industry-as-laboratory approach contrasts the *research-then-transfer* methodology [Pot93], which can be summarised as "conceive an idea, analyze the idea, and advocate the idea" [Gla94]. Researchers using the research-then-transfer methodology expect their ideas to find their way to practice as soon as possible, but the research method is characterised by the absence of any evaluation or validation. It is therefore no surprise that practitioners are not inclined to adopt ideas conceived in such a way.
This is confirmed by Parnas, who states that in the field of software engineering none of the most influential papers presented at software engineering conferences during the last decade, nor any other papers for that matter, have had an influence on practitioners. "Engineers use methods if they work", Parnas says [Par95], but the sad conclusion is that managers are seldom able to assess the benefits of the application of a method. This is acknowledged by Fenton et al [FPG94]: "Much of what we believe about which approaches are best is based on anecdotes, gut feelings, expert opinions, and flawed research, not on careful, rigorous software engineering experimentation."

So what Glass, Fenton et al, Potts, and Parnas are saying is that because the laboratory research often fails to address significant real world problems, researchers better adopt an industry-as-laboratory approach [Pot93], in which research is done in close co-operation with industry. The foundation of the approach is that software engineering research and its application in industry are not separate, sequential activities, but complementary activities that reinforce each other and together produce better and practicable results.

The case study plays the role of the industrial laboratory in this research. The industry-as-laboratory approach is adopted to develop ideas in close co-operation with industry in order to get immediate feedback so that bad ideas are dismissed early, and to test methods and tools on real-world software instead of academic or toy applications.

3.3.2 Requirements Provider

Probably the most important role played by the case study is the role of requirements provider. Thorough investigation of the software and the employed development process, and many discussions with the software engineers and the project managers have produced a list of requirements for the recovery process, the model underlying the recovery process and the tools supporting the recovery process. Although the list of requirements is the result of investigating the case study, the requirements are not case study specific. They are concerned with considerations that go beyond the case study. They are mainly driven by concerns about the practicality of the solutions, in particular with respect to the software development tools and the software development process. The requirements are discussed in Chapter 5.

3.3.3 Platform for Experiments, Evaluation and Feedback

The broadcast management system serves as an excellent case study for this work. It is large (more than 2000 classes), it is complex (not one software engineer has a complete picture of the whole system), and it is subject to constant evolution (more than 30 new specifications and bug reports per month), so that it is representative of the large and complex evolving software systems in use today. It is representative of many software systems in another respect as well: there is not much software documentation.
The original software engineers are still around, so that the results of validation experiments can be compared against their knowledge of the software system. All source code since the software's conception is available in the repository of a versioning system, so that the evolution of the software can be traced.

Discussions with the software engineers and the project leaders have been invaluable with respect to the evaluation of ideas. Many ideas that seemed to be promising from a research point of view were rejected by the software engineers as unpractical and too hard to integrate in the software development process. This early feedback has been very helpful in directing the research. The industry-as-laboratory approach resulted in prototypes of which it was known in advance that they would be acceptable in practice. Elaboration of the prototypes has been under constant evaluation by the software engineers as well as by the project managers.

3.4 Summary

This chapter has given the context in which a large part of this research was carried out and where the methods and tools presented in this dissertation were validated. The subject software, the employed development style to produce the software, and the process model were presented. Furthermore, the software evolution problems were reconsidered in this context. The discussion on the software evolution problems confirms that the absence of software documentation is a major problem in software evolution. This chapter concluded with a discussion on the role of the case study in this work. The industry-as-laboratory approach has proven to be a very good approach to produce practical solutions. One of the case study's major roles is the role of requirements provider. The next two chapters discuss the list of requirements for the recovery process that is the objective of this work.

Chapter 4

Multiple Views on Software

The study of the subject software has spawned insight into how software developers look at the software they are building. Apparently, software engineers look at software in many ways. Software engineers take a view on the software that best matches the development task at hand. This chapter discusses two kinds of views on software: architectural views and other views that have no connection with the architecture but are still useful in software development. Each view on software is accompanied by an example taken from the broadcast management software. Chapter 10 discusses how multiple views on software can be recovered.

4.1 Architectural Views

Remember the definition of software architecture given in Section 1.2. One aspect described by the software architecture is of particular concern here: "the software architecture describes the organisation of the software as a composition of components". Architectural views render the structure of a software system in terms of architectural components. They show how architectural components, usually large sets of classes, are composed to form a software system. Four architectural views are discussed here: the software layers view, the modules view, the (framework) customisations view, and the features view.

4.1.1 Software Layers View

Many software systems are conceived as layered architectures [SG96]. A well-known example is a layered architecture with three software layers (see Figure 4.1). The domain model layer consists of all so-called *domain classes* that represent domain concepts. The user interface layer consists of the classes that are needed to display and manipulate domain model layer objects through a (graphical) user interface. The persistency layer consists of the so-called *storage classes* that represent domain model layer objects when they are stored in a database.

The purpose of this layered architecture is a clear separation of concerns [HL95]. Each layer addresses a particular concern: user interface, domain logic and persistency. The boundaries between the layers define a protocol between the layers that may be not be breached. Breaching the protocol results in architectural drift, as explained in Section 2.6.4.

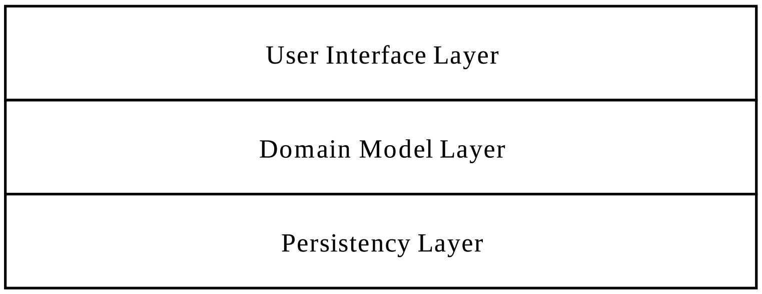


Figure 4.1: Software layers

In the broadcast management software, the domain layer includes domain classes for broadcast management: programmes (news, series, sports event, ...), time frames (season, week, day, ...), schedules, television stations, etc. The user interface layer contains many special-purpose widget classes to display and manipulate planning objects: a season planner widget, a week planner widget and a day planner widget. The broadcast planning applications, also part of the user interface layer, are compositions of these special-purpose widgets.

4.1.2 Modules View

Software can also be looked on as the combination of several functional requirements. A module defines which classes implement some functional requirement. The modules view describes how the software is divided into modules.

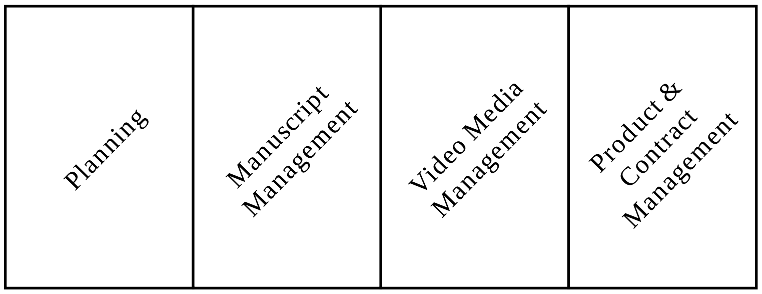


Figure 4.2: Software modules

Figure 4.2 shows some of the modules in the broadcast management software. The Planning module holds all classes related to planning. The Manuscript Management module is concerned with scripts for home-made programmes. The Video Media Management module is responsible for the management of the video media library holding tapes, disks, etc. The Product & Contract module handles the purchase of programmes and the contracts that come with them.

Software modules are not used in isolation. In fact, a close co-operation of the modules is essential for the broadcast management software to support the complex work processes of television and radio stations. For example, the Planning module and the Product & Contract Management module have to work together so as to ensure that a programme (product) is planned during the period stipulated in the contract that comes with it. In the same vein, the Planning module and the Video Media Management module work together to ensure that a planned programme can only be broadcasted if the tapes are actually present in the video media library.

Each module has three layers as shown in the software layers view, so when the modules view and the software layers view are combined, the combined view on the broadcast management software looks as depicted in Figure 4.3.

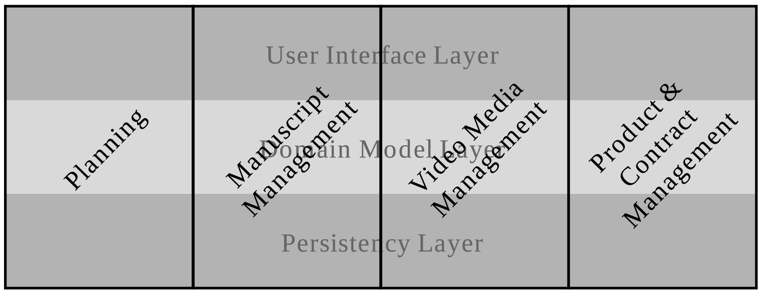


Figure 4.3: How modules relate to layers

The three layers in one module interact with each other to achieve module-related behaviour and the different modules work together to provide an integrated broadcast management solution. The same layers of different modules interact to achieve that integration. For instance, the domain layers interact to validate programme planning as already mentioned above, and the user interface layers interact to open applications of other modules.

4.1.3 Customisations View

In framework development, the framework is the core of the software. It is customised for different products or for different customers. This means that the framework is instantiated and that the variations for a product or a customer are defined.

The customisation view clearly marks the boundaries between the classes belonging to the framework and those belonging to the different customisations. This view accords with the framework development style, in which framework engineers are responsible for the framework and the application engineers focus on the framework customisations. Application engineers can employ this view to get an overview of the classes that constitute a customisation and framework engineers use it to get insight into how the framework classes have been reused.

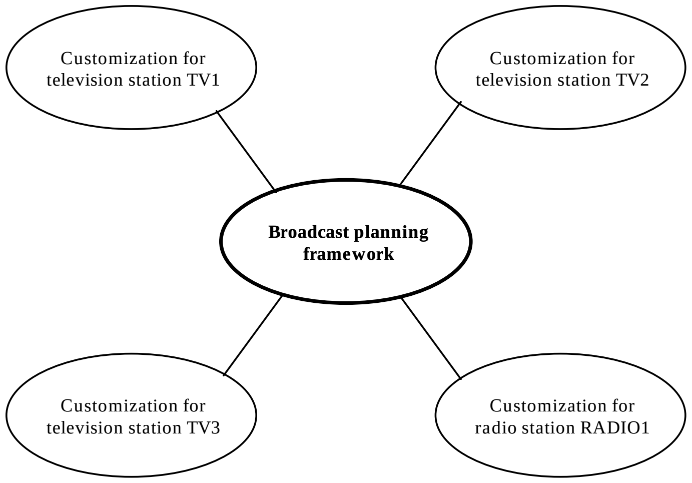

Customization for television station TV1
Customization for television station TV2
Broadcast planning framework
Customization for television station TV3
Customization for radio station RADIO1

Figure 4.4: The broadcast planning framework and its customisations

Figure 4.4 shows the customisation view on the broadcast planning software. Three television stations and one radio station reuse the framework. For each station, a customisation of the framework has been made.

The customisations view can be combined with the combined software layers and modules view to give insight into how customisations relate to modules and software layers. For example, Figure 4.5 shows the combined view for two customisations. The black customisation customises the framework in all four modules, but only in two software layers. The persistency layer is not customised. The white customisation changes the user interface layer and the domain layer of the two modules on the right, and it customises the persistency layer of the rightmost module.

4.1.4 Features View

A feature [JGJ97] is an aspect of the software that extends across a large number of classes. Examples in the user interface layer are on-line help, the ability to undo commands and translation of text displayed in the user interface. A permission system, granting or refusing access to some parts of the software, is an example of a feature in the domain layer that affects the user interface layer. When a user does not have reading access to some information, that information is not displayed in the user interface. When the user has reading, but not writing access, the user interface displays the information in read-only views.

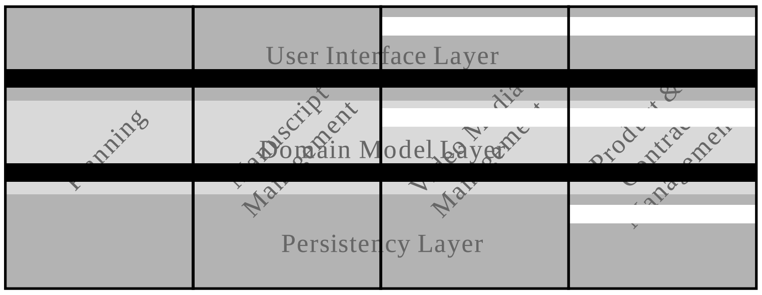


Figure 4.5: Combined customisations/layers/modules view (for two customisations)

The features view stands out against the other views, because the view does not include entire classes, but instead includes only methods in the classes that address a feature. Since methods addressing a feature are scattered over many classes, and because current development tools are class-centred, a features view is hard to establish. Consequently, feature views usually only exist in the heads of the software developers.

Although a features view is not supported by the object-oriented paradigm, nor by the development environments, it is an indispensable view for software developers. Because features extend across a large number of classes, changes to a feature have an immense impact on the software. In the current state-of-the-art of object-oriented programming, a change to a feature implies a large amount of work to find and change the methods that address the feature. Having a features view considerably reduces the time to search for the methods that address a feature.

4.2 Other Views

In software development, views other than architectural views may be helpful to reason about the software. Views that provide insight into the structure of the software with respect to the development process are particularly interesting. Two are given here: the ownership view and the traceability view.

4.2.1 Ownership View

Software developers often have to carry out activities that are only related to the classes under their responsibility. For example, when a class is adapted by another developer, the owner of this class has to check whether the adaptation is consistent with the change requirements, whether it does not clash with other changes to the class (adaptations by yet other developers), and whether the class still defines a behaviour that is expected by its client classes. If no problems are found, or when they are solved, the owner of the class can release the class to the other members of the development team.

The ownership view helps to keep an overview of how class ownership and responsibility is distributed among the members of the development team. It shows which classes belong to a given software developer (see Figure 4.6).

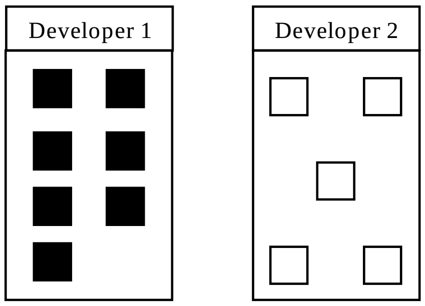


Figure 4.6: Ownership view for two developers

The set of owned classes does not necessarily correspond to customisations, modules or software layers. How and when class ownership is granted depends on the organisation of the development team and the development style. Ownership may be granted based on expertise in the domain, based on technical expertise required to implement the classes, or based on the fact that the developer did the initial implementation. The combination of the ownership view, the software layers view, and the modules view may very well look like a patchwork, as shown in Figure 4.7. Black squares are classes that belong to one developer; the white ones belong to another developer.

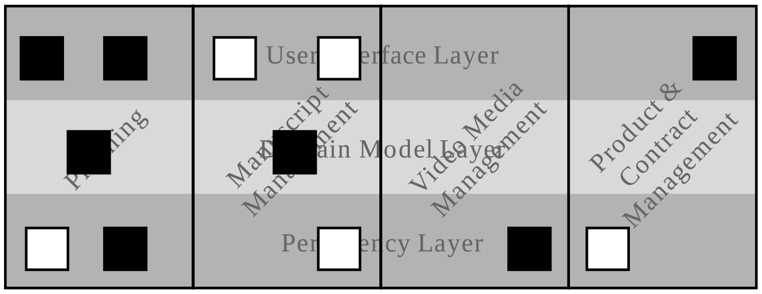


Figure 4.7: Combined ownership/layer/module view for two developers

4.2.2 Traceability View

Classes and methods are created and changed according to new specifications, changed specifications, or bug reports. In absence of a request/defect system, there does not exist a record of how changes made in the implementation correspond to a specification or a bug fix. Such a trace would be very helpful, however, because it provides insight into the dependencies between specifications at implementation level. When methods are changed

in response to different change specifications or bug fixes, this indicates that those methods are used in several contexts and changing them for one context may invalidate them for another context.
For example, in a development environment where several developers may change the same classes, development may get into the following situation. A change made by one developer in response to one bug fix may introduce an unnoticed bug somewhere else. The latter bug, which shows up some time later, may be fixed by another developer by undoing the changes made to fix the former bug, which results in a circular and endless bug fixing scheme.

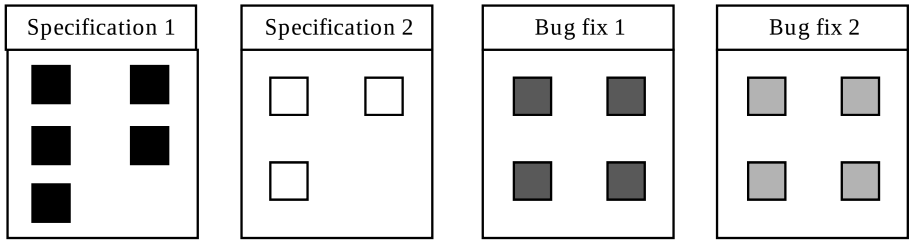


Figure 4.8: Traceability view

A traceability view shows how classes and methods relate to specifications and bug fixes (see Figure 4.8). It gives insight into how specifications are implemented and it helps to assess what parts of the software may be affected when a change is made. If the changed methods are part of other specifications, these specifications should be examined for correctness. The traceability view is thus helpful for software change impact analysis.

4.3 Conclusion

This chapter challenges the prevalent opinion that software systems have one predominant software architecture. The study of the subject software clearly demonstrates that software systems may have multiple architectures. The different views on software architecture discussed here show that a software architecture can be decomposed along several equally important lines. The customisations view, the software layers view, the modules view, and the features view are different views on the architecture.

The discussion of the different views also indicates that software engineers would benefit from having these views. Each view gives insight into a different aspect of the software, which is important when a piece of the software needs to be changed.

The list of interesting views is not restricted to the ones given in this chapter. Some development tasks may require other views, more targeted at the development task at hand. Software engineers may even have their personal views that help them in their daily development tasks.

Chapter 5

Requirements for the Recovery Process

The case study presented in Chapter 3 has played the important role of requirements provider. The subject software has been studied carefully, the development process has been investigated, and the software engineers have been interviewed. These activities have spawned a list of requirements for the recovery process, for the model used to represent recovered software entities, and for the tools that support the recovery process. This chapter discusses these requirements.

5.1 Take Multiple Views on the Software into Account

The previous chapter has shown that multiple views on software are desirable and useful. Different views give insight into the different aspects of the software, which is important when a piece of software needs to be changed. For example, a framework engineer requires another view on the software than an application engineer. The problem is, however, that usually software development environments do not provide a means to establish multiple views on the software. Thus, a model for architectural recovery cannot rely on existing means to express multiple views. It must include the necessary concepts so that it can be used to express multiple views on software, in particular multiple views on the software architecture.

5.2 The Recovery Process Should be Incremental

Software engineers often do not reverse engineer the source code completely. They reverse engineer up to a point at which they know enough to evolve the software. Reverse engineering a very large software system completely would be unfeasible anyway. It would take too much time and too much manpower to reverse engineer collaboration contracts or architectural components in a software system with a few thousand classes. Therefore, a recovery process should be incremental.

An incremental approach to recovery also implies that the tools for recovery do not introduce modes in the development environment. The recovery tools should not block other development activities. The software engineer should be able to switch between programming and recovery at all times.

5.3 Motivate the Software Engineer

Software engineers show a very reluctant attitude towards changes in the development process and the software development environment they work with. The development of a fully-fledged methodology for disciplined reuse and evolution based on reuse contracts, of which this work is an important step, requires changes to the development process. These changes may involve major changes to the daily working habits of the software engineers. The problem is, however, that it is hard to convince software engineers of changes of which it is not certain what the outcome will be. Forcing software engineers to radically change their working habits may even have the undesired effect that their productivity decreases. Experience shows that in order to win over the software engineers, the changes should be introduced gradually, and the changes should spark off new benefits. Motivation is a strong force to convince people. Software engineers are motivated to change their way of working or to do (little) extra work if they know that they will get something in return that makes their work simpler or easier. A little change or a little overhead that will pay off afterwards appears to be acceptable.

5.4 Keep the Model Simple

A recovery process requires a model to represent the software entities that are to be recovered. In this work, this means that the model should include collaboration contracts, reuse contracts and large architectural building blocks, but also the smaller software entities they are composed of, such as classes and collaboration contract participants.

One approach could be to provide a different model for each software entity that is considered for recovery. Another approach is to provide one generic model that allows describing all software entities. The latter approach has the advantage that software development tools are easier to build, because the tools all use the same unified model.
Whatever approach is chosen, the model should be simple, so that it is easy to learn and easy to understand. Moreover, the model should be open, so that future extension of the model is possible.

5.5 Keep the Recovery Process Lightweight

The rationale behind a lightweight approach is that recovery should be fast and relatively easy, and without devoting too much resources. Complex and time-consuming reverse engineering activities are likely to be dismissed as unpractical and contra-productive. Related work [Mur96] has already presented successful lightweight approaches to recovery. Speed and the incremental refinement of the recovery parameters are essential.

5.6 Integrate the Model and the Recovery Process in the SDE

In absence of reverse engineering tools, software engineers reverse engineer manually. Manual recovery is generally based on browsing the source code with the development tools, and by consulting external software documentation, if available. Recovery of abstractions from the source code is based on filtering and selecting software entities that are present in the development environment, such as classes and methods.
Discussions with software engineers revealed that a recovery process and its underlying model to represent recovered software entities would not be accepted by software engineers if they were not integrated in the software development environment. The integration is essential to break down barriers for the software engineers.

5.7 Integrate Existing Software Entities in the Model

Software engineers do not work with the software entities that are the target of our recovery objectives. They work with classes and with software entities to organise classes, as provided by the employed software development environment. Often these organisational entities are essential in the development environment, as they are used as compilation units, or as units that can be loaded/unloaded. The integration of these organisational software entities in the model is thus essential. Otherwise, the software engineers would not be able to work with the model and the tools that are based on that model.
In case of the subject software, which has been developed with Envy/Developer, meeting this requirement means that classes, methods, Smalltalk categories and Envy applications must be integrated in the model and in the supporting tools.

5.8 Make the Results of Recovery Tangible in the SDE

In general, the results of recovery are design abstractions (see the definition of design recovery in Section 1.1). Collaboration contracts and reuse contracts are design abstractions too. A collaboration contract is an abstraction of an interaction structure of a set of classes. It makes an interaction structure of a set of classes explicit that is buried in the source code. A reuse contract is an abstraction of the reuse or the evolution of a collaboration contract. It makes explicit how an interaction structure of a set of classes is reused or evolved. Reuse is implicit in the source code. Evolution is implicit in the version control system (if available, otherwise evolution leaves no trace).

Recovery is done to get a mental model of the software. The mental model serves as the foundation for further analysis or for making changes. When the recovered abstractions cannot be represented in the software development environment, it is up to the software engineer to relate the recovered entities with the classes found in the development environment. Frequent switching between the recovery tools and the development environment is necessary. Needless to say that this way of working is annoying, unpractical, and error-prone.

Therefore, the results of recovery should be tangible in the software development environment. The requirement that the model and the recovery process should be integrated in the environment is a prerequisite for this requirement. When the model and the recovery process (and the tools) are integrated, the results can be made available, so that they can be used in forward engineering. The recovered abstractions can be linked with the classes in the software system, so that the software engineer is able to use the abstractions actively, instead of passively, as is the case when the results are not tangible in the development environment.

5.9 Cope with an Obscure Software System

Recovery of design abstractions is not a trivial problem. Recovery might be easier when some characteristics of the software are known in advance. For example, if design decisions and implementation strategies are known, it is easier to recover some design abstractions, because that knowledge can be hardwired into the recovery algorithm. In the same vein, if naming conventions are known, it is easier to identify the nature of variables and other named entities. If the source code is annotated with special comments, these annotations can be taken into account when abstractions are recovered.

In general however, no assumptions whatsoever can be made about the software. It is likely that design decisions have been bypassed, that implementation strategies have not been followed, that naming conventions have not been respected, and that expected source code annotations are missing. Moreover, software may not be correct, especially in Smalltalk and other non-statically typed languages: it is possible that non-existing messages are sent, that abstract classes are instantiated, etc. Since software is developed by humans, and since humans are liable to error, and often under deadline pressure, a recovery process cannot make assumptions about the (quality of the) target software.

Therefore, the recovery process should start from the idea that little or nothing is known about the subject software.

5.10 Summary

This chapter has discussed the requirements for a recovery process and for the model that is used to describe the target software entities. The list of requirements is the result of investigating the subject software and the development process used to build the software.

Many of the requirements boil down to the selection of a proper model. The model should be simple, yet powerful enough to describe existing software entities as well as the target software entities. Moreover, the model should be integrated in the software development environment, as does the recovery process, so that the software engineer does not need to leave his familiar working environment, and so that the results of recovery are tangible in the development environment.
A recovery process based on such a model should be incremental, so that no modes between reverse and forward engineering inhibit flexible software development. It should assume nothing about the subject software, so that it can be used without prior knowledge of the (quality of the) software. Moreover, the recovery process should be lightweight, so that no complex and time-consuming operations are required. All these requirements are bound to the requirement that the recovery process should be motivating. If the software engineer is not motivated to use the recovery process, the recovery process has no practical value and consequently it is useless.

The next chapter proposes a model that satisfies the requirements given here. Chapter 7 proposes a recovery process for collaboration contracts based on that model and Chapter 8 discusses the recovery of reuse contracts. The recovery of architectural components, and the integration of the model in the tools and in the development environment is discussed in Chapter 9 and Chapter 10.

Chapter 6

Software Classification

This chapter presents software classification. Software classification is a model, as well as a technique.

The *software classification model* is a metamodel that describes how recovered software entities should be modelled to make them tangible in the software development environment. The model will be used as the foundation of architectural recovery activities presented in Chapter 7, Chapter 9, and Chapter 10.

The metamodel is divided into two parts. The first part is the *participant model.* It describes how participants in a collaboration contract are modelled. This model accords with the theoretical model on reuse contracts. The theory does not say how collaboration contracts and reuse contracts can be organised, however. The *classification model*, the second part of the metamodel, covers those organisational aspects. It defines a flexible structure to describe and organise software entities. It will be shown how the participant model fits in the classification model, making the classification model the superstructure of the participant model. Instead of giving formal definitions, the important concepts in the metamodel are presented using UML notation [OTI97], [Lar98].

The *software classification technique* is a way to carry out classification. The technique is based on *software classification strategies.* The model thus states what the target entities of software classification are, while the strategies define how these entities can be set up. Many classification strategies are conceivable, but only four are introduced here: *manual classification*, *virtual classification*, *classification with advanced navigation tools*, and *automatic classification through method tagging.* The latter two deserve more elaboration, especially with respect to the integration in the software development environment and their applications in architectural recovery and software evolution. They are presented in detail in Chapter 9 and Chapter 10.

6.1 Participant Model

The participant model describes the metamodel concepts that correspond with classes, methods, method invocations, and acquaintance relationships. In short, it provides concepts to create abstractions of classes found in the source code.
The participant model is the basis for modelling collaboration contracts, because a set of participants comprises a collaboration contract. The participant model does not include the concept of a collaboration contract, however. The classification model takes care of that by representing a collaboration contract as a classification of participants.

The participant model accords with the theory on reuse contracts [Luc97]. The definitions referred to in the following sections can be found in Appendix A.

6.1.1 A Participant is a View on a Class

The concept of a class is interpreted in different ways, depending on the point of view. A class can be interpreted as:

The modification of its superclass. A (sub)class is often thought of as the modification of its superclass. That means that only the changes with respect to the superclass (added methods, overridden methods, added instance variables) are considered. This view on a class is strongly influenced by the mechanism object-oriented languages employ to define classes: incremental modification of a root class.

The flattened class. A class is more than the modification of its superclass. It is the result of successive incremental modifications as laid down by its superclass chain. The flattened class gives "the whole picture": all methods and instance variables defined by the class and inherited from the superclasses.

A partially flattened class. In some situations, software developers are only interested in a partially flattened class chain. For example, in Smalltalk many developers are not interested in the interface of the root class Object, because it contains a lot of methods that are required to let every object behave correctly in the Smalltalk environment. Many methods of Object are metalevel methods that have little bearing on the correct behaviour of the instances of the subclasses of Object.

A role. A class usually plays several roles, depending on the context in which the class is used. The roles are reflected in the collaboration contracts in which the class participates. Typically for each role a different part of the class' interface is used. A role is thus a class restricted to a subset of its methods, i.e. a class with partial interface.

A partially flattened class with partial interface. This view on a class is the combination of the previous two views.

The last and most general view on a class is adopted by the participant model. A participant represents a role played by a partially flattened class in a collaboration contract.

The last view on a class can be used as a representation for the first four views. It can be used to represent a modification X of a superclass, through inclusion of all methods in a class chain X..X. It can be used to represent a flattened version of class Y, through inclusion of all methods in Y's superclass chain Y..Object (assuming Object is the root class). In the same vein, it can represent a partial flattened class Y..X, where X is an arbitrary superclass of Y. Finally, it can be used to represent a role when only part of the class' interface is included.

6.1.2 Participant

According to the theory (Definition 9), a participant in a collaboration contract has a name, an acquaintance clause, and an interface. The acquaintance clause is a set of acquaintance relationships (Definition 10) and the interface is a set of methods (Definition 11).

The participant model accords with the theory. Figure 6.1 depicts how a participant is modelled.

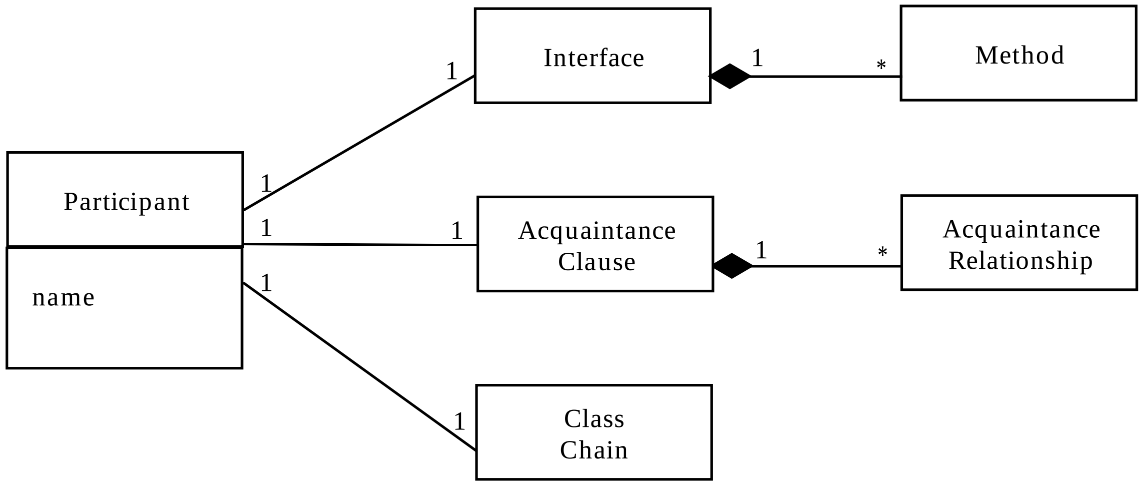


Figure 6.1: Participant

A participant is a view on a partially flattened class. This is reflected by the association between Participant and Class Chain. Class Chain represents the partially flattened class of which Participant is an abstraction. Participant is a view because Participant's Interface may hold only part of the interface of the flattened class, and its Acquaintance Clause may hold only part of the acquaintance relationships Participant is involved in.

The associated Class Chain is traceability information. It establishes the link between the model (design level) and the source code (implementation level). The traceability is new with respect to the reuse contract theory, in which traceability is not worked-out.

6.1.3 Acquaintance Relationship

Acquaintance relationships are modelled according to the theory. An acquaintance relationship is an association between an acquaintance name and a participant name (Definition 10).

Again, the model deviates from the theory with respect to traceability. An Acquaintance Relationship keeps a link to the source code through an Implementation Stereotype.

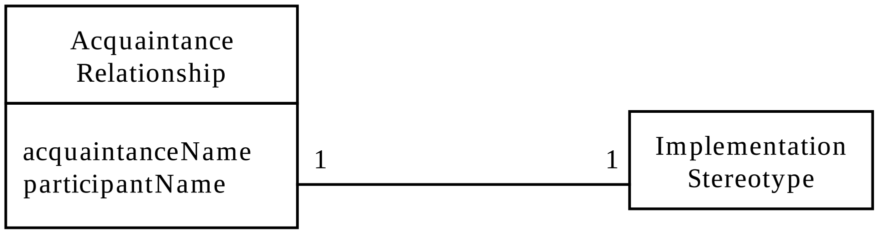


Figure 6.2: Acquaintance relationship

An implementation stereotype records how an acquaintance relationship is established in the source code. There are multiple ways to set up an acquaintance relationship in source code[1]. For each of them, there exists an implementation stereotype, as shown in Figure 6.3. The number and the nature of the stereotypes is language dependent. Note that implementation stereotypes are not merely types, or tags attached to acquaintance relationships. They record the link to the source code. For example, the Parameter Stereotype and the Temporary Stereotype keep a reference to the method in which the acquaintance relationship is established.
While the theory states that acquaintance names should be unique, in this model this condition is relaxed. The combination of an acquaintance name and a stereotype must be unique. The same acquaintance name can be used to refer to different acquaintances, as long as the corresponding implementation stereotypes are different. This is crucial for acquaintance relationships established in different methods, for instance through parameters that have the same name.

6.1.4 Method

The interface of a participant holds methods. According to the theory, a method has a method signature, an abstractness attribute (`isAbstract`), and a specialisation clause

[1]How acquaintance relationships are set up in source code will be discussed in detail in Chapter 7.

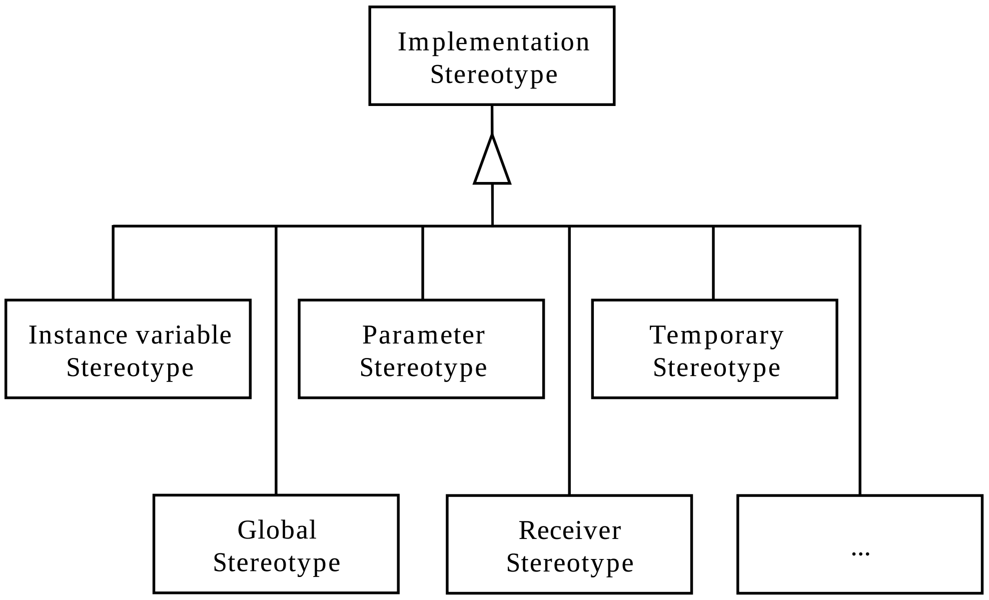


Figure 6.3: Implementation stereotypes

(Definition 11). The specialisation clause is a set of method invocations (Definition 12). As depicted in Figure 6.4, the model follows the theory.

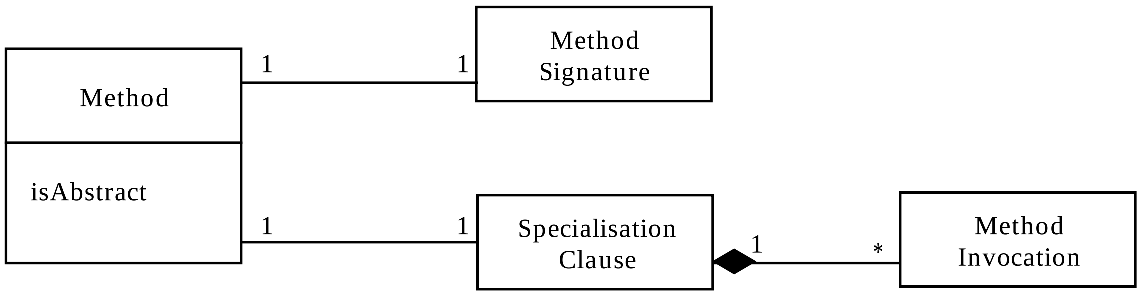


Figure 6.4: Method

The Method Signature is modelled explicitly so that a Method can be used to model methods for different languages. Method Signature can then be specialised as depicted in Figure 6.5.

Since Smalltalk has no type system, Smalltalk signatures are actually nothing more than the names of the methods (so-called selectors)[2]. Java signatures carry the method name, the formal types of the arguments, and the return type.

[2]A Smalltalk selector also includes the number of arguments, thanks to the way selectors are represented.

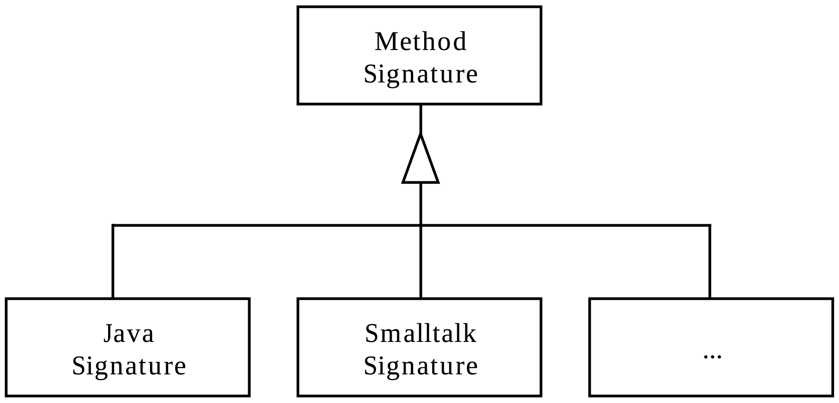


Figure 6.5: Method signature

6.1.5 Method Invocation

According to the theory, a method invocation is an association between an acquaintance name and a method signature (Definition 12). The acquaintance name denotes the receiving object, and the method signature identifies the message being sent.

The model deviates from the definition by associating an acquaintance relationship with a method signature, instead of an acquaintance name. This choice is in harmony with the fact that an acquaintance relationship cannot be uniquely identified by an acquaintance name, but by the combination of an acquaintance name and an implementation stereotype (see Section 6.1.3). Moreover, this way of modelling reflects that messages are sent across acquaintance relationships.

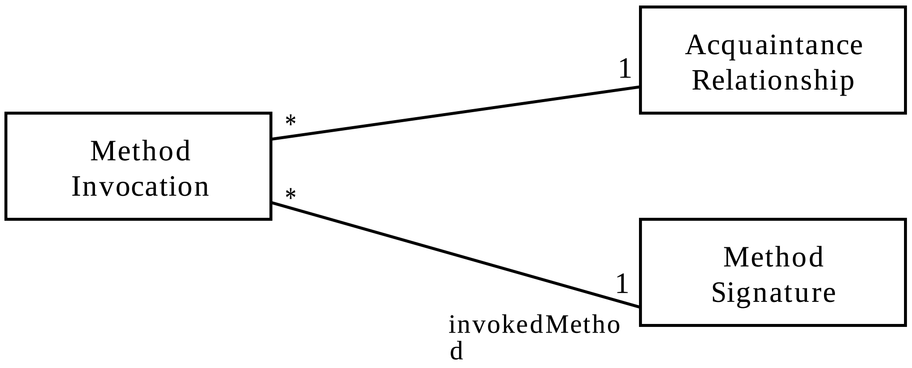


Figure 6.6: Method invocation

6.2 Software Classification Model

While the participant model describes how abstractions of classes are represented, the classification model focuses on the organisational aspects. It provides concepts to organise software entities in a flexible way.

6.2.1 Classifications and Items

A *classification* is an entity in which *items* can be classified. A classification is thus a container for items. There are several types of classifications. Some classifications are very simple. They just hold the items without putting constraints on them. A Smalltalk category is an example of a classification that holds classes without constraining them. Other classifications only hold items that have some relationship. A classification with all subclasses of a certain class is an example of a classification that expresses a relationship between its items.

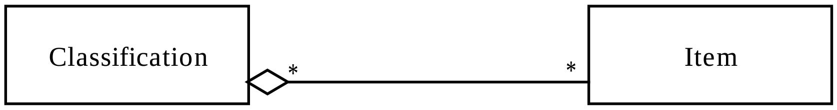


Figure 6.7: Classifications and items

At the highest abstraction level, classifications, items, and their relationship can be modelled as depicted in Figure 6.7. There are no restrictions on the number of items in a classification and an item may be classified in more than one classification[3].

6.2.2 Classifications as Items

Classifications can carry other classifications as items. This is useful to model subclassifications and other kinds of decomposition. For example, an Envy application may have subapplications, and a classification representing a module may contain classifications that represent submodules. Figure 6.8 shows how a classification is wrapped with an Itemized-Classification object to turn it into an object (item) that can be put in a classification.

6.2.3 Classes as Items

It seems obvious that classes are entities that should be classified. After all, they are the basic entities that are browsed and manipulated by the software engineer. The model includes a special item that represents a class in a classification. A Class Item holds a reference to a class in the software system. Class Item is an item that has an interface. Since other kinds of item may have an interface as well (see Section 6.2.4)[4], an intermediate concept (Item With Interface) is included in the model.

[3]This accounts for the hollow diamond in the UML diagram, which expresses a shared aggregation.

[4]Only two items with an interface are used in this document, but software modules and software layers can also be considered as items with an interface.

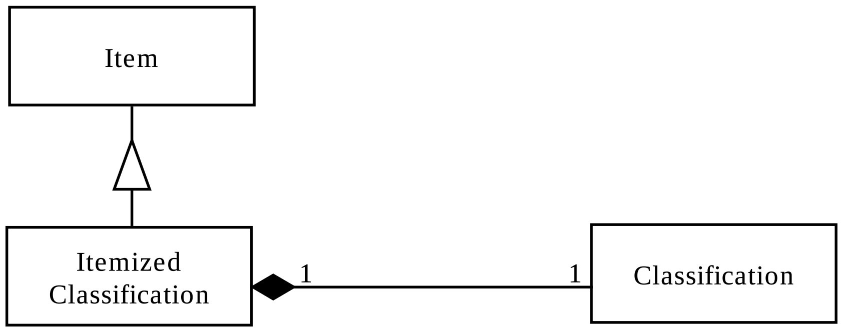


Figure 6.8: Classifications as items

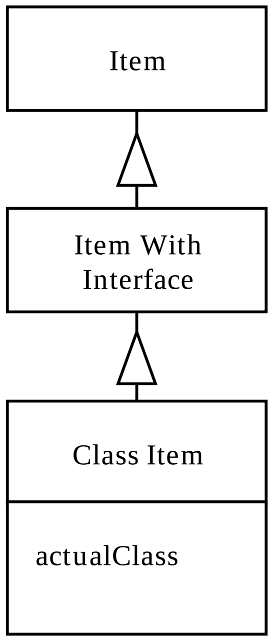


Figure 6.9: Classes as items

6.2.4 Participants as Items

The classification of participants in classifications is the link between the participant model and the classification model. Figure 6.10 shows how participants show up as items. ParticipantItem is a kind of item that represents a participant as an item in a classification.

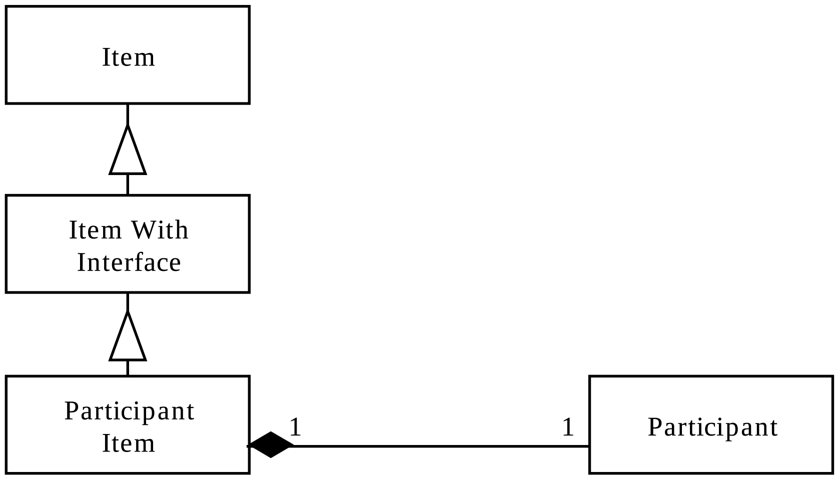


Figure 6.10: Participants as items

The use of ParticipantItem is not restricted to modelling participants in a collaboration contract. It can be used to classify parts of classes (see Section 6.1.1) in classifications. This allows the software engineer to reason about partial classes outside the context of collaboration contracts. For example, ParticipantItems can be employed to capture the methods that are changed during a bug fix. The ParticipantItem then represents the method changes made to the corresponding class.

6.2.5 Classifications Have a Structure

While some classifications do not expect the classified items to have any relationship, some classifications do. To express relationships between items, classifications have a structure that states how the items are related. For the simplest classifications, no structure is required. For others, a specific structure is necessary. For example, classifications representing Smalltalk categories have a Smalltalk Structure, expressing that categories are nothing but a grouping mechanism without semantics. Classifications holding a complete class hierarchy would have an Inheritance Structure to express that the Class Items are related by inheritance.
The structure can also be used to define how the contents of a classification should be displayed. The items in a classification with a Smalltalk Structure would be displayed in an alphabetic list; the items in a classification with Inheritance Structure would be displayed in a hierarchical manner.

Figure 6.11 shows that each classification has a structure, possibly an explicit NoStructure.

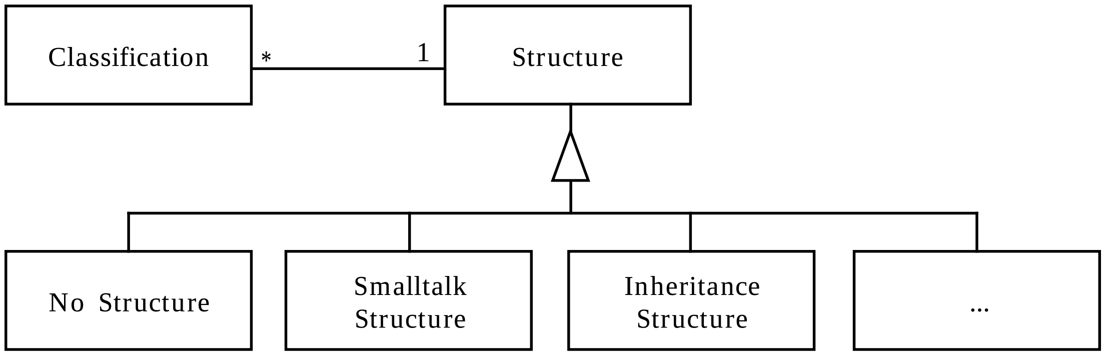


Figure 6.11: A classification has a structure

6.2.6 Classifications Have a Classification Policy

Until now, there were no restrictions on the items that can be put into a classification. In general, however, not all items can be put in any given classification. For example, Envy applications may contain classes as well as other (sub)applications, but they cannot hold Smalltalk categories. Smalltalk categories hold only classes[5], no other items. Moreover, the classes should have unique names.

Therefore, classifications have a classification policy (see Figure 6.12). A classification's policy determines whether a given item can be classified in the classification, and whether items in a classification can be changed (renamed for instance). It is also responsible for keeping the structure of a classification consistent.

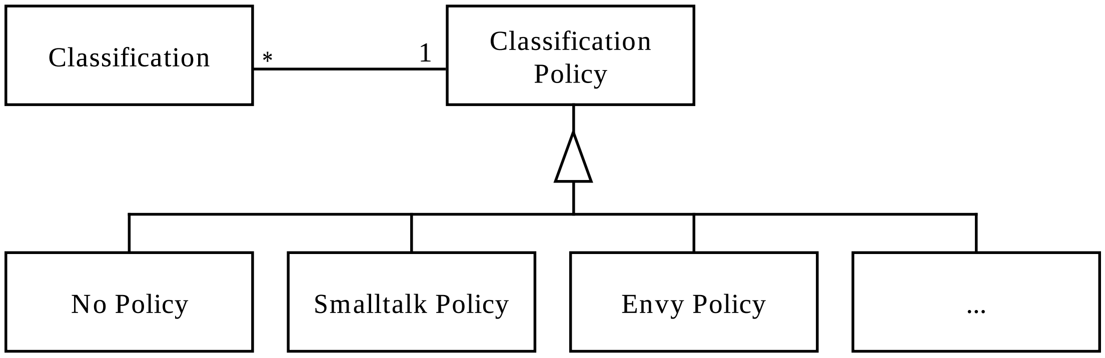


Figure 6.12: A classification has a classification policy

[5]With Smalltalk categories, *class* categories are meant, as opposed to method categories (usually called *protocols*).

Some classifications have no policy, indicating that they can classify all kinds of item. Some classifications require a special-purpose classification policy, of which some are shown in Figure 6.12. For example, Envy applications have an associated Envy Policy to ensure that only Class Items and Envy Applications are classified.

6.2.7 Collaboration Contracts as Classifications

According to the theory, a collaboration contract consists of a name and a set of participants, each with a unique name within the collaboration contract (Definition 8). Since a classification has a name and a classification is a container for items, a collaboration contract can be represented by a classification of participants with a naming restriction. A classification with Participant Items only, all having a unique name, models a collaboration contract. The naming restriction is enforced by the Collaboration Contract Policy. The fact that the participants in a collaboration contract have a relationship is expressed by the Collaboration Contract Structure. Figure 6.13 collects elements from previous diagrams to show how a collaboration contract is modelled by a classification.

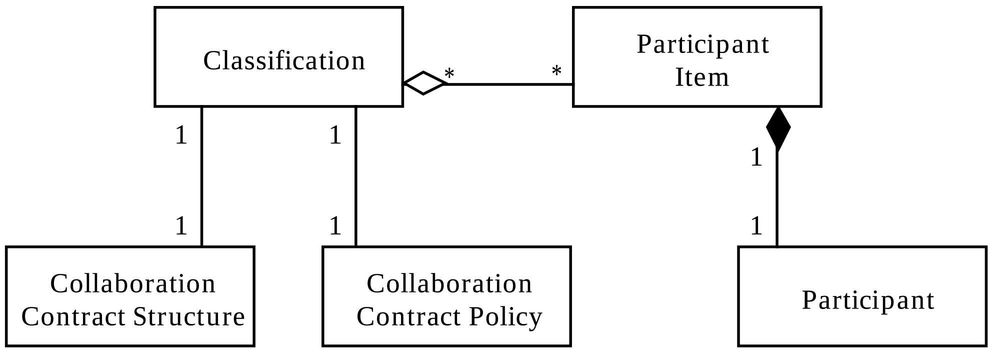

Classification
Participant Item
Collaboration Contract Structure
Collaboration Contract Policy
Participant
*
*
1
1
1
1
1
1

Figure 6.13: Collaboration contracts as classifications

6.2.8 Reuse Contracts as Classifications

Like collaboration contracts, reuse contracts can be considered as classifications as well. A reuse contract describes how a collaboration contract is reused to derive a new collaboration contract. According to the theory, a reuse contract consists of a name, a provider clause, a contract type, and a reuser clause (Definition 14). It will be shown in Section 9.4 that the contract type and the reuser clause can be computed automatically from an initial and a derived collaboration contract. In that view, a reuse contract can be seen as a classification of two collaboration contracts, of which the contract type and the reuser clause are virtual (i.e. computed) attributes. Figure 6.14 collects elements from previous diagrams to show how a reuse contract is modelled by a classification.

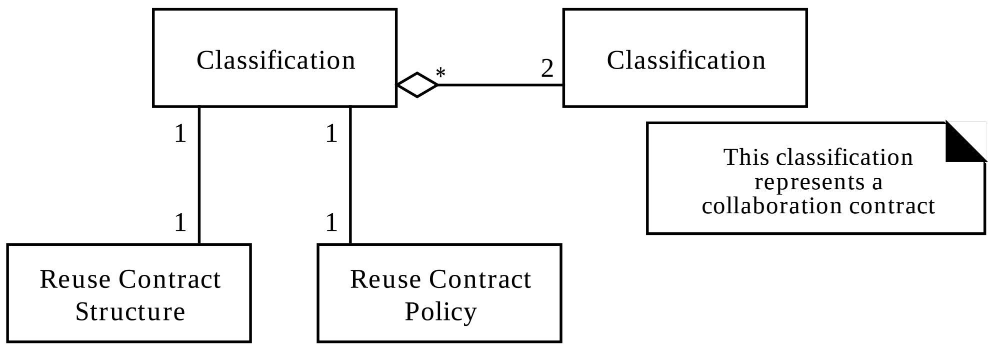


Figure 6.14: Reuse contracts as classifications

6.2.9 Virtual Classifications

Besides classifications defined by the software developer, there are classifications that are native, that is, already present in the development environment, or that can be computed from data retrieved from the environment, or from other classifications.
In order to integrate existing classifications or computed classifications, virtual classifications are introduced. A virtual classification is a front-end for a native or computed classification. Examples are:

- categories in a Smalltalk environment
- applications in an Envy environment
- a classification holding all classes in the environment
- all senders of a given message
- all implementers of a method

In an environment based on classifications, virtual classifications of classes are essential to bootstrap the use of classifications for software entities that are larger than classes. Virtual classifications can be exploited to create classifications of entities extracted from software. These entities are not necessarily classes and methods. They can also be metrics, search results, or anything else. The inclusion of user-definable virtual classifications (not shown in Figure 6.15) opens the door to easy and flexible extendibility of the model from within a development environment.

6.2.10 The Classification Repository

Classifications are stored in the classification repository. The repository is responsible for the creation, manipulation, consistency and persistency of classifications. Classification and manipulation of items is done in co-operation with classification policies. The repository can be queried for classifications and items. Since items can be classified in different

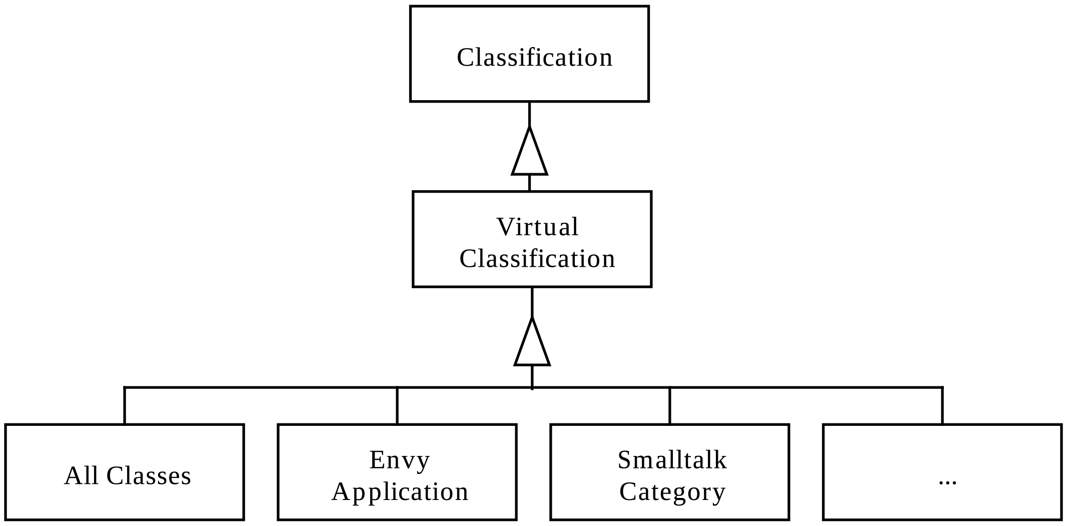


Figure 6.15: Virtual classifications

classifications, one important query involves the inverse mapping between items and classifications. The classification repository maintains inverse classifications (not shown in the figure below) that are used to process such queries rapidly.

As depicted in Figure 6.16, the classification repository has one privileged classification, called the *root classification*. It is the classification that holds all other classifications and items, directly or indirectly.

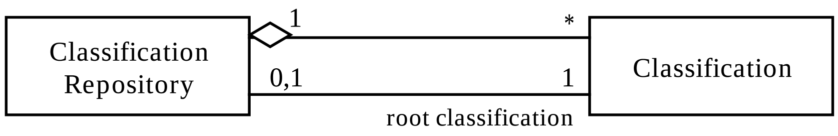


Figure 6.16: Classification repository

The classifications in the root classification are referred to as the *top-level classifications*. Many virtual classifications are top-level classifications, so that they can be found easily. Section 9.1.3 explains how top-level classifications show up in the tools that make use of classification.

6.3 Classification Strategies

A classification strategy is a method for setting up classifications. Many classification strategies can be devised, ranging from setting up classifications manually, to generating classifications automatically.

Four classification strategies are introduced here: *manual classification*, *virtual classification*, *classification with advanced navigation tools*, and *automatic classification through method tagging*. The first two strategies are straightforward and will not be discussed in detail. The last two strategies are introduced here, and they will be discussed in detail in Chapter 9 and Chapter 10 respectively. These chapters will also present the tool support to perform software classification with the presented classification strategies, and they will discuss the application of the classification strategies in architectural recovery and in software evolution.

6.3.1 Manual Classification

Manual classification is the simplest classification strategy: manually putting items in classifications. The strategy can be employed by the software engineer to organise software entities according to his wishes.

Since the software classification model states that items can reside in multiple classifications, manual classification has a direct application in the creation of multiple views on software. For example, consider the modules view and the software layers views as presented in Chapter Four. With the manual classification strategy, the software engineer is able to create two classifications called 'Modules' and 'Software layers', with subclassifications called 'Module 1', 'Module2',... and 'User interface layer', 'Domain model layer', 'Persistency layer' respectively. The software engineer creates multiple views on the software by putting classes in these classifications. After classification, each classification represents an abstraction (module or software layer) that was not tangible before.

Manual classification is supported by the Classification Browser, which is introduced in Chapter 9. The browser provides commands to create and manipulate classifications, and to classify items in classifications.

6.3.2 Virtual Classification

Virtual classification is a software classification strategy to draw software entities of the software development environment into the software classification model. The strategy is based on virtual classifications in the software classification model.

Virtual classification is an automatic classification strategy. The strategy does not require human intervention. The items of a virtual classification are retrieved from the development environment, or they are computed from information retrieved from the development environment or the source code. Virtual classifications keep themselves consistent, that is, they re-retrieve or re-compute their items automatically when necessary. Retrieval or computation is necessary when the classification repository signals changes that affect the classifications[6].

Virtual classification will be used by the Classification Browser to provide classifications that serve as starting points for browsing.

6.3.3 Classification with Advanced Navigation Tools

When classification of items is based on the relationships between the items, a simple manual classification strategy is not sufficient. The idea behind *classification with advanced navigation tools* is that advanced navigation tools are necessary to browse the possibly large number of relationships between items. The advanced navigation tools not only provide excellent support for browsing/navigating relationships; they also provide extensive support for classification.

Basically, *classification with advanced navigation tools* is a manual classification strategy. Based on the results of browsing the relationships between items, the software developer decides whether items must be classified. He creates the classifications and he puts the items in those classifications.
The Classification Browser is an advanced navigation tool to browse interaction structures and to classify classes, methods, acquaintance relationships and method invocations. How the Classification Browser supports *classification with advanced navigation tools* will be explained in detail in Chapter 9.

6.3.4 Automatic Classification Through Method Tagging

In many cases, a manual classification strategy is not a feasible option. For large software systems it would take a long time to classify all classes by hand, especially when multiple classifications have to be created according to different points of view on software, of which some were given in Chapter 4. Often classification of a software system is an activity that cannot be done by one software engineer alone, since one software engineer seldom knows the whole system. From a managerial standpoint, such an endeavour is a costly operation, since it takes a lot of time and possibly many software engineers to accomplish it.

When manual classification is not a valid option for the classification problem at hand, automatic classification may provide a solution. The question then, however, is how classification can be done automatically.

The idea behind *automatic classification through method tagging* is that when software engineers carry out a development task, for example implementing a new or changed specification, or fixing a bug, they usually know the context in which changes are made. They know the module they are changing, the software layer a class belongs to, the

[6]When classifications are integrated in the software development environment, the classification repository should signal changes to the source code as well as changes to classifications. Section 9.1.1 discusses the relationship of the classification repository and the system in which it is integrated.

specification they are implementing, the bug they are fixing, etc. Normally, this knowledge is kept implicit in the heads of the software developers.
The classification strategy *automatic classification through method tagging* is based on making that knowledge explicit, by asking the software engineers to transfer that knowledge in the form of classification information when changes are made. The classification information is tagged onto the methods that have been changed. These tags are processed automatically to generate tag-based classifications. The classifications can then be used in future software development activities.

Since software engineers are usually lazy when (source code) documentation is concerned, relying on discipline is not realistic. It is up to the software development environment to make sure that classification knowledge about the software is recorded. Since development environments are not geared towards software classification, this classification strategy requires changes to the software development environment. The changes are needed to add support for entering classification information as tags, for processing the tags, and for storing the generated classifications in the classification repository.

Automatic classification through method tagging is inherently incremental. Each change to the software increases the (partial) classification knowledge about the changed software system. For example, suppose that the software development environment asks for the module of the class each time a class is subject to change. During development, classes are changed and the module information is recorded in the repository. The more classes are changed, the more information on their containing modules is available. The repository constantly grows until all classes have been changed one time or another. At that point, classification is finished. Note that while incremental classification is going on, the repository can already be queried, although for parts of the software only.

The generated classifications only hold information that is provided in the tags. Therefore, the choice of tags determines the applications of this classification strategy. Chapter 10 discusses this classification strategy in detail. It will present the tags that are used in the context of the broadcast management software, and it will discuss applications of the classification strategy in architectural recovery and software evolution.

6.4 Summary

This chapter has presented software classification. Software classification has two aspects: the software classification model and the software classification technique, the latter embodied by software classification strategies.

The software classification model is a metamodel. It consists of two parts: the participant model and the actual classification model. Each part focuses on another level of granularity. The participant model describes how participants in a collaboration contract are modelled. Largely, the participant model parallels the theory on collaboration contracts.

The model deviates from the theory with respect to traceability. Traceability information is explicitly present for participants and acquaintance relationships. A participant keeps a reference to the corresponding partially flattened class in the source code. Acquaintance relationships have an implementation stereotype that defines the link between the acquaintance relationship at design level and the way that relationship is established in the implementation.

While the participant model is concerned with modelling entities found in the source code, the classification model defines and describes the entities of the superstructure. The superstructure is a flexible organisational structure, based on classifications and items. A classification is a container for items. An item can be any software entity. Methods, classes, and classifications are obvious items, but the classification model is flexible enough to handle metrics, query results and other entities as well.

The participant model is embedded in the classification model by considering participants as items, and collaboration contracts and reuse contracts as classifications. Participants can be looked at as classifications as well. The model defines a participant as a view on a (partially flattened) class. The methods and acquaintance relationships in the view can be considered as being classified in the participant. The other ones are not classified.

Four software classification strategies were introduced. Two are straightforward: *manual classification* is a strategy to put together classifications manually, and *virtual classification* is a strategy to draw existing software entities in the classification model. *Classification with advanced navigation tools* and *automatic classification through method tagging* require specialised tool support. This chapter only gave an introduction of these two strategies. They will be discussed in depth in Chapter 9 and Chapter 10.

Chapter 7

Recovery of Collaboration Contracts

Reverse engineering collaboration contracts is a way to make object interactions buried in the source code tangible in the development environment. This chapter presents an incremental approach to recover collaboration contracts based on software classification, whereby a classification incrementally evolves from an informal aggregation of classes to a formal description of class collaboration. The chapter discusses issues involved with the recovery of collaboration contracts, gives an elaborate list of clues, guidelines, and heuristics to identify key classes and key collaborations, and treats technical matters in detail.
This chapter treats technical matters concerned with the recovery of collaboration contracts and paves the way for a classification strategy to recover collaboration contracts. The classification strategy itself, *classification with advanced navigation tools*, is discussed in Chapter 9.

7.1 Problem and Overview of the Proposed Solution

The fact that a collaboration contract is an abstraction of the source code gives rise to the following problems for which the reverse engineering method should provide solutions.

1. If collaboration contracts are to be extracted from the source code, how do they map on the source code and vice versa?

2. If a collaboration contract is an abstraction of the source code, what abstraction does it represent? Naively documenting the method invocations between a set of classes does not necessarily provide useful and meaningful collaboration contracts.

3. How are collaboration contracts discovered in the source code?

4. Is it feasible to extract collaboration contracts from the source code automatically? If not, what can be extracted and what must be supplied by the reverse engineer?

5. Part of the previous problem is interesting enough to treat it separately: the extraction of acquaintance relationships from source code. Finding the acquaintance objects of an object in the source code is relatively easy. Determining the class of the acquaintance object is another matter, however, especially in dynamically typed languages such as Smalltalk.

6. How should the reverse engineering process be supported by tools, and how do collaboration contracts show up in the development environment?

In order to solve these problems, an incremental approach to reverse engineering is proposed here. The problems stated above are addressed as follows:

1. In their current state, collaboration contracts cannot capture all aspects of the source code, but the limitations do not restrict their applicability much. Section 7.2 describes how concepts found in the source code map to concepts in collaboration contracts.

2. A class plays a role in several collaboration contracts. Each of these collaboration contracts corresponds to a different concern addressed by the class. Therefore, collaboration contracts contribute to separation of concerns [HL95]. Section 7.3 discusses the issues involved with recovery of collaboration contracts.

3. The collaboration contract recovery process is a process of source code examination. It is not easy to find useful and meaningful collaboration contracts in the large set of object interactions found in the source code. This chapter proposes an incremental approach to collaboration contract recovery based on software classification. An elaborate set of guidelines and heuristics for identifying concerns, key classes and key collaborations is given. Section 7.4 presents the overall method for recovery and sections 7.4 through 7.8 discuss the identification process.

4. Collaboration contracts cannot be extracted from the source code automatically, because the inclusion of classes as participants and the inclusion of method invocations is based on subjective decisions. However, a large part of a participant in a collaboration contract can be extracted from the source code: the interface, including the specialisation clauses, and even the acquaintance clause (also see the next item). Section 7.9 explains what can be extracted and how the extraction is achieved.

5. Determining the class of an acquaintance object found in the source code sounds like a typing problem. In a language without explicit types, such as Smalltalk, this poses a major problem that has been addressed by many [BI82, Joh86, JGZ88, Gra89, PS91, BG93, AH95]. Instead of employing a full-blown type inference engine, or requiring extra type annotations in the source code, a lightweight approach is proposed here. Section 7.10 explains the computation of the acquaintance classes based on the required interfaces of the acquaintance objects found in the source code. This approach is simple, yet produces good results. Moreover, it is fast, and it can also be applied on source code written in typed object-oriented languages.

6. The Classification Browser supports the incremental reverse engineering of collaboration contracts: participants in a desired collaboration contract are collected in a classification, and these participants are subjected to incremental refinement until a collaboration contract between the participants emerges. The classification that represents the collaboration contract makes the contract tangible in the development environment. The Classification Browser is presented in Chapter 9.

7.2 How does Source Code Map on Collaboration Contracts?

Before going into the issues involved with collaboration contract recovery, it should be clear how concepts found in the source code map on the concepts in a collaboration contract. This knowledge is necessary for best understanding of the subsequent sections.

In their current state, collaboration contracts can be employed to document the source code of an object-oriented language as follows.

Class interfaces map to participant interfaces. Participants are at the heart of the collaboration contract model, since they hold all the information. A participant that describes a class can be given the same name as the class. A participant is not equipped to hold information on instance variables. A participant has an interface; therefore, it is suited to describe the interface of a class. In many object-oriented languages the interface of a class is richer than can be described by the interface of a participant, since the latter only holds the method signatures and the methods' abstractness attributes. Language-specific concepts, such as access restrictions (e.g. public, private and protected in C++), restrictions on overriding (e.g. the final attribute in Java), and the like, are not recorded in a participant. On the other hand, participants in the reuse contract model have a richer interface structure than traditionally seen in object-oriented languages, since a method signature in the interface of a participant is annotated with a specialisation clause that records the messages in the method body.

Message sends in method bodies map to specialisation clauses. All messages in the body of a method are listed in the specialisation clause of that method. A specialisation clause does not impose any order on the message sends it lists. It just states that the listed messages may be sent when the method is executed.

Acquainted classes are listed in the acquaintance clause. The classes with which a class is acquainted, that is of which it requires services, are listed in the acquaintance clause of the participant that describes the class. Each acquainted class is described by a participant, of course. The acquaintance clause only states which participants are acquaintances of a target participant. It does not state any specifics about the acquaintance relationships. For instance, the acquaintance clause does not state the multiplicity of the acquaintance relationships.

Apart from the restrictions mentioned above, assignments in method bodies are not recorded in a reuse contract. Table 7.1 summarises the mapping from object-oriented language concepts onto collaboration contract concepts.

Language Concept	**Collaboration Contract Concept**
Class	Participant
Method in class	Method signature in interface of participant
Message sends in method body	Specialisation clause associated with method signature
Abstract method	Abstractness attribute of method signature
Acquainted class	Participant Acquaintance relation
Instance variables	*not recorded*
Assignments in method bodies	*not recorded*
Order of message sends/control flow	*not recorded*
Kind of acquaintance relations	*not recorded*
Other (language-specific) concepts	*not recorded*

Table 7.1: Mapping from language concepts onto collaboration contract concepts

7.3 Issues in Recovering Collaboration Contracts

Naively recovering collaboration contracts may lead to useless and meaningless collaboration contracts. Several issues have to be taken into account in order to produce ”good” collaboration contracts. The issues discussed here have an impact on the recovery process presented later.

7.3.1 A Collaboration Contract is a Unit of Reuse

In the reuse contract model, a collaboration contract plays the role of the provider clause in a reuse contract and a reuse contract states how that provider clause is reused. The collaboration contract is thus the unit of reuse.

Reusability increases when the units of reuse are small. The less dependencies between the different units, the more reusable these units are. This also holds for collaboration contracts. A large collaboration contract states many dependencies between classes. Reusing a large collaboration contract may require many adaptations, possibly breaching the contract's design. By keeping collaboration contracts small, they can be varied more easily. The collaboration contract model does not state anything about the size of a collaboration contract[1]. It is up to the software engineer to determine the ideal size of collaboration contracts.

[1]The research on the collaboration contract model suggests that collaboration contracts should not have more than seven participants, but no hard evidence exists that this suggestion is good or bad.

7.3.2 Where to Start and Where to Stop

When a software engineer is confronted with a piece of software in which he should find his way, it is not easy to find a starting point. Any available software documentation, like analysis documents, design documents, etc, may significantly help to find an entry point in a large and complex software system. When such documentation is not available — not a rare case in practice — the software engineer should have at his disposal a set of guidelines that help him to search for starting points.

When starting points have been found, the next question is where to stop. Simply adding all methods and method invocations found in the source code does not produce good results. Since collaboration contracts are units of reuse, one should strive for a set of collaboration contracts that describe the different aspects of a software system, that use each other to achieve global behaviour, and that can evolve separately.

7.3.3 Level of Detail

A software system can be looked at from different levels of abstraction. A software engineer likes to look at software from different angles, so he likes to have collaboration contracts that describe a software system at different levels of granularity.

For example, while collaboration contracts at the class level (participants represent classes) are useful, a software engineer likes to have collaboration contracts at module level (participants represent modules), or software layer level (participants represent layers) as well. This means that larger architectural building blocks need to be considered as components that invoke each other's operations. Very interesting in this context is the mapping from invocations between these components on invocations between objects that constitute those components.

In this work however, the recovery of collaboration contracts is restricted to the class level, although the concepts and the mechanisms to handle larger-scale components (classifications as components) are available. Recovery of collaboration contracts for larger-scale components is deferred to future work (see Section 13.3.1).

7.3.4 Coding Practice and Coding Conventions

A reverse engineering process cannot make assumptions about the quality of the examined source code. It should be robust towards bad coding practice. A good example is the bad use of super sends. The coding rule says: "A method x should only perform a super send of x, not of y." The use of bad super sends in a class indicates badly decomposed (and thus difficult to reuse) methods in the superclass. The reuse contract model expects methods to conform to the rule above. The reverse engineering process, however, should not break when a bad super send is encountered. Instead it should indicate the problem and propose a solution to make the source code agree with the conditions imposed by the reuse contract model.

Much software is crammed with coding conventions. Development teams often define naming conventions for classes, methods, variables and types. A reverse engineering process should exploit these conventions to the fullest.

7.3.5 Tool Support

Tool support is very important in reverse engineering, because it is very hard to manage the number of classes and dependencies in a large and complex software system manually.

Finding starting points for the recovery of collaboration contracts should be supported by giving clues in the user interface and by supplying powerful query facilities. Reverse engineering tools should not only support the recovery process. They should also provide a means to organise the results of the recovery process, so that the results can be consulted later, and so that they can be used for further recovery activities. Reverse engineering is a recurrent software development activity. Reverse engineering tools should thus be integrated into the software development tools, so that the software engineer can record and consult recovered documentation with the tools he is accustomed to.

7.4 Incremental Recovery Based on Software Classification

The overall method for collaboration contract recovery is based on the application of classifications. Classifications are used to recover collaboration contracts in an incremental way, from an informal aggregation of classes to a formal description of class collaboration. The idea behind incremental recovery is that a classification passes through four stages during collaboration contract recovery (also see Figure 7.1).

Stage 1: a classification of classes. The initial classification holds a set of classes found in the source code.

Stage 2: a classification of participants. The classification holds participants, each corresponding to a partially flattened class (see Section 6.2.3), and each having a partial interface of that partially flattened class.

Stage 3: a classification of acquainted participants. The classification holds participants, each with a partial interface of the corresponding partially flattened class, and each with an acquaintance clause (including traceability information in the form of implementation stereotypes).

Stage 4: a classification that represents a collaboration contract. The classification represents a collaboration contract. This means that each participant holds a partial interface of the corresponding partially flattened class, an acquaintance clause, and a specialisation clause for each method.

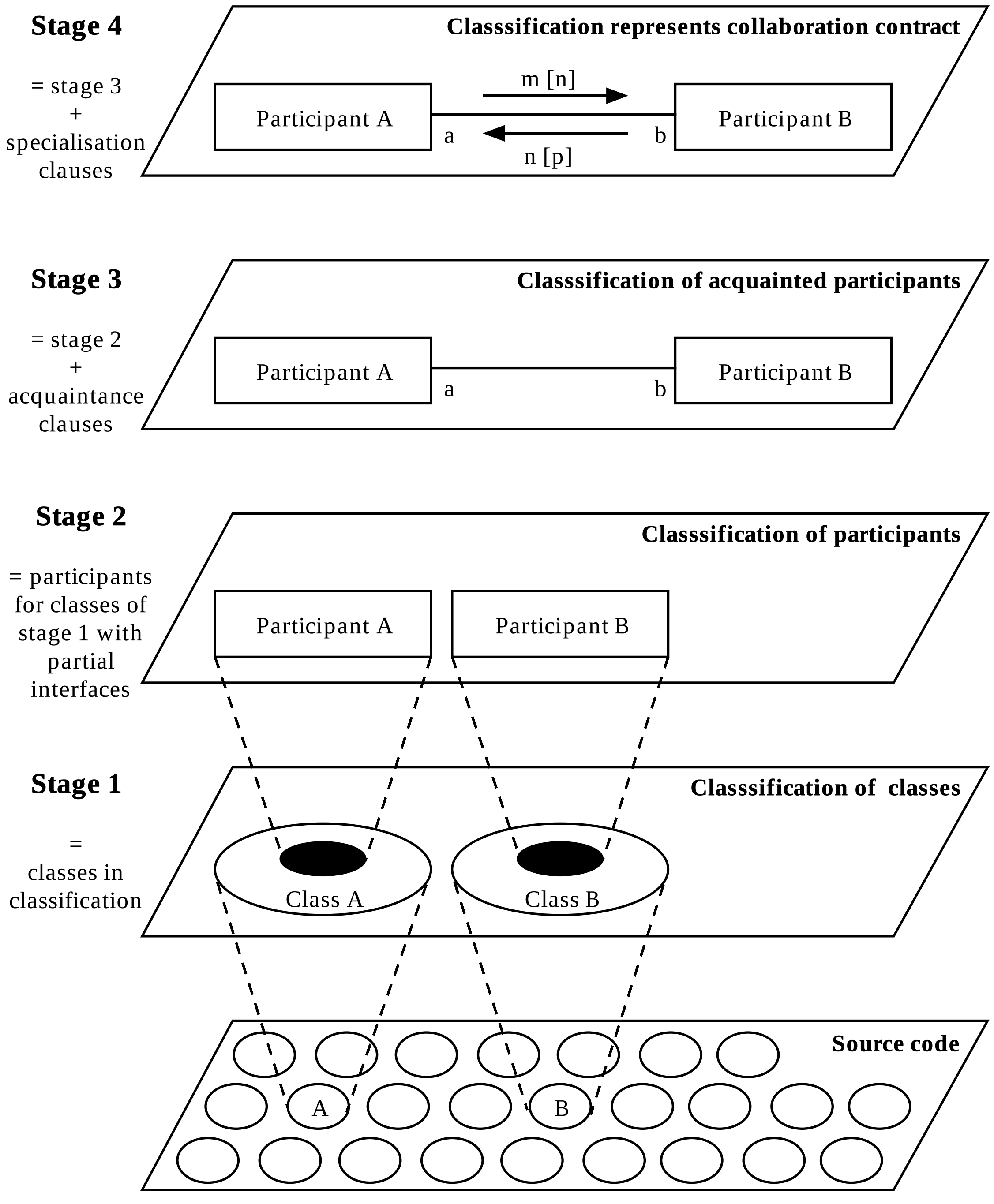

Stage 4
= stage 3 + specialisation clauses
Classsification represents collaboration contract
Participant A
m [n]
a
b
n [p]
Participant B
Stage 3
= stage 2 + acquaintance clauses
Classsification of acquainted participants
Participant A
a
b
Participant B
Stage 2
= participants for classes of stage 1 with partial interfaces
Classsification of participants
Participant A
Participant B
Stage 1
= classes in classification
Classsification of classes
Class A
Class B
Source code
A
B

Figure 7.1: Incremental recovery of collaboration contracts

In stage 1, the classification has no special semantics. It is nothing more than an informal grouping of classes found in the source code. Each transition from stage N to stage N + 1 produces an increasingly formal description of the target collaboration contract. The transitions are based on the following procedure:

> *The reverse engineer decides on the target participants, and edits their interfaces and their acquaintance clauses through classification. Specialisation clauses are not edited. They are computed automatically from the interfaces and the acquaintance clauses.*

The decision on the target participants results in the initial classification of stage 1.

Transition to stage 1. Collaboration contract recovery starts by identifying the concern of interest for which collaboration contracts are to be recovered. The process proceeds by identifying the classes that play a role in that concern. These key classes are then subjected to further examination. The identification of concerns is discussed in Section 7.5. The identification of key classes is discussed in Section 7.6.

Three transitions incrementally transform the initial classification of classes into a representation of a collaboration contract.

Transition 1 – 2. Since collaboration contracts hold participants, not classes, this transition is the first step to turn the informal grouping of classes into a classification that will be used to recover a collaboration contract. A participant is a view (a filter) on the corresponding partially flattened class. Its interface holds only methods that are of interest in the context of the target collaboration contract. To make the transition, first the classes from stage 1 are classified as participants in the collaboration contract (the classes are removed because they are not needed anymore). Then, methods are added to the interfaces of the participants by classifying them in the participants. The identification of methods involved in key collaborations is discussed in Section 7.7.

Transition 2 – 3. When the set of participants has been determined, the acquaintance relationships between the participants can be formalised. The source code level acquaintances referenced in the methods of a participant are added to the acquaintance clause by classifying them in the participant. For each acquaintance, traceability information is recorded in an implementation stereotype, so that a link between the acquaintance relationship and the source code is maintained. The acquaintance clause is completed after determining the acquaintance classes. For each acquaintance, the participant corresponding to its acquaintance class is recorded in the acquaintance clause. The identification of acquaintances is discussed in Section 7.8.

Transition 3 – 4. At this point, only the specialisation clauses of the methods in the participants must be determined in order to transform the classification into a complete representation of a collaboration contract. Based on the methods and the acquaintance relationships in the participants, the specialisation clauses extracted from the source code can be computed automatically. All source code level acquaintances that are not part of the acquaintance clauses, and all messages that are not part of the interfaces can be ignored. The remaining method invocations are recorded in the specialisation clauses of the participants' methods. A well-formedness check concludes the incremental recovery of the target collaboration contract.

Although the presentation above suggests a sequential process to recover collaboration contracts, in practice these transitions happen concurrently for different participants. After all, finding key collaborations may result in a better understanding of the software which in its turn may spawn concerns that were not known before. While browsing interactions between classes, a reverse engineer may find co-operating classes that were not known to co-operate, and the identification of these key classes may again lead to newly discovered concerns. Therefore, the process of recovering a collaboration contract cannot be divided into sequential steps. The steps usually take place concurrently while the reverse engineer is browsing the source code.
Consequently, the state of a classification might not be as clear-cut as presented here. Classifications may be in a state that corresponds to a mix of stage 2 and stage 3. Some participants may already have an interface and an acquaintance clause, while others may not.

The recovery process presented here, and elaborated in the next sections, is not a ready-made solution to reverse engineering collaboration contracts, as the decision to include or exclude participants and methods in a collaboration contract highly depends on the subject software and on subjective criteria. Recovering a good collaboration contract requires some experience, just as creating good classes requires some expertise. Although it is hard to define a clear-cut method for collaboration contract recovery, a set of guidelines and clues can be given to steer the recovery process. The following sections first discuss clues and guidelines that help with identifying concerns, key classes and key collaborations, and then explain which acquaintances and therefore which method invocations are interesting to be included in a collaboration contract.

7.5 Identifying Concerns

As indicated by Sections 2.8.2 and 2.8.3, the behaviour of a class often has many aspects, and each aspect corresponds with a role played in a collaboration contract. A collaboration contract consists of several participants, each playing their role. The collaboration between the participants expresses that the participants share a concern for which each provides part of the solution. The collaboration contract describes how the participants address that concern. The different concerns of a class are separated by formalising the different collaboration contracts in which the class plays a role.

For example, the concern 'undoing commands' corresponds to the task 'add an undo feature to the application'. All methods that have behaviour to set up an undo context (typically before an undoable command is executed) or that execute an undo command are part of the ‘undoing commands’ concern. The ‘undoing commands’ concern is an example of a concern that spans several classes (typically all command classes).

The ‘intent’ section in the pattern form that describes a design pattern typically states the task corresponding to a non-functional concern. For example, the intent of the State design pattern reads: "Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class." [GHJV94].

Since collaboration contracts address a concern, it is obvious to look for concerns during recovery of collaboration contracts. When concerns addressed by the software are known in advance, the recovery process can be driven by those concerns and collaboration contracts can be recovered in a top-down fashion. When no concerns are known, recovery proceeds in a bottom-up fashion: concerns are discovered by examining object interactions in the source code.

7.5.1 Top-Down Identification

Top-down identification of concerns is possible when concerns are known before source code examination starts. When documentation about the target software is available, it must be exploited to the fullest to identify concerns that may drive the recovery process. A concern may cover a large part of the software, so that decomposition of the concern is recommended. The documentation may or may not provide insight into possible subconcerns. If it does, the initial concern can be decomposed easily. If it does not, decomposition proceeds by source code examination.
Top-down identification can also be applied when the problem domain and the (top-level) concerns involved are known.

A good example of top-down identification of concerns comes from an experiment[2] to set up collaboration contracts for the user interface builder framework of VisualWorks\Smalltalk. Based on a developer's guide [How95], students have identified the major concerns involved with opening an application with a graphical user interface. Partial results are shown in Figure 7.2.
Documentation may not cover all concerns addressed by the software. In that case, bottom-up identification is necessary. Experience shows that the top-down and bottom-up approach are often applied together to identify concerns.
Top-down identification based on documentation often gives insight into the classes that address the identified concerns. This means that key classes have already been identified.

7.5.2 Bottom-up Identification

When no software documentation is available, concerns can only be deduced from the source code. This kind of identification is driven by identifying key classes (see Section

[2]Last year students were asked to document software with collaboration contracts. The software was unknown to them and they had to use the Classification Browser to recover collaboration contracts.

User interface opening
- User interface construction
  - Platform dependent UI construction
    - Widget construction
  - Platform independent UI construction
    - Component construction
    - Aspect support
    - Resource management
- Opening the User Interface

Figure 7.2: Example of top-down identification of concerns

7.6).

7.5.3 Extra Clues in Smalltalk

A Smalltalk system provides some information not found in other environments that may be exploited for the identification of concerns. While Smalltalk class categories are probably too coarse-grained to be good clues for identification of concerns, names of method protocols may be extremely helpful. Method protocols often have a name that corresponds to some aspect of the behaviour defined by the class that can be identified as a concern. Examples are: ‘printing’, ‘displaying’, ‘updating’. Many of the method protocols are found on many classes in the Smalltalk system, indicating that there are concerns that apply to many classes. Since all classes in Smalltalk inherit from the same root class, there are even concerns that are shared by all classes in the system. A good example is the ‘printing’ concern that refers to the methods `printString` and `printOn:`. If a developer likes to see a descriptive printable string in the development tools for instances of a class, the method `printOn:` on that class should be implemented. Another example is the ‘conversion’ concern expressed by the ‘converting’ protocol: it contains conversion methods (typically of the form `asXXX`) for doing arithmetic with numbers of different classes. If a new number class is added to the system, conversion methods must be added for the new class, as well as for the existing classes.
A method may play a role in several concerns, but a method in Smalltalk can be categorised in only one protocol, so method protocols do not always indicate all concerns addressed by a method.

7.6 Identifying Key Classes

When analysing a complex software system to recover collaboration contracts, it is hard to find a starting point. Identifying some key classes is the crucial first step in the design recovery process. Examining these key classes may result in more key classes. This section discusses the techniques that can be applied to find key classes.

A software development environment should be able to supply a software engineer with all the crucial clues that can be extracted from the source code to support him in his detective work. Most development environments do not supply that information, however, and therefore many of these techniques are based on heuristics and coding conventions.

7.6.1 Problem Domain Concepts

Many software developers use classes that map directly to concepts in the problem domain. Moreover, such classes typically carry a name that is the same as the name of the concept they represent. Classes that map directly to problem domain concepts are mostly concrete classes. Therefore, it is likely that such a class concretises or adapts a collaboration contract that is defined for an abstract or a concrete class on its superclass chain.

This observation leads to the following heuristics when the software engineer is concerned with understanding the domain model implementation: classes with names that map to names in the problem domain are probably important classes in the software architecture and are worth looking at when searching for collaboration contracts. This heuristics produces very good results in the presence of the original analysis documents, because the problem domain names are readily available. In absence of such documents, the software engineer must fall back on his experience and his general knowledge about the problem domain.

7.6.2 Abstract Classes

When looking for a design framework, the identification of abstract classes provides useful information. An abstract class has three kinds of methods: abstract methods, template methods and concrete methods. Abstract methods are methods without implementation. One abstract method is enough to make the containing class abstract. Template methods are methods that invoke abstract methods, directly as well as indirectly, and on the class as well as on other classes. Concrete methods are all other methods.

Template methods define the design of the abstract class: they lay down how the behaviour of the class' instances is defined in terms of abstract methods that should be defined by the subclasses of the abstract class. Due to the capability of defining an abstract design that should be followed by its subclasses, abstract classes play an important role in the architecture of a software system. Therefore, abstract classes are important to be considered when one searches for collaboration contracts.

Spotting abstract classes in a class hierarchy may not be easy, since development environments usually do not give any indication in that respect. The names of the classes may help, however. Many developers use names of abstract classes that contain the word ‘abstract’. Names of abstract classes typically refer to abstract concepts, or concepts that are generalisations of the concepts represented by the subclasses. When the team that developed the software has defined naming conventions, these would also help a lot in spotting abstract classes.

Although these are not foolproof heuristics, they might help when the software development environment fails to provide crucial information on the abstractness of classes and methods. Ideally, a development environment should provide a query mechanism to find abstract classes.

7.6.3 Classes with Many Subclasses

A class with many subclasses is a good starting point to look for collaboration contracts, because the fact that it has many subclasses means that the class has many variations. In general, these variations are reflected in variations in the collaborations with other classes.

Classes in a class hierarchy often interact with classes from a parallel class hierarchy. The Model-View-Controller framework [KP88] is a good example thereof: classes from three class hierarchies (Model, View and Controller) work together to define the behaviour of a widget. Increased reusability and the ability to evolve the different class hierarchies separately are the reasons behind the division of behaviour into several classes. The division of behaviour gives rise to an interaction between the different classes to achieve the global behaviour. That interaction is the collaboration contract in which the reverse engineer is interested.

The collaboration contract laid down by the root classes is subject to adaptation by their subclasses. Figure 7.3 shows a situation in which the interaction between the root classes of two parallel class hierarchies is defined in a collaboration contract. Two subclasses of the root classes adapt the collaboration contract to fit their needs and two other subclasses adapt the collaboration contract again.

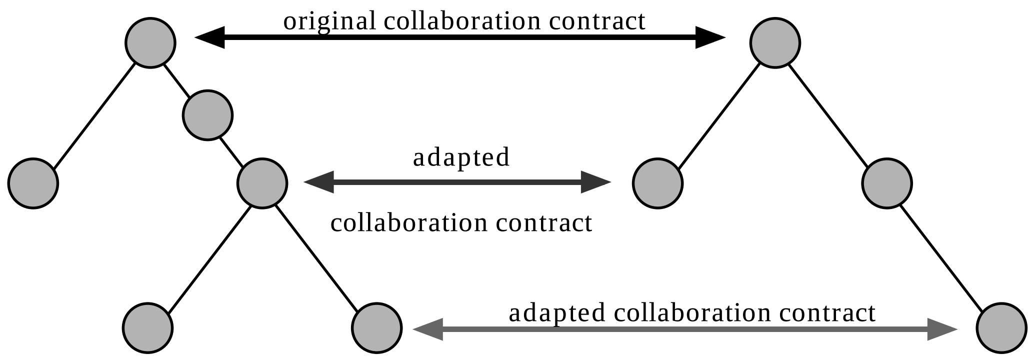

original collaboration contract
adapted
collaboration contract
adapted collaboration contract

Figure 7.3: Collaboration contracts often exist in parallel class hierarchies

Collaboration contracts can be adapted by implementing abstract methods, adding methods, and overriding methods. The latter adaptation indicates that identifying classes with many subclasses is linked with identifying frequently overridden methods (see Section 7.7.3).

7.6.4 Classes that Participate in Design Patterns

A design pattern is a recurring solution to a standard problem [SFJ96]. Design patterns are described according to the design pattern form, a structured documentation format in which each section describes a different aspect of a pattern. Two sections are of particular importance in this context. The 'Structure' section gives a graphical representation (class diagram) of the classes that participate in the pattern. It may also include interaction diagrams to illustrate object interaction. The 'Collaboration' section explains how the participants in the design pattern collaborate and divide responsibilities. The section often clarifies collaborations that can be found as pseudo-code in the structure diagram.

Therefore, design pattern catalogues are an important source of information for collaboration contract recovery. When software is documented with design patterns, even if that documentation is in fact nothing more than a list of class names connected to a design pattern name, this immediately sheds light on the key classes in the collaborations that exist between them, as described by the design pattern. The key collaborations can easily be read from the design pattern description (also see Section 7.7.5).

7.6.5 Classes or Instances Returned by Factory Methods

A factory method is a method that returns a class (in object-oriented languages in which classes are first-class citizens, such as Smalltalk) or an instance of a class (in all object-oriented languages, but certainly in languages in which classes are not first-class citizens, such as C++). Factory methods are introduced by developers to avoid hard-coding of class names in method bodies. The result is an increase in reusability, because subclasses can override the factory method and return another class or instance without having to change the sites where the factory method is invoked.

Factory methods are important because they explicitly state what the acquainted classes of the target class are. They are invoked to set up an acquaintance relationship. Factory methods are thus important methods to look at when searching for key classes for collaboration contracts.

As is the case for abstract methods, development environments usually do not indicate whether a method is a factory method. Therefore, developers look for factory methods based on heuristics. In object-oriented languages with first-class classes, factory methods typically carry names matching `xxxClass`, where `xxx` is the name of a concept (often a class name), for instance `programmeClass`. In object-oriented languages in which classes are not first class citizens, factory methods typically carry names matching `makeYYY`, `createYYY`, or `newYYY`, where `YYY` is the name of the concept (often a class name), for

instance `createProgramme`.
The factory method has been included as design pattern in the gang-of-four design patterns catalogue [GHJV94]. Due to the book's popularity, many developers use the naming conventions described there (and here), so that most factory methods are easily spotted in a class implementation.

7.6.6 Abstract and Concrete Factory Classes

Related to factory methods are abstract factory classes. An abstract factory (also a design pattern) provides an interface for creating families of objects, while a factory method provides an interface for creating a single object only. Abstract factories consist of several abstract factory methods. Concrete factory classes inherit from abstract factory classes and override each abstract factory method to return a family-specific class. One major advantage of using abstract factories is that the factory can be changed, even at runtime. This results in high flexibility.

Recognising concrete factory classes is crucial, because they list all the classes or objects that together form a family. Since classes in a family are usually closely related and collaborate to achieve some family related behaviour, concrete factory classes supply key classes that should be further explored in the search for collaboration contracts.

Finding abstract and concrete factory classes is again based on naming conventions. The name of a class that implements a factory typically matches `xxxFactory` or `xxxFamily` (also `AbstractXXXFactory` for an abstract factory), or has a name that corresponds to an important problem domain concept that defines a family. For instance, the class `TVSite` could be an abstract factory class for a family of site-specific classes with factory methods `programmeClass`, `weekViewClass`, etc.

7.7 Identifying Key Collaborations

7.7.1 Template Methods

When a template method invokes abstract methods of the same class, it implements a collaboration contract in which the class is the only participant. If a template method invokes abstract methods of other classes, it implements a collaboration contract with other classes.

Template methods define a piece of the abstract behaviour of their containing class. The fact that the abstract class defines how the method's behaviour is distributed over several methods and that some behaviour must be supplied by the subclasses (by implementing the abstract methods), means that a template method always implements (part of) a collaboration contract.

Spotting template methods is not easy, since the body of a method must be examined to determine whether one of the invoked methods is an abstract method. Ideally, a software development environment should give an indication whether a method is a template method.

7.7.2 Method Invocations Between Key Classes

Besides the invocations of abstract methods by template methods, in general method invocations between key classes should be examined to determine whether they should be included in the target collaboration contract. The choice to include an invocation is made by the software engineer, who decides based on his experience and his knowledge about the software system.

7.7.3 Frequently Overridden Methods

Looking for frequently overridden methods is connected with spotting classes with many subclasses, as described in Section 7.6.3. Frequently overridden methods in a class hierarchy indicate that some collaboration contract is implemented differently by many classes. When the subclasses do not conform to the collaboration contract defined at the top of the class hierarchy, the collaboration contract has many variations. The many variations make the collaboration contract very interesting for further examination.

7.7.4 Invocations of Methods with the Receiver as Argument

A method that sends a message to another object with the receiver (*self* in Smalltalk or *this* in Java) as argument, is a very important clue in finding a collaboration contract. When such a method is found, it is very likely that the invoked method sends a message back to the originating object. If that is the case, a collaboration contract between two classes has been found.

When an object sends a message to another object with itself as argument, the first object sets up an acquaintance relationship between the other object and itself. The acquaintance relationship that is established at message sending time may be prolonged, depending on what the two objects do with it. The following three cases can be distinguished for a message `object2 msg: self` sent by an object `object1`:

Volatile acquaintance relationship. If the argument of the message is not stored by `object2`, `object1` establishes a volatile acquaintance relationship from `object2` to `object1`.

One-way volatile acquaintance relationships are set up when objects delegate behaviour to other objects that need the sending object to achieve that behaviour. These acquaintance relationships are typically encountered after refactoring a large and complex class. The original behaviour of the complex class is spread over several new classes and the objects that delegate behaviour to the appropriate objects pass themselves along. This is also the basis for the implementation of several design patterns, such as State and Strategy for instance, and for double dispatching techniques in languages that do not support double dispatching directly.

Consider the simple Smalltalk example with two classes shown in Figure 7.4. Class `Employee` implements method `salary` which sends the message `salaryFor:` with the receiver (`self`) as argument to the result of the `company` message to itself. Class `Company` implements `salaryFor:` by sending `baseSalary` to its argument, so that the original employee object receives that message. This clearly shows the collaboration contract between class `Employee` and class `Company` to compute the salary of an employee.

```
Class Employee

salary
^self company salaryFor: self

baseSalary
^50000
```

```
Class Company

salaryFor: anEmployee
^anEmployee baseSalary
+ self raiseFor: anEmployee
```

Figure 7.4: The receiver as argument is a clue for finding a collaboration contract

One-way lifetime acquaintance relationship. If `msg:` is a mutator message (see Appendix B for the definition of accessor and mutator messages), and if `object1` does not keep a reference to `object2`, `object1` establishes a one-way lifetime acquaintance relationship between `object2` and `object1`. The acquaintance relationship established by sending `msg:` is lost. If `object1` should keep a reference to `object2`, then the next case applies.

One-way lifetime acquaintance relationships are set up when objects are composed into bigger wholes, typically through object creation, or when `object2` has to do something for which it needs a long-term binding with `object1`.

An example of the former case is found in the Adaptor design pattern. When `object1` creates an Adaptor for itself, it typically performs `Adaptor on: self`, where `Adaptor` is a class and `on:` is an instance creation message that returns a new Adaptor instance with `object1` stored in an instance variable. The acquaintance relation from the Adaptor object to `object1` exists for the lifetime of the Adaptor object.

An example of the latter case is found in the Builder design pattern. Builders are typically objects that exist for a short period in which building of a data structure is performed. When `object1` needs a Builder to build a datastructure based on data retrieved from it, it configures a new Builder object by sending it a mutator message with itself as argument. For instance: `aBuilder source: self`, where `aBuilder` holds an instance of the Builder class. The acquaintance relation from `aBuilder` to `object1` exists for the lifetime of `aBuilder`.

Mutual lifetime acquaintance relationship. If `msg:` is a mutator message and if `object1` keeps a reference to `object2` as well, a mutual lifetime acquaintance relationship is established. Mutual lifetime acquaintance relationships are set up when the composition of the objects represents a bigger whole of which the behaviour is distributed among the objects, but requires a tied co-operation to achieve it.

A well known example comes again from the Model-View-Controller triad. A View and a Controller are intimate partners in achieving input and output behaviour of a widget. When a View is instantiated, the new View instance creates the corresponding Controller object and stores it in an instance variable. The View object then asks the Controller object to store it through `self controller view: self`. From that moment, the two objects know each other for life.

The cases above clearly indicate that invocations of methods with the receiver as argument are important in object composition and delegation. The recursion makes these object interactions very interesting for examination during collaboration contract recovery. Spotting the receiver as argument requires method body examination. Ideally, a software development environment should provide a query mechanism to find them, or at least give an indication of receiver arguments.

7.7.5 Collaborations in Design Patterns

As already mentioned in Section 7.6.4, design pattern catalogues clearly state what the major interactions between design pattern participants are. If it is known that some classes play a role in a design pattern, the interactions found in the catalogue help to find the corresponding method invocations in the source code. These invocations are part of the collaboration contract, or contracts, that are (implicitly) present in the design pattern. Note, however, that the interaction structure found in the design pattern catalogue does not necessarily have a one-to-one correspondence in the source code. Design patterns can be implemented in several ways, often only partially. The result is that the target collaboration contract may deviate from the interaction structure suggested by the design pattern.

Finding classes that play a role in a design pattern may not be simple. Naming is again an important indication. For instance, classes that play the role of the composite in the Composite design pattern often have names starting or ending with "composite". The same holds for "policy" or "strategy" in names of classes that play the role of the strategy participant in the Strategy design pattern. Other ways to find classes participating in design patterns relies on clues found in the source comments or any other form of documentation, and (as always) the general knowledge of the software system.

7.8 Identifying Acquaintances

A collaboration contract does not hold classes; it holds participants that represent roles played by classes. Each participant knows other participants through acquaintance relationships. Methods in a class may send messages to acquaintance objects of which many may not be of importance for a collaboration contract. This section discusses which acquaintance objects are important to include as participants and which are not. Acquaintance relationships are name – participant pairs. This section also proposes a naming scheme for the participants to which a given participant is acquainted.

7.8.1 Source Code Level Acquaintances

Acquaintance relationships play a very important role in collaboration contracts, because they define the participants to which a participant is allowed to send messages.

In order to document acquaintance relationships, it is necessary to define clearly what an acquaintance is. According to Agha [Agh86], the acquaintances of an object are all objects the object has knowledge of, or can directly refer to. In an object-oriented language, acquaintance relationships can be realised in several ways: through instance variables, through method arguments, through temporary variables, through global variables, and through object creation. Objects can be created in two ways: by sending an instance creation message to a class and by writing down a literal object. Acquaintance relationships also originate from message sending, where the returned object is an acquaintance of the sending object. The Law of Demeter stresses the importance of recognising different kinds of acquaintances.

The Law of Demeter (class form) [Lie96].
Inside an operation O of class C we should call only operations of the following classes, called preferred supplier classes:

- the classes of the immediate subparts (computed or stored) of the current object;
- the classes of the argument objects of O (including the class C itself);
- the classes of objects created by O.

So when programs conform to the Law of Demeter, the only acquaintances an object is allowed to have are held in the object's instance variables, passed as method arguments, returned by self sends, and created by the object.

Not many programs obey the Law of Demeter and consequently many object-oriented programs are littered with expressions such as `self window topComponent menubar someMessage`.

The question that immediately rises is what acquaintance relations are important here for the collaboration contract that is to be reverse engineered from the source code. Is only the resulting object from the compound message expression `self window topComponent menubar` important, or are all intermediate objects (the results from `self window` and `self window topComponent`) important as well? If all intermediate objects are important, they all need a name for reference in the collaboration contract.

Consider `self window topComponent menubar height: 25`. This message expression consists of several messages. The top level message `height: 25` is sent to the result of three consecutive messages: `window`, `topComponent` and `menubar`. Looking at this message expression in isolation, we assume that the sender of the compound message is not interested in the results (objects) of the intermediate messages. The sender is only interested in the object to which it sends the `height:` message. Therefore, the resulting object from the `menubar` message is the only acquaintance of importance to the sending object. The compound message expression `self window topComponent menubar` is the way the acquaintance relationship between the sending object and its acquaintance is set up.

When considering the whole scope in which such compound message expressions, or parts thereof, can occur, it is possible that the results of the intermediate messages become important to be considered acquaintances as well. For instance, when the same, or another method contains the message expression `self window topComponent width: 150`, the result from the compound message expression `self window topComponent` is also an acquaintance of the sending object.

The above discussion leads to the following definition of an acquaintance for the purpose of software documentation by means of collaboration contracts.

Definition 2 (Direct acquaintance of an object)

A **direct acquaintance of an object O of class C** is one of the following:

- the object O itself;
- objects held in O's instance variables;
- objects that result from sending messages to O (i.e. results of self sends);
- objects passed as arguments of methods in C;
- newly created objects in methods in C;
- objects stored in temporary variables in methods in C;
- objects stored in global variables.

Note that this definition defines the direct acquaintances of an object to be the instances of the preferred supplier classes, as defined by the Law of Demeter, together with the objects stored in temporary and global variables.

Definition 3 (Indirect acquaintance of an object)

An **indirect acquaintance of an object O of class C** is the object that is the receiver of the message `msg`$_{n+1}$ in a compound message expression `<direct acquaintance>` `msg`$_1$ `msg`$_2$ ...`msg`$_n$ `msg`$_{n+1}$ in a method in C, according to the following conditions:

- `<direct acquaintance>` is a reference to a direct acquaintance, different from O itself;
- `<direct acquaintance>` `msg`$_1$ `msg`$_2$ ...`msg`$_n$ `msg`$_{n+1}$ is a top level message, that is, it is not the receiver of a message expression;
- $n \geq 1$.

This definition does not say anything about the receivers of the intermediate messages. The receivers of messages `msg`$_2$, `msg`$_3$, ..., `msg`$_n$ are thus not considered to be acquaintances. There are no indirect acquaintances if all methods of a class obey the Law of Demeter.
These two definitions define the acquaintances of an object as being the objects to which the object is able to send messages. The direct acquaintances are the objects to which the object can refer directly in a method, while the indirect acquaintances are all the objects that are results of messages sent to the direct acquaintances.

Definition 4 (Acquaintance)

An **acquaintance of an object O of class C** is a direct acquaintance of O or an indirect acquaintance of O.

Definition 5 (Acquaintance Class)

An **acquaintance class of a class C** is the class of an acquaintance of any instance of C. The term acquaintance class is qualified with *direct* and *indirect* to refer to classes of direct acquaintances and indirect acquaintances, respectively.

7.8.2 Categorisation of Source Code Level Messages

The previous section defined what the source level acquaintances of an object are. Many source level acquaintances are not interesting enough to be included in a collaboration contract, however. Since collaboration contracts are concerned with a collaboration between participants, it is natural to define acquaintances as the receivers of the messages that are to be considered for inclusion in a collaboration contract. The question then, of course, is what messages are important to be included in a collaboration contract. This question cannot be answered easily, because the answer strongly depends on several subjective considerations: the desired level of detail, decomposition in several collaboration contracts, choice of the concern to be documented, etc. What can de done, however, is to state which messages should certainly not be included in a collaboration contract. The complementary messages are then the only ones that should be considered for inclusion. This section categorises the messages that can be found in the source code and discusses why certain messages should not be included. The next section defines collaboration contract acquaintances based on this categorisation.

For the purpose of documenting software with collaboration contracts, we recognise the following categories of message expressions.

Navigation messages. When source code does not conform to the Law of Demeter, the source code often contains compound message expressions. An important observation is that many of these compound message expressions are in fact expressions to navigate an object structure, where each message in the expression is an accessor message (see Appendix B for a definition of accessor messages). Navigation code is bad for reuse, because it creates unnecessary data dependencies among objects. It is also bad for maintenance, because by definition navigation code is scattered all over the source code, thereby giving responsibilities to clients that should only reside with the object. Despite these clear disadvantages, navigation code is frequently encountered in software, making it an important factor to reckon with in the collaboration contract recovery process.

Navigation messages are used to set up acquaintance relationships, and therefore they seldom contribute to a collaboration. Navigation messages are implemented by accessor methods. Accessor methods come in several flavours, but their common property is that they return an object without taking part in any interaction with other objects. Since they are commonly used to access a stored object, accessor methods are seldom overridden in a subclass, making it uninteresting to include them in a collaboration contract.

Since navigation code usually does not contribute to the essence of a collaboration contract, navigation code should be recognised in the source code so that inclusion in a collaboration contract is avoided.

Messages to rock-bottom objects. In pure object-oriented languages, such as Smalltalk, rock-bottom objects, such as numbers, characters, strings and Boolean values are also objects. In that case, rock-bottom objects are identified as source code level acquaintances. However, rock-bottom objects do not take part in collaborations with other, non-rock-bottom objects, due to their general-purpose nature. Since rock-bottom objects do not interact with other objects except other rock-bottom objects, and since messages to rock-bottom objects return other rock-bottom objects, messages to rock-bottom objects are not interesting for inclusion in a collaboration contract.

Note that rock-bottom objects do not exist in impure object-oriented languages. Consequently, messages to such object are not found there and will never show up in a collaboration contract anyhow.

Messages to literals. Literals generally denote rock-bottom objects. Therefore, they belong to the previous category, which means that messages to literals should not be included in a collaboration contract.

Control structure code . Some object-oriented languages express control structures by means of message passing. Receivers of control structure messages are typically Boolean values, thus rock-bottom objects. In the current state of the reuse contract model, in which iteration and branches in the control structure are not addressed, control structure messages should not be part of a collaboration contract.

Object creation code. Some object-oriented languages express object creation by means of sending an instance creation message to a class. Whether instance creation messages should be included in a collaboration contract depends on the target software. When one of the Abstract Factory and Factory Method design patterns [GHJV94] is used in the implementation, it is probably a good idea to include the instance creation messages. In other cases, where the class name is hard coded in the source code, the instance creation message probably has no bearing on the collaboration contract and can be omitted.

Interaction code . All messages that cannot be categorised in one of the categories above are part of the interaction code. These messages should be considered for inclusion in a collaboration contract. Inclusion or omission depends on several aspects of the desired documentation: level of detail, decomposition in several collaboration contracts, choice of the concern to be documented.

7.8.3 Collaboration Contract Acquaintances

Given the categorisation of messages in the previous section, it is now time to define which source code level acquaintances are candidates for inclusion in a collaboration contract.

Definition 6 (Candidate collaboration contract acquaintance)

A source code level acquaintance Q is a **candidate collaboration contract acquaintance** when it is sent at least one interaction message.
This means that:

- Q receives at least one message
- Q is not a literal
- Q is not the receiver of a control structure message
- Q is not the result of navigation code

According to this definition, the following source code level acquaintances have no bearing on the essence of a collaboration contract:

Acquaintances not receiving messages. Source code level acquaintances of an object O that do not receive messages clearly do not play a role in the interaction in which O is involved. Therefore, it seems obvious not to include them in a collaboration contract.

In practice, excluding method arguments that receive no messages may be problematic. After all, when methods in a collaboration contract have arguments, they are intended to be used. Classes that define a collaboration contract in which no messages are sent to a method argument, typically lay down an (abstract) collaboration contract that is to be adapted by their subclasses. In the adapted collaboration contracts, the acquaintance may play a role after all.

Acquaintances being rock-bottom objects. Since messages to rock-bottom objects are considered as unimportant interactions, the objects cannot be acquaintances in a collaboration contract.

Receivers of control structure messages. As explained before, control structure code should not be part of a collaboration contract. Receivers of control structure messages should thus not be included as acquaintances in a collaboration contract.

Acquaintances obtained by navigation code. Intermediate results of compound message expressions consisting of navigation messages should not be included as acquaintances in a collaboration contract.

So, when at least one message is sent to an instance variable, method argument, temporary variable, global variable, or to the result of a non-navigation message, these objects are considered acquaintances of the sending object.

If no messages are sent to these objects, they are not considered to be an acquaintance, and will not be recorded in a collaboration contract. This choice deviates from the definition given by Agha [Agh86], which defines an acquaintance to be any object that can be referred directly regardless of whether any message is sent to the object.

For compound messages, the receiver of the top-level message is the acquaintance, unless other top-level messages are sent to intermediate receivers. In the example `self window topComponent menubar height: 25`, the receiver of the `height:` message is an acquaintance of the sending object, because `height:` is a top-level message. The receivers of the messages `menubar` and `topComponent` are not acquaintances, unless other top-level messages have the same expressions as receiver. So, if an expression `self window topComponent someMessage` can be found in the rest of the source code of the class under consideration, the receiver `self window topComponent` is an acquaintance.

Table 7.2 summarises when source code level acquaintances are to be considered as collaboration contract acquaintances.

	Source Code	**Acquaintance When**
Self	`self msg`	receiver of at least one message in class
Super	`super msg`	receiver of at least one message in class
Instance variable	`iv msg`	receiver of at least one message in class
Global variable	`g msg`	receiver of at least one message in class
Argument of method m	`a msg`	receiver of at least one message in m
Temporary variable in method m	`t msg`	receiver of at least one message in m
Literal object in method m	`l msg`	receiver of at least one message in m
Newly created object in method m	`C new msg`	receiver of at least one message in m
Result of message expression	`x y z msg`	receiver of at least one top level message in class

Table 7.2: Mapping from source code to acquaintances

7.8.4 Acquaintance Names

Messages to acquaintances are recorded in the specialisation clause of the method in which the message send resides. A specialisation clause is a collection of acquaintance name – method invocation pairs. When the body of a method is examined, it is not always clear what an appropriate acquaintance name for the receiver of a message is. On top of that, acquaintance names should be unique within a participant, since acquaintance names are also used in the acquaintance clause of the participant and the acquaintance clause is globally defined for the participant and thus for all methods and specialisation clauses in that participant.

Self and super. For self and super sends it is clear what the acquaintance name of the acquaintance object is, namely 'self' or 'super'.

Instance variables . When a message is sent to an instance variable, the name of the instance variable can serve as acquaintance name, because instance variable names are unique within a class and all its superclasses and subclasses.

Global variables. The name of the global variable can serve as acquaintance name, because it is unique in all source code under consideration.

Method arguments and temporary variables. Since the names of method arguments and temporary variables may not be unique over all methods in a class, the name of a method argument or temporary variable cannot serve as acquaintance name. Another way of identification is needed. A simple, but practical solution is to take the combination of the method signature and the method argument or temporary variable name as the acquaintance name.

Literals and newly created objects. Literal objects are not referred to by name, so another way of referral is required. The same holds for newly created objects. Since the class of a literal is usually defined by the language, and the class of the newly created object can be read from source code, a solution to the absence of a name is to take the name of the class concatenated with some index. For a newly created instance of class Person, for example, one could use the name 'aPerson', under the condition that only one instance of the Person class is created. If more than one instance is created in the same method, a numbering scheme could be adopted: 'Person1', 'Person2', ... The combination of the method signature and the instance's name gives a unique acquaintance name. For some literals, the textual representation of the literal's value could be taken[3] as unique acquaintance name in a method.

Result of a self send. The name of the message cannot serve as acquaintance name, because the name may clash with the acquaintance name of an instance variable. Prefixing the method signature with 'self' using the dot notation gives a unique acquaintance name.

Result of a compound message expression. Similar to the previous case, a unique acquaintance name for the result of a compound message expression (being an indirect acquaintance) can be formed by combining the acquaintance name of the initial receiver (which is a direct acquaintance) with all method signatures of the messages in the message expression using the dot notation.

[3]This is actually the way numbers, Boolean values, strings, and nil are displayed in the Classification Browser (see Section 9.1.5).

Acquaintance	**Message Expression**	**Acquaintance Name**	**Notation in Specialisation Clause**
Self	`self msg`	self	self.msg
Super	`super msg`	super	super.msg
Instance variable	`iv msg`	iv	iv.msg
Global variable	`g msg`	g	g.msg
Argument of method m	`a msg`	m.a	m.a.msg
Temporary variable in method m	`t msg`	m.t	m.t.msg
Literal object in method m	`l msg`	aC	aC.msg
Newly created object in method m	`C new msg`	aC	aC.msg
Result of self send	`self x msg`	self.x	self.x.msg
Result of compound message expression to a direct acquaintance d	d m_1 ...m_n msg	d.m_1. ....m_n	d.m_1. ....m_n.msg

Table 7.3: Acquaintance names

Table 7.3 summarises how acquaintance names are chosen from message expressions found in the source code, and how these message expressions map to invocations listed in the specialisation clause of the containing method.

7.9 Extraction From the Source Code

In order to give tool support for incremental collaboration contract recovery, many collaboration contract concepts have to be extracted from the source code. This section explains what can be extracted automatically from the source code and how the extraction is achieved.

According to the definitions given in Section A.2.1, and as explained in Section 7.2, the following information needs to be extracted from the source code, and mapped to collaboration contract concepts (also see Table 7.1 on page 78):

- classes
- methods defined in a class
- the abstractness attribute of methods
- self sends and super sends
- message sends to acquaintance objects

- acquaintance relations

The following subsections explain how this information can be extracted from the source code.

7.9.1 Extraction of the Participant Interfaces

The first three items are concerned with participants and their interfaces. In file-based object-oriented languages, such as C++ and Java, these items can be found in the parse tree provided by a parser. In Smalltalk, the definition of a class and its interface are stored in the Smalltalk image and can be consulted without parsing. Due to the Smalltalk convention that an abstract method sends the message `subclassResponsibility`, one should expect that determining the abstract methods of a class requires parsing all methods in the class. However, checking whether a Smalltalk method is abstract can be performed far more efficiently by inspecting the compiled code of the method[4]. No parsing is involved.

The name of the extracted participant is the same as the name of the class from which it is extracted. The method signatures in the participant's interface and their abstractness attribute are the same as in the class.

7.9.2 Extraction of the Interaction Structure

Self sends, super sends, and message sends to acquaintance objects in a method can be found by inspecting the parse tree of that method. All message sends are part of the specialisation clause of the method under consideration.

7.9.3 Extraction of Acquaintance Relations

While finding message sends to acquaintance objects is relatively easy, finding the actual acquaintance relations is another matter, because the classes of the acquaintance objects need to be determined. In dynamically typed object-oriented languages, such as Smalltalk, the lack of typing information hinders detection of acquainted classes. Consequently, extraction of acquaintance relations cannot be fully automated and must be performed with human assistance. The good news is that with proper tools the human assistance is only required to make choices between options presented by the extraction tool, as we will see later.

Since the work discussed in this dissertation had to be validated in a Smalltalk environment, a means for extraction of acquaintance relations was required. While in statically typed languages the types of the instance variables and method arguments give a good indication of the acquainted classes, in Smalltalk this information is not available and must be extracted from the source code somehow. In Section 7.10, a lightweight acquaintance

[4]Finding senders of a message in Smalltalk is achieved by inspecting the byte codes of all methods. The names of the messages are stored as symbols in the byte code stream.

class inference scheme will be introduced to find acquaintance relations. The approach does not require extensive type inference, nor explicit type annotations, which means that the technique can be used for programs written in explicitly typed languages as well as in implicitly typed languages.

7.9.4 Traceability of the Corresponding Flattened Class

To record the trace from a participant to its corresponding flattened class, the participant is annotated with that flattened class (see Section 6.1.2). The following notation is used: `<participant name> : <subclass name>...<class name>`. It indicates the chain of classes of which the participant is a flattened representation. `<class name>` is a class on the super class chain of `<subclass name>`.

7.10 Determining Acquaintance Classes In Smalltalk

According to the model, an acquaintance relationship is a pair consisting of an acquaintance name and a participant name, expressing how the participant involved in the relationship refers to another participant (see Section 6.1.3). Participants in a collaboration contract represent (partial) classes in the source code, so in order to determine an acquaintance relationship, the class of the acquaintance must be determined.

Determining acquaintance classes does not require a full-blown type inference engine, as one may expect (see Section 12.3 for related work on type inference). A lightweight algorithm to determine the class of an acquaintance is sufficient. The algorithm proposed here collects the set of messages sent to an acquaintance, and finds the classes that implement those messages. The reason for choosing such lightweight algorithm is threefold:

Speed. Acquaintance classes are computed frequently during collaboration contract recovery. Therefore, the computing algorithm should be fast. The proposed algorithm is based on set inclusion: the set of messages sent to an acquaintance is compared with the interfaces of the classes in the program. Intentionally, the algorithm does not perform data-flow analysis, since data-flow analysis is often a major performance bottleneck. Due to the absence of data-flow analysis, the lightweight algorithm contrasts strongly with type inference algorithms. The price that must be paid, of course, is that the acquaintance class cannot be deduced for acquaintances that receive no messages. In general, such acquaintances are not added to a collaboration contract, since they do not contribute to the collaboration anyway. In case such acquaintance must be added[5], other means for determining the acquaintance class must be used, such as exploiting dynamic information for instance.

Tuning ability. The fact that the algorithm is lightweight means that it can be enhanced easily with extra heuristics, coding conventions and other rules. Coding conventions

[5] In the course of this research, we have not encountered one example where such an acquaintance had to be added to the collaboration contract.

may influence the number of acquaintances. For example, coding conventions about accessor methods (see Appendix B) may state that an accessor message and the corresponding instance variable in fact denote the same acquaintance. Heuristics may influence the number of matching classes, based on knowledge about special messages that identify certain classes. The combination of the message `isNil` and the class `UndefinedObject` is an example thereof.

Applicability. An algorithm to compute acquaintance classes should not impose restrictions on the source code, as do many type inference schemes for Smalltalk. Imposed restrictions on the source code limit the applicability of the algorithm, which makes the algorithm impractical. The proposed algorithm puts only one but fair limitation on the source code: it must be in a form that can be parsed. It does not rely on typing information. Since the algorithm produces the best match that can be found according to the set of message sends found in the source code, it can be used for typed object-oriented languages as well as for Smalltalk. Moreover, it can be used for typed object-oriented programs that are not statically type-correct.

7.10.1 A Lightweight Approach

Our approach to computing acquaintance classes is based on the observation that objects are implicitly typed by the messages they receive and therefore must understand. The set of messages sent to an object defines the interface that is expected from it, the *required interface* (terminology borrowed from work in component-oriented programming [HLS97]).

Definition 7 (Required interface of an acquaintance)

The **required interface of an acquaintance** is the set of messages that is sent to the acquaintance in all the methods in the scope of that acquaintance.

For instance variables, the scope is the defining class and all its subclasses, so the interesting methods are all the methods in the defining class and all its subclasses. For temporary variables, the scope is the method in which they occur. For class variables, the scope includes all methods in the defining class and all its subclasses, including the meta-classes. For global variables, the scope includes all methods in the Smalltalk system.

Note that the class of an acquaintance can change in a subclass of the class in which the acquaintance is referred. This means that all methods of a class (being the methods defined by the class and all methods of its superclasses) referencing an acquaintance must be enumerated in order to correctly determine the class of the acquaintance, and thus to capture possible re-definitions of acquaintances in subclasses of the defining class.

Based on the required interface of an acquaintance the set of conforming classes can be computed, and a best match can be determined. The following sections discuss this process in detail. For some source code level acquaintances, inferring the class is overkill, because the class can be determined directly, as it is stated in the definition of the language. The class of an acquaintance representing a class reference is the class' class, thus the meta-class. The class of a literal in Smalltalk is determined by the language, so that gives the following table.

Literal Example	**Class**
`nil`	UndefinedObject
`-100`, `0`, `5`	SmallInteger
`0.31415d2`	Double
`3.1415`, `0.0`	Float
`true`	True
`false`	False
`#(1 2 3 4)`	Array
`#[5 6 7 8]`	ByteArray
$A, $z	Character
`'Hello'`	ByteString
`#reuse`, `#'reuse contract'`	Symbol
`[:arg| arg printString]`	BlockClosure

Table 7.4: Classes of Smalltalk literals

7.10.2 Collecting the Required Interface

Collecting the required interface of an acquaintance is straightforward. All source code in the scope of the acquaintance must be analysed. By enumerating the parse trees of all methods referencing the acquaintance, the messages sent to the acquaintance can be collected easily.

7.10.3 Computing the Conforming Classes

When the required interface of an acquaintance is determined, the system can be queried for the classes that conform to the required interface, that is, the classes of which the interface is a superset of the acquaintance's required interface. The set of conforming classes is potentially large, in the worst case holding all classes currently in the target Smalltalk system, in the best case holding one class. Four cases can be distinguished:

The set of conforming classes is empty. This means that no class in the system conforms to the required interface. This happens when either the acquaintance may be bound to objects of different classes and meta-level messages are used to check their interface before sending them interface-specific messages, or either when the program is incorrect. The latter situation represents an error condition.

The set of conforming classes contains all classes in the system. This means that the required interface is shared by all classes. This happens when the required interface only includes methods implemented by all the root classes in the system[6].

The set of conforming classes contains one class. This is the ideal situation. This happens when the required interface is a combination of methods that is unique to the class.

The set of conforming classes contains several classes. This means that either all classes of one subhierarchy of the class hierarchy conform (this is a generalisation of the previous case, in which the subhierarchy consists of only one class), or either all classes of separate subhierarchies conform. The latter typically happens when the required interface contains methods that have generic names that are used polymorphically across subhierarchies, for instance `name` or `value`.

7.10.4 Determining the Best Match

A reverse engineer is not interested in all the classes that conform to the required interface of an acquaintance, but in the best match that can be found; preferably one class. To trim down the set of conforming classes to a best match, the common superclasses of the conforming classes are determined.

The common superclass of a set of conforming classes is the class, as high up in the class hierarchy containing all conforming classes, that still conforms to the required interface. The common superclass is thus an approximation of a set of classes *with respect to the required interface*. For example, when the required interface consists of the Smalltalk messages `select:` and `collect:`, the set of conforming classes will include all collection classes. The best match will be the class `Collection`, because it is the common superclass that still conforms to {`select:`, `collect:`}.

The purpose of calculating the common superclass is to reduce the number of best matches for the acquaintance class. Note that a set of conforming classes often has several common superclasses, if the conforming classes reside in several separate class hierarchies. In the worst case, no common superclasses can be found, which means that the conforming classes themselves are the best match.

Based on the set of conforming classes, a best match is determined as follows and as shown in Table 7.5.

The set of conforming classes is empty. No best match can be determined.

The set of conforming classes contains all classes in the system. The set of root classes in the system is the best match.

The set of conforming classes contains one class. The best match is that class.

[6]Some programming languages/systems allow more than one root class. In Smalltalk, Object usually is the only root class, but it is possible to create other root classes.

The set of conforming classes contains several classes. The set of common superclasses of the conforming classes is the best match.

Conforming Classes in Name Space	Best Match
None	–
All classes	Root classes (Object)
All classes of one or more separate subhierarchies	Roots of separate subhierarchies
One class	The class

Table 7.5: Best match for an acquaintance class

7.10.5 Enhancing the Search for Conforming Classes

When applied on a whole Smalltalk image, the lightweight algorithm may select all classes as candidate acquaintances if the required interface only lists methods understood by all classes. Even when a small set of conforming classes is found, it may happen that several separate class hierarchies are found to conform to the required interface. This happens when the classes in the different hierarchies have overlapping interfaces.

The obvious solution is to reduce the name space in which the classes reside. By focussing on a subset of all the classes in the Smalltalk system, the results get much better, since less classes have to be examined and less classes conform to a given required interface.

Introducing a name space is actually a natural step to take, because most of the time several subhierarchies of the class hierarchy do not interact with each other. A name space thus provides a way to keep non-collaborating classes out of focus. A development environment may provide mechanisms that can serve as name space. Envy/Developer, for instance, organises classes in Envy applications and Envy applications in configuration maps. Configuration maps define configurations of classes by grouping a set of Envy applications. A configuration map must include all classes that are required for the correct behaviour of a program. Therefore, the set of all classes in a configuration map can be used as the name space for the computation of acquaintance classes. As will be explained later, classifications play the role of the name space in reverse engineering with the Classification Browser (see Chapter 9).

7.10.6 Problems with Metalevel Code

The presented algorithm to compute acquaintance classes does not take metalevel code into account. Therefore, it may produce no results when applied to code that contains such code.

For example, consider the following Smalltalk classes, unrelated by inheritance: `ClassA` with a method `mA` and `ClassB` with a method `mB`. Further consider a class `ClassC` with a method `mC:`. The method expects an argument that is an instance of ClassA or ClassB, and that is implemented as:

```
mc: arg
  ^arg isKindOf: ClassA)
    ifTrue: [arg mA]
    ifFalse: [arg mb]
```

The presented algorithm would find `mA`, `mB` as the required interface for acquaintance `arg`. The problem is that no class conforms to that interface, and consequently the algorithm would not produce an acquaintance class. Solving this problem would require data-flow analysis to determine the class of the argument. The algorithm can, however, raise an exception to indicate that no class conforms to the required interface. In that case, the user has to determine the acquaintance class manually.
Note that the same problem arises when messages are found that do not exist anymore. The required interface is then too large to find a conforming class.

Similar problems occur when message selectors are passed as arguments to methods, or when message selectors are computed. Message selectors are typically passed or computed to use them as arguments of `perform:` messages. The algorithm to compute acquaintance classes does not consider `perform:` messages as special messages. The result is that a message sent by invoking `perform:` is not included in the required interface. Considering these messages would require data-flow analysis to determine what message is actually sent.

7.11 Summary

This chapter has shown how collaboration contracts can be recovered. The overall process consists of four stages. First, a classification of target classes is created. Second, the classes are stripped from unnecessary methods and unnecessary acquaintances to form a classification of participants. Third, the participants' acquaintance relationships are set up by determining the classes of the acquaintances. Fourth, the classification is turned into a collaboration contract by computing the specialisation clauses automatically based on method invocations found in the source code, and acquaintance relationships and methods classified in the participants.

The transitions between the stages require the reverse engineer to identify concerns, key classes and key collaborations. When documentation about the software is available, it may include pointers to interesting parts of the software, especially about concerns addressed by classes. The sad fact is that in many cases, no documentation is available. Then, it is up to the reverse engineer to find starting points in the software that may lead to interesting collaboration contracts. An elaborate list of clues and guidelines for identifying concerns, key classes and key collaborations have been discussed. Many of these guidelines are based on heuristics.

The recovery of collaboration contracts also has a technical aspect. Large parts of participants in collaboration contracts can be extracted from the source code. This quality is especially useful for determining the acquaintances of a participant and for recording the trace from implementation to design. A difficult problem in the recovery process is the mapping from acquaintance names to participants in a collaboration contract. A lightweight algorithm for computing acquaintance classes has been presented. It is based on matching the required interface of an acquaintance with the interfaces of the classes in the system.

This chapter has treated technical matters concerned with the recovery of collaboration contracts. The presentation of the 4-stage recovery process paves the way for the classification strategy that will be used to recover collaboration contracts with tools. The classification strategy, *classification with advanced navigation tools*, will be discussed in Chapter 9.

Chapter 8

Recovery of Reuse Contracts

A reuse contract describes how a collaboration contract is reused. Reverse engineering reuse contracts is the means to discover how reuse of collaboration contracts is achieved in the source code. This chapter proposes a 5-step approach to reverse engineering reuse contracts. After setting up an initial and a derived collaboration contract, the reuse contract between the two can be computed automatically.
The definitions of the concepts used in this chapter can be found in Appendix A.

8.1 How Does Source Code Map on Reuser Clauses?

Before introducing the 5-step approach to reverse engineering reuse contracts, it should be clear how concepts in the source code map on the concepts in a reuse contract, or in reuser clauses in particular. This knowledge is required for best understanding of the subsequent sections.

Reuse in an object-oriented program can be documented as follows.

Subclasses map to participants. Since classes map to participants, it is natural to map subclasses to participants too. However, the name of the participant corresponding to the subclass determines whether the participant is subject to a renaming or not. When the participant corresponding to the subclass does not have the same name as the participant corresponding to the superclass, a renaming of a participant is performed from the provider clause to the reuser clause. If the names are the same, no renaming is necessary.

Extra acquaintance classes are part of a context extension and refinement. When a subclass has more acquaintance classes than its superclass, the extra acquaintance classes map to participants in a context extension if the extra acquaintance classes were not part of the provider clause, and they also map to participants in an associated context refinement to set up the actual acquaintance relation.

Removed acquaintance classes are part of a context coarsening and cancellation. When a subclass has less acquaintance classes than its superclass, the dropped acquaintance classes map to participants in a context coarsening stating which acquaintance relationships are removed. These participants are also part of a context cancellation if they are not referenced anymore after the context coarsening.

A change in the abstractness attribute of a method is part of a participant abstraction or concretisation. When an abstract method in a superclass is overridden and implemented in a subclass, the overridden method is part of a participant abstraction reuser clause in which the method is listed in the interface of the participant that corresponds to the subclass. When it is the other way around, that is, when a concrete method in the superclass is made abstract in the subclass, the method is part of a participant concretisation reuser clause.

Abstracting a method in a subclass is a language-specific operation that can be expressed in Smalltalk, but not in C++ or Java. It is achieved through a Smalltalk coding convention: overriding a method with a `subclassResponsibility` message (usually to `self`) in its body. Although abstraction may seem a strange operation at first, it appears to be used often to introduce abstract layers in Smalltalk class hierarchies.

Extra methods in a subclass are part of a participant extension. When a subclass introduces extra methods with respect to the superclass, these methods are listed in the interface of the participant corresponding to the subclass in a participant extension reuser clause. The reuse contract model requires that a participant extension reuser clause does not reference any methods in a specialisation clause that are not part of the same extension. Consequently, a participant extension reuser clause must be complemented with a participant refinement reuser clause if any added method invokes any method that is already present in the provider clause.

Removed methods in a subclass are part of a participant cancellation. When a subclass removes methods that are present in the superclass, these methods are listed in the interface of the participant corresponding to the subclass in a participant cancellation reuser clause.
As method abstraction, method cancellation is an operation that can be expressed in Smalltalk, but not in C++ or Java. It is achieved through a Smalltalk coding convention: overriding a method with a `shouldNotImplement` message (usually to `self`) in its body. Method removal comes in handy when (intrinsic) multiple inheritance hierarchies are implemented in a single inheritance language.

Extra method invocations are part of a participant refinement. When a subclass overrides a method and invokes more methods than the overridden method, the extra method invocations are part of a participant refinement reuser clause in which the interface of a participant corresponding to the subclass lists the overridden method with the extra method invocations in its specialisation clause. Frequently, an overridden method removes some method invocations and introduces other method invocations. In that case the participant refinement reuser clause is accompanied by a participant coarsening reuser clause.

Removed method invocations are part of a participant coarsening. When a subclass overrides a method and invokes less methods than the overridden method, the removed method invocations are part of a participant coarsening reuser clause in which the interface of a participant corresponding to the subclass lists the overridden method and the removed method invocations.

Method invocations in a method performing a super send are part of a participant specialisation. When a method is overridden in a subclass, and the overriding method performs a super send, the other method invocations are part of a participant specialisation reuser clause in which the interface of a participant corresponding to the subclass lists the overridden method and the method invocations.

Other ways of reuse, possibly language-specific ways, cannot be captured by the current state of the reuse contract model. Table 8.1 summarises the mapping from reuse concepts onto reuse contract concepts.

Reuse Concept	**Reuse Contract Concept**
Subclass	Participant with same name as participant corresponding with superclass in provider clause Renaming
More acquainted classes	Context extension and refinement
Less acquainted classes	Context cancellation and coarsening
Methods become concrete in class	Participant concretisation
Methods become abstract in class	Participant abstraction
More methods in class	Participant extension
Less methods in class	Participant cancellation
More message sends in method body	Participant refinement
Less message sends in method body	Participant coarsening
Super send in method body	Participant specialisation
Other (language-specific) concepts	*not recorded*

Table 8.1: Mapping from reuse concepts onto reuse contract concepts

8.2 A 5-step Approach to Reverse Engineering

The approach proposed here consists of five steps. This section gives a short overview, while the five subsequent sections explain the steps in detail, with a simple example for illustration.

In the first step, an initial collaboration contract is set up by the reverse engineer, as discussed in the previous chapter.
In the second step, a derived collaboration contract is put together in which some classes mentioned in the initial collaboration contract are replaced by their subclasses, and in which collaborations described by the initial collaboration contract are adapted for those subclasses.
When the initial collaboration contract and the derived collaboration contract are determined, and the two are well-formed, the third step can be performed: the determination of the adaptation (the difference) between the initial collaboration contract and the derived collaboration contract. This adaptation is in fact a reuser clause that states how the initial collaboration contract is reused/changed by the derived collaboration contract. Since such extracted reuser clause is not defined in the reuse contract model (there exists no contract type for such a clause), it is decomposed into basic reuser clauses together with their contract types in the fourth step of the reverse engineering process.
After performing that step, all key ingredients are available to perform the fifth step: setting up a reuse contract with the provider clause (which is the initial collaboration contract), the extracted basic reuser clauses and their associated contract types. Supplying a name for the reuse contract completes the reverse engineering process.
This process can be repeated from the second step to reverse engineer other reuse contracts describing other reuses of the initial collaboration contract.
Reverse engineering reuse contracts is a semi-automatic process: the reverse engineer has to set up the initial collaboration contract and the derived collaboration contract, from which the reuse contract between the two can be determined automatically.

8.3 Step 1: Setting up the Initial Collaboration Contract

8.3.1 A Collaboration Contract for a Set of Classes

The previous chapter discussed how collaboration contracts can be recovered from the source code. The result of this step is a well-formed collaboration contract that describes a collaboration between a set of classes.

8.3.2 Example

An example will be used throughout this and the following three sections to illustrate the reverse engineering process.
Consider the `Collection` class in a Smalltalk system. `Collection` has a method `add:` to add a new object to a collection. Since the `Collection` class is merely the abstract root class of the `Collection` class hierarchy, it does not implement the method and thus declares it abstract. The interface of the `Collection` class can be extracted from the source code automatically. Since for this example we are only interested in adding behaviour of collections, we neglect all other methods in the interface of the `Collection` class. Since

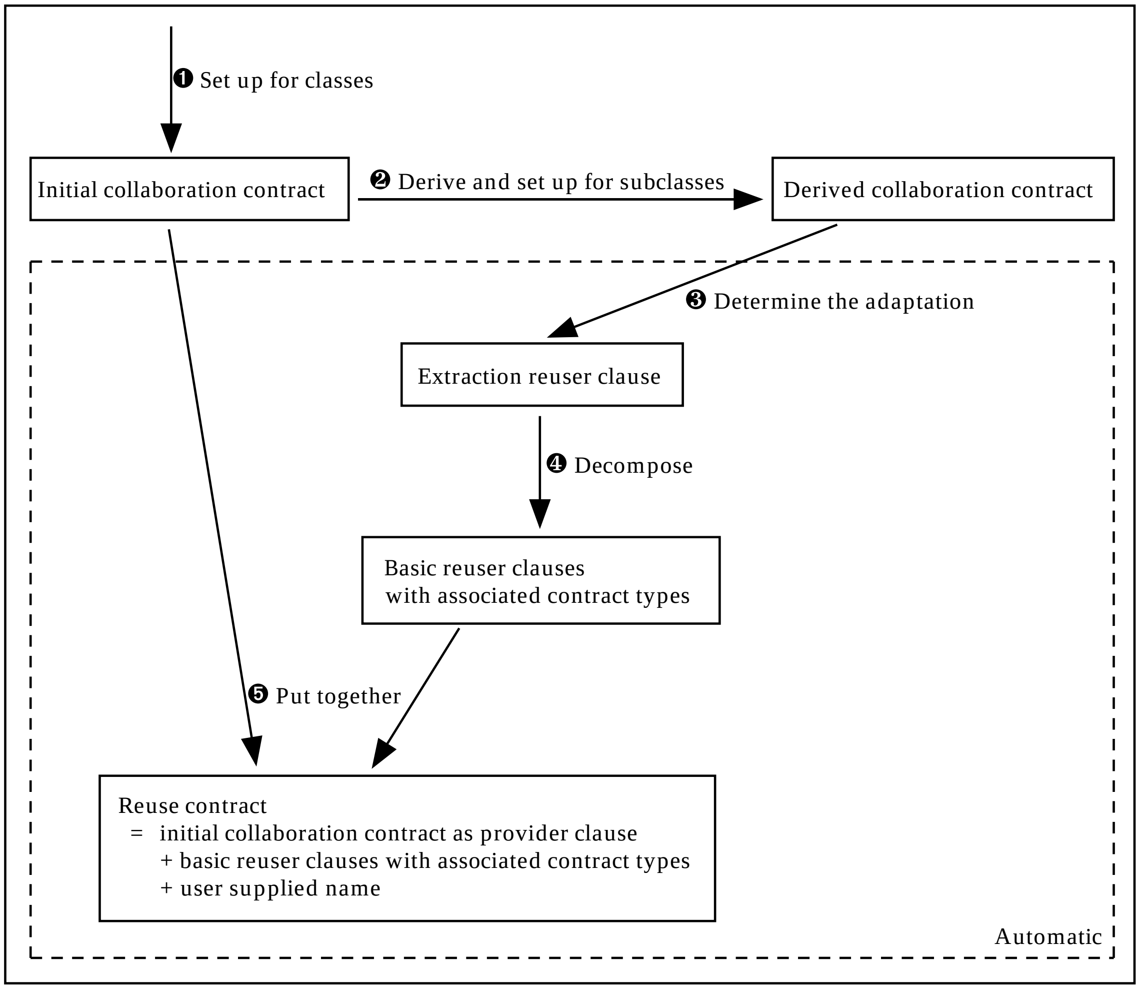

❶ Set up for classes
Initial collaboration contract
❷ Derive and set up for subclasses
Derived collaboration contract
❸ Determine the adaptation
Extraction reuser clause
❹ Decompose
Basic reuser clauses with associated contract types
❺ Put together
Reuse contract = initial collaboration contract as provider clause + basic reuser clauses with associated contract types + user supplied name
Automatic

Figure 8.1: 5-step approach to reverse engineering reuse contracts

a collection does not interact with the object argument of the `add:` method, no other participants are included in the collaboration contract under consideration. The resulting provider is depicted in Figure 8.2.

```
Participant Collection
      acquaintance clause:
            <empty>
      interface:
            abstract #add:
```

Figure 8.2: The initial collaboration contract describes part of the Collection class

8.4 Step 2: Setting up the Derived Collaboration Contract

A derived collaboration contract is derived from the initial collaboration contract by replacing some classes by their subclasses, and by adapting the collaboration contract to reflect how the subclasses work together.

8.4.1 A Collaboration Contract for a Set of Subclasses

Subclasses adapt the collaboration contracts in which their superclasses participate. They do so by overriding, adding and removing methods and by adding and removing acquaintance relationships that are part of the collaboration contracts they adapt. So when the reverse engineer wants to find how collaboration contracts defined for a set of classes are adapted for a set of subclasses of those classes, he should look for these changes.

8.4.2 Traceability of Initial Participants

It is common to use class names as participant names in a collaboration contract. When a collaboration contract for subclasses is set up, the participant corresponding to a subclass often gets a new name (the subclass' name). This results in a derived collaboration contract of which it is not known which participant is derived from which participant in the initial collaboration contract. In other words, there is no trace from the derived to the initial contract. To solve this problem, the original participant's name is recorded in the description of the derived participant. In the graphical notation, this backtrace information is written between parentheses after the name of the participant in the head of a participant box. The same notation is adopted for the textual form (see Figure 8.3).

In combination with the backtrace information on extracted classes (see Section 7.9.4), a participant named `xxx` that corresponds to a class chain `Bag..Collection` and is derived from a participant with name `yyy` is notated as `xxx:Bag..Collection (yyy)`.

8.4.3 Example

Class `Set` is a subclass of class `Collection` which expects some behaviour from the added objects. Since it uses a hash table to store the objects in a set, it requires that the added objects understand the message `hash` to provide a hash value that can be used as index in the hash table. Class `Set` introduces a method `findElementOrNil:` to find an index in the hash table where the added object should be stored. As shown in Figure 8.3, method `add:` sends `findElementOrNil:` to the receiver, and `findElementOrNil:` invokes `hash` on the object that is to be added. The notation `Set (Collection)` means that participant Collection is renamed to participant `Set` in this reuser[1]. Since participant `Set` does not require more than the method `hash` from the objects it stores, and because class `Object` provides this method for all objects in the Smalltalk system, a participant named `Object` is part of the reuser. `Object`'s interface only lists method `hash`, and its acquaintance clause is empty.

```
Participant Set (Collection)
      acquaintance clause:
            self -> Set
            anObject -> Object
      interface:
             #add: {self.#findElementOrNil:}
             #findElementOrNil: {anObject.#hash}

Participant Object
      acquaintance clause:
            <empty>
      interface:
             #hash
```

Figure 8.3: The derived collaboration contract describes part of the Set and Object classes

8.5 Step 3: Determination of the Extraction Reuser Clause

The third step in our approach is to compare the initial collaboration contract and the derived collaboration contract and to determine how they relate to each other. This is achieved by computing a reuser clause that represents the difference between the initial collaboration contract and the derived collaboration contract.

[1]The extractor tool used to provide this example gives the extracted participant the same name as the class from which it is extracted. It renames a participant when a subclass is extracted, but keeps the name of the original provider for reference.

The resulting reuser clause, called *extraction reuser clause*, is not a clause that is defined by the reuse contract model. It not only describes a static structure and an interaction structure, but also includes information on participants and methods that are not present in the derived collaboration contract anymore. While basic reuser clauses contain either positive information (i.e. additions), or either negative information (i.e. removals), an extraction reuser clause contains positive *and* negative information.
Apart from negative information about the removals, an extraction reuser clause also has to hold information about the participants that are renamed from an initial collaboration contract to a derived collaboration contract, since renamings cannot be represented by general reuser clauses. Computing an extraction reuser clause from a given initial collaboration contract and a given derived collaboration contract can be performed automatically by computing the difference between the two collaboration contracts with respect to the corresponding participants' interfaces, acquaintance clauses, specialisation clauses and abstractness attributes of methods.

8.6 Step 4: Decomposition of the Extraction Reuser Clause

An extraction reuser clause seldom maps directly onto one of the basic reuser clauses, since many ways of reusing a collaboration contract are coarser-grained than can be described by one basic reuser clause. Decomposition of an extraction reuser clause is therefore nearly always necessary to break it down into basic reuser clauses, that are at the heart of the reuse contract model. The decomposition is performed in this step: the extraction reuser clause is broken down into basic reuser clauses according to the definitions in Appendix A. According to those definitions, the following basic reuser clauses may be detected when CC is the collaboration contract and ERC is the extraction reuser clause:

- Renaming: ERC holds renaming information for a participant in CC.
- Context extension: ERC holds participants that are not present in CC.
- Context cancellation: ERC holds negative (i.e. removal) information on participants present in CC.
- Context refinement: ERC holds acquaintance relationships not present in CC.
- Context coarsening: ERC holds negative (i.e. removal) information on acquaintance relationships present in CC.
- Participant concretisation: ERC holds a concrete method on participant P that was abstract on P in CC.
- Participant abstraction: ERC holds an abstract method on participant P that was concrete on P in CC.
- Participant extension: ERC introduces methods on participants that are not present on those participants in CC.

- Participant cancellation: ERC holds negative (i.e. removal) information on methods in participants present in CC.
- Participant refinement: ERC holds the specialisation clause of method M in participant P that is a superset of the specialisation clause of method M in P in CC.
- Participant specialisation: ERC holds the specialisation clause of method M in participant P that holds all methods invoked besides the super invocation of M in participant P in CC.
- Participant coarsening: ERC holds the specialisation clause of method M in participant P that is a subset of the specialisation clause of method M in P in CC.
- Participant redefinition: ERC holds methods that are present in CC and cannot be categorised in one of the previous cases.

The decomposition can be fully automated, as explained in the following subsections.

8.6.1 Decomposition Issues

Decomposition of an extraction reuser clause into basic reuser clauses can be performed in several ways. As a degenerate case, the extraction reuser clause can be decomposed into one context cancellation to remove the participants in the initial collaboration contract, and one context extension to add all participants mentioned in the extraction reuser clause. Such decomposition would not only lead to bad reuse documentation, and therefore non-optimal description of how an initial collaboration contract is reused, but it would also give extremely bad results when the resulting reuse contracts are subject to impact analysis later on, because of the coarse-grained nature of the extracted reuser clauses. For most detailed reuse documentation and best impact analysis results later on, a fine-grained decomposition is the best option. A fine-grained decomposition is a decomposition in which the smallest changes are recorded in the most appropriate basic reuser clauses, instead of larger clauses.

Typically, several basic reuse clauses are decomposed from an extraction reuser clause. Since methods can be part of several reuser clauses, care must be taken that the order in which extraction of reuser clauses is performed does not inhibit extraction of other reuser clauses, so that obtaining correct and complete fine-grained decompositions is ensured. In general, care must be taken for all contract types that refer to the same part of a contract clause (participant, method, specialisation clause, acquaintance clause), because the extraction of these contract clauses may interfere with each other. The order of the extracted basic reuser clauses in the resulting combined reuser clause is also important to ensure the applicability of the reuser clauses. For instance, a participant refinement cannot precede an extension that introduces a new method that is refined or referred to in the refinement.

The order of decomposition used in all experiments carried out in this work is:

1. renaming

2. context extensions and context cancellations
3. context refinements and context coarsenings
4. participant concretisations and participant abstractions
5. participant extensions and participant cancellations
6. participant refinements, participant coarsenings, and participant specialisations
7. participant redefinitions

The decomposition is always a combined reuser clause with twelve[2] or less basic reuser clauses.

8.6.2 Renaming

Finding renamings is achieved by consulting the renaming information held by ERC. A renaming reuser clause holds all renamings, i.e. a mapping from participant names to participant names.

8.6.3 Context Extension and Context Cancellation

Finding context extensions is achieved by checking which participants are present in ERC, while not present in CC. All such participants are part of a context extension. Analogously, a context cancellation is found when some participants are part of CC, but are included in the negative information of ERC.

8.6.4 Context Refinement and Context Coarsening

A context refinement is found when the interaction stucture in ERC is a superset of the interaction structure in CC. The context refinement holds all added acquaintance relationships. Analogously, a context coarsening is found when the interaction structure is a subset of the interaction structure in CC. The context coarsening holds all removed acquaintance relationships.

8.6.5 Participant Concretisation and Participant Abstraction

Finding participant abstractions and concretisations is achieved by comparing the interfaces of a participant P in ERC and its interface in CC. When a method's abstractness attribute is present in CC, but not in ERC, the method is part of a participant concretisation. If it is the other way around, the method is part of a participant abstraction. In the two other cases, where the abstractness property has not changed during subclassing, the method is not affected by a participant abstraction or concretisation.

[2] 12 and not 13 because participant redefinitions are not kept for further processing, since the reuse contract model does not support it.

8.6.6 Participant Extension and Participant Cancellation

Finding participant extensions is achieved by comparing the interfaces and the specialisation clauses of corresponding participants P in CC and ERC. Comparing the specialisation clauses is necessary, because the comparison determines whether methods are only part of a participant extension, or also part of a subsequent participant refinement. The comparison is required to enforce self-containedness of the extension, as stated in the definition of the participant extension. When the specialisation clause of a method refers to a method that is not added to the interface of P in ERC, the method is part of the participant extension, but its specialisation clause is added through a subsequent participant refinement. When the specialisation clause of a method only refers to methods that are also part of the same extension, the method, together with its specialisation clause, is part of the participant extension.

Finding cancellations is achieved by comparing the interfaces of corresponding participants P in CC and ERC. When a method is part of the interface of P in CC, and part of the negative information in ERC, that method is included in a participant cancellation. Otherwise, the method is not affected by a participant cancellation.

8.6.7 Participant Refinement, Participant Specialisation, and Participant Coarsening

Finding participant refinements is achieved by comparing the specialisation clause of a method in participant P in CC and the specialisation clause of the same method in ERC. When the specialisation clause of a method in CC is a subset of the specialisation clause of the same method ERC, the method is part of a participant refinement that lists the method together with the extra invocations.

Finding participant specialisations is achieved by detecting a message send to `super` in the specialisation clause of a method in a participant in ERC. If a message send to `super` is found, the method with a specialisation clause listing the extra invocations is recorded in a participant specialisation.

Finding coarsenings is analogous to finding refinements. When the specialisation clause of a method in participant P in CC is a superset of the specialisation clause of the same method in ERC, the method is part of a participant coarsening that lists the method together with the invocations that were removed from the specialisation clause.

8.6.8 Participant Redefinition

Since contract types are only concerned with additions and removals to and from class interfaces and specialisation clauses, and with changes to the abstractness property of methods, it is possible that an extraction reuser clause cannot be completely covered by a decomposition in the basic reuse operators. After all, it is impossible to describe changes to methods that do not perform message sends, but perhaps perform more or less assignments to instance variables, or perhaps return other values than the overridden method.

Experience shows that often several methods in a class' interface are not found in the decomposed reuser clauses. Not displaying these methods in tools that list decomposed reuser clauses tends to confuse the user of those tools. For the sake of clarity, these methods should be displayed in tools in a separate semi-reuse clause.

Since such semi-reuser clause holds all overridden methods that cannot be captured by other contract types, it is called the *participant redefinition contract type*. Note that, in general, the participant redefinition reuser clause holds all methods unaffected by the reverse engineered reuser clauses. Currently, only eleven contract types are considered, but in the future more could be taken into account, such as contract types and clauses that record changes to methods with regard to state changes. Even then there will be methods that do not belong to any of the considered contract types. The participant redefinition contract type and reuser clause should thus be looked upon as a container for all the changes to methods that cannot be categorised into one of the contract types currently supported by the reuse contract model. Also note that the participant redefinition contract type is a concept that is only required during reverse engineering. Such a semi-reuse clause is not used during forward engineering of reuse contracts.

8.6.9 Example

Given the initial collaboration contract from Figure 8.2 and the reuser from Figure 8.3, the extraction reuser clause computed in step 2 is decomposed as shown in Figure 8.4.

The first reuser clause is a renaming which renames participant `Collection` to `Set`. The second is a context extension which adds participant `Object`, with only one method (`hash`). The next reuser clause is a context refinement which states that participant `Set` from now on knows participant `Object` through acquaintaince name `anObject` and itself through acquaintance name `self`. The fourth reuser clause is a participant concretisation, which states that participant `Set` concretises method `add:`. The fifth reuser clause, a participant extension, adds method `findElementOrNil:` to the interface of participant `Set`. Finally, a participant refinement describes how the two participants work together: from within method `add:` participant `Set` sends `findElementOrNil:` to itself, which in its turn invokes method `hash` on `Set`'s acquaintance named `anObject`.

8.7 Step 5: Setting up the Reuse Contract

Given the initial collaboration contract and the extracted combined reuser clause resulting from the previous step, all that needs to be done to set up a reuse contract is to provide a name for the reuse contract. This concludes the reverse engineering process.

Further reverse engineering based on the extracted reuse contract can be done when the extracted reuse contract is considered as a new collaboration contract.

Renaming

```
rename participant 'Collection' to 'Set'
```

Context extension

```
Participant Object
  acquaintance clause:
    <empty>
  interface:
    #hash
```

Context refinement

```
Participant Set
  acquaintance clause:
    self -> Set
    anObject -> Object
  interface:
    <empty>
```

Participant concretisation

```
Participant Set
  interface:
    #add:
```

Participant extension

```
Participant Set
  interface:
    #findElementOrNil:
```

Participant refinement

```
Participant Set
  interface:
    #add: {self.#findElementOrNil:}
    #findElementOrNil: {anObject.#hash}
```

Figure 8.4: Decomposition of the change between initial and derived collaboration contracts

8.8 Summary

This chapter has shown how reuse contracts can be reverse engineered. Reuse contract recovery is based on the recovery of an initial and a derived collaboration contract and the computation of a reuser clause that represents the adaptation from the initial to the derived collaboration contract.

The computation of the reuser clause can be automated. This quality is crucial for the integration of reuse contract recovery in a software development environment. Manually determining the reuser clause would be very labour-intensive.

Chapter 9

Classification with Advanced Navigation Tools

This chapter presents the software classification strategy *classification with advanced navigation tools*, as introduced in Section 6.3.3. It is a classification strategy to set up classifications of items based on relationships between the items. It is a manual classification strategy supported by advanced navigation tools. In this chapter, *classification with advanced navigation tools* will be restricted to the interaction structure of objects. The relationships between the items are thus method invocation dependencies.

First, this chapter presents the Classification Browser. The browser is not only the main tool to create, manipulate, and browse classifications through manual and virtual classification strategies; it is also an advanced browser to browse interaction structures. Second, the strategy *classification with advanced navigation tools* is applied in the recovery of collaboration contracts. Third, an experiment of collaboration contract recovery is discussed.

9.1 The Classification Browser

The software classification model is the foundation of the Classification Browser, a browser that integrates Smalltalk browsing and editing facilities with support for the creation, manipulation, exploration and exploitation of classifications. The Classification Browser has been developed based on the browser framework delivered with ApplFLab [SHDM95], [SHDB96], [Hon98]. The browser is able to handle Smalltalk categories and Envy applications, the primary means of structuring classes in Smalltalk and Envy/Developer, and has a familiar user interface, so that daily Smalltalk and Envy/Developer users are comfortable with it.

9.1.1 A View on the Classification Repository

A Classification Browser is a view on the classification repository. Figure 9.1 shows how all open Classification Browsers share the same repository.

The classification repository is the model (in MVC terminology) of a browser. It handles all changes to the classifications (by Classification Browsers or other agents that manipulate the repository) and reports all changes to the open Classification Browsers. The latter keep themselves consistent with the repository. The simultaneous update of all browsers avoids inconsistencies across browsers. The classifications in the Classification Browsers are thus always up-to-date.

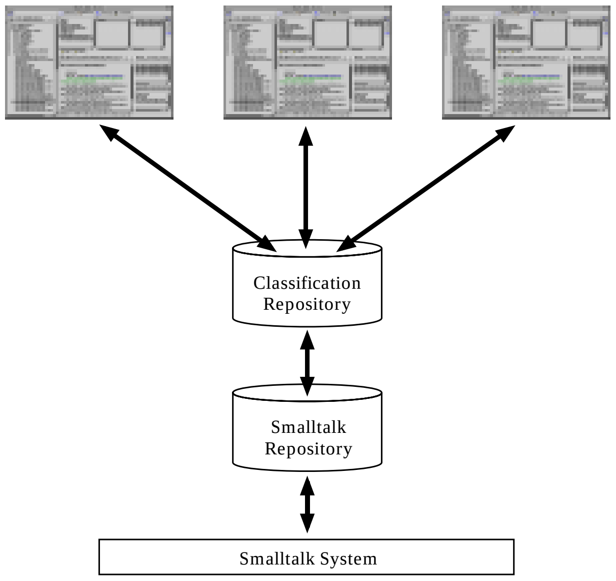


Figure 9.1: A Classification Browser is a view on the classification repository

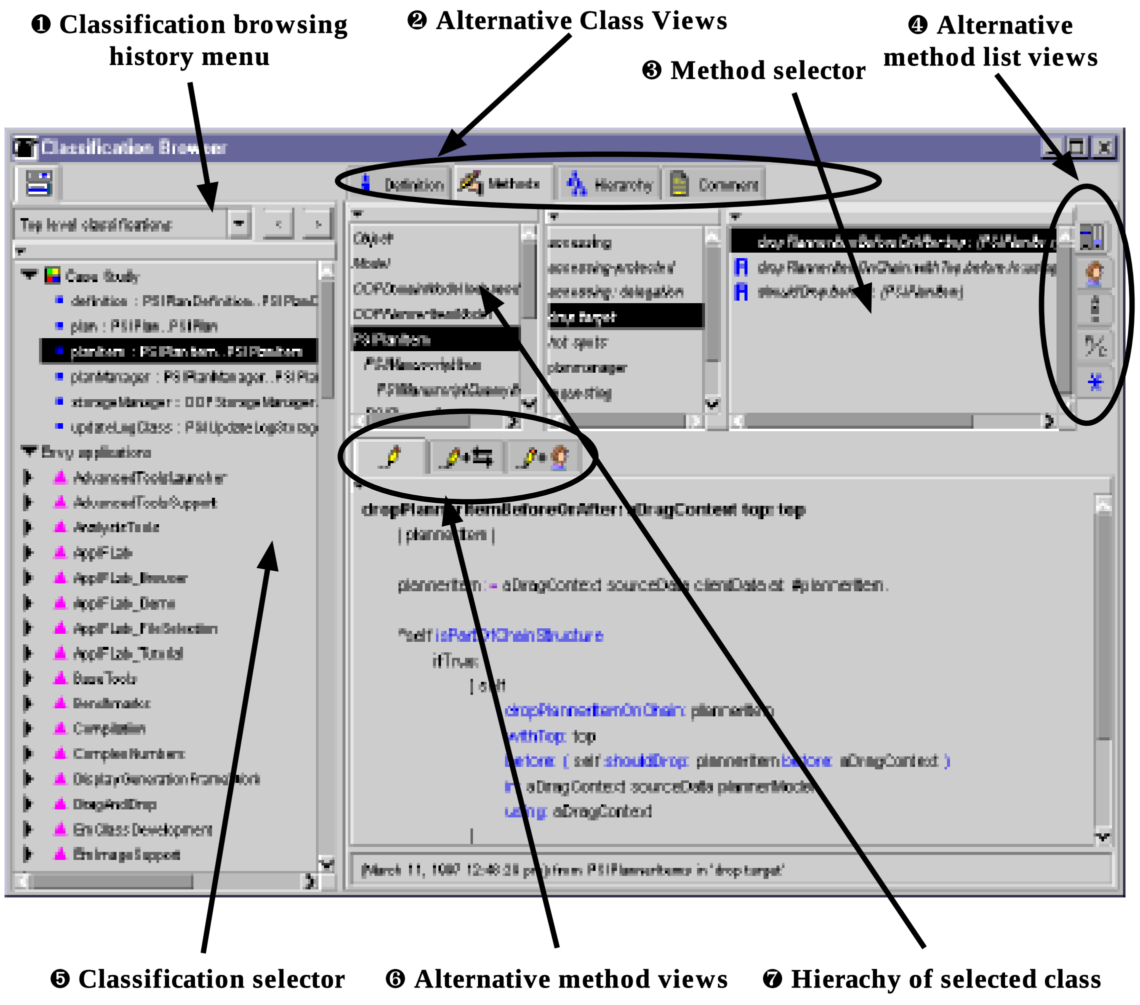
A Classification Browser only interacts with the classification repository. If Smalltalk-related information is necessary, the classification repository interacts with the so-called Smalltalk repository, actually a façade of the Smalltalk system.

9.1.2 Brief Overview of the Functionality

The Classification Browser looks like a standard Smalltalk class browser, but in fact enhances the standard browser in many ways. Besides the ability to work with classifications, it supplies the software engineer with many alternate views and a multitude of additional functionality. Only a general overview of the Classification Browser is given here. Subsequent sections elaborate on some of the functionality to support software classification strategies.

Figure 9.2 shows the Classification Browser. Only part of the browser is visible because many parts of the browser's interface are on other pages in the several notebooks (multi-page widgets with tabs at the top or on the right) that together form the general layout of the browser window. The browser consists of two major parts: a classification selector on the left-hand side (5 in the figure) and an editor on the right-hand side.

The classification selector lists the classifications and their items in a hierarchical manner. Each kind of classification or item has its own icon[1]. Classes have a small red dot as icon; participants have a small blue dot as icon (for instance 'definition' in the figure). Classifications have a four-colour square as icon (for instance 'Case Study' in the figure). Envy applications carry a pink triangular icon (for instance 'AdvancedToolsLauncher' in the figure). The black triangular icon in front of a classification can be toggled to expand and collapse that classification. The figure shows two expanded classifications: 'Case Study' and 'Envy Applications'.



```
dropPlannerItemBeforeOrAfter: aDragContext top: top
    | planneritem |

    planneritem := aDragContext sourceData clientData at: #planneritem.

    ^self isPartOfChainStructure
        ifTrue:
            [ self
                dropPlannerItemOnChain: planneritem
                withTop: top
                before: ( self shouldDrop: planneritem before: aDragContext )
                in: aDragContext sourceData plannerModel
                using: aDragContext
            ]
```

(March 11, 1997 12:48:28 pm) from PSIPlanneritems in 'drop target'

Figure 9.2: Classification Browser

[1]The icons currently in use are just simple visual clues. Not much time has been devoted yet to select and design icons that intuitively reflect the kind of classification or item.

The classification selector provides menu commands for creating, classifying, moving, and removing items and classifications. Classifications can be imported from and exported to external media. The history menu (1 in the figure) displays the name of the current classification, that is, the classification of which the items are currently displayed in the classification selector. When the software engineer (the user) zooms into a classification, the context where he started from is logged in the history, so that he has a trace of his navigation activities. The history menu (and the buttons next to it) can be used to go back and forward in the history[2].

The editor is a part of the browser with many faces. In the figure, four alternative views (2 in the figure) are available. These views correspond to the four views provided by a standard Smalltalk browser: the definition of a class, the methods of a class, the hierarchy in which the class resides, and the on-line documentation (comment) about the class. A Classification Browser may have more views on classes or classifications. Each extra view is another page in the notebook.

The 'Methods' page in the editor's notebook (as shown in the figure) provides the software engineer with an interface to add, modify, remove and browse methods of classes selected in the classification selector. It looks like a standard Smalltalk browser, but incorporates many new features.

The hierarchy of the method's containing class is displayed in the hierarchy list (7 in the figure). In general, only the class selected in the classification selector is selected in the hierarchy list. However, the software engineer can select more than one class in the hierarchy list. The method selector on the hierarchy list's right side will display the methods of all classes selected in the hierarchy list. This feature provides a way to browse overridden methods in a class hierarchy[3].

Methods are selected with a method selector (3 in the figure). A frequently used method selector is the combination of a protocol and a method list, but the browser provides alternative views (4 in the figure), such as an acquaintance view (with an icon that looks like a head) to browse methods that refer the selected acquaintance (explained later), an alphabetic view ('ABC'), a view that splits up abstract and concrete methods ('A/C'), and a pattern match view ('*'). Each alternative view corresponds to another way of searching for a method. Each alternative view has at least a method list. The method list displays abstract methods in a special way (with a preceding icon 'A') so that they are spotted easily.

While the classification selector provides menu commands for classifying classes in classifications, the class hierarchy list provides menu commands to classify classes as participants in classifications. When a class is classified as participant, the newly created participant

[2]The history is similar to the history facilities of world-wide-web browsers.

[3]This functionality is also known as *full browser* functionality in VisualWorks\Smalltalk [PD95].

item does not include methods. The protocol list and the method list provide menu commands to classify methods in participants[4].

When a participant is selected, the protocol list and the method list display the methods that are classified in the participant, as well as the methods that are not classified. The former are displayed in plain text, the latter are displayed in italic text. Thus, the user sees the participants as views on classes. The methods in plain text are in the view (classified), the methods in italic text are outside the view (unclassified).

Information on the selected method is displayed in a page of the notebook at the bottom of the window. The notebook provides alternative views on the method (6 in the figure). The first page in the notebook displays a simple text editor to edit the body of a method. The other views on a method will be explained in subsequent sections.

9.1.3 Default Top-Level Classifications

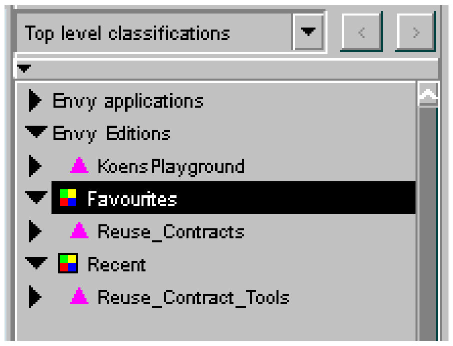


Figure 9.3: Default top-level classifications (some expanded)

When a Classification Browser is opened, it displays at least the following top-level classifications that serve as starting points for browsing.

Envy Applications. (Envy/Developer only) This virtual classification holds all loaded Envy applications in the Smalltalk image. Each application is a virtual classification of Class Items (see Section 6.2.3) and Envy Applications (see Section 6.2.9).

Envy Editions. (Envy/Developer only) This virtual classification holds all Envy applications that are open for edition. Each application edition is a virtual classification of Class Items and Envy Applications.

[4]Note the use of 'classification of methods in a participant', although participants are not defined as classifications, but as items.

Smalltalk categories. (VisualWorks only). This virtual classification holds all categories in the Smalltalk image. Each category is a virtual classification of Class Items.

Recent. This special classification is used to keep a history of recently used classifications. The 'Recent' classification is filled automatically when the software engineer creates classifications and classifies items. The number of included classifications is limited by the browser; the software engineer has no control over this classification.

Favourites. This special classification holds frequently used classifications. The software engineer decides when and which classifications should be added or removed. Since it is a classification, addition and removal of favourite classifications is performed through classify and unclassify commands. The number of favourite classifications is unlimited. The classification is under the software engineer's control.

9.1.4 Browsing the Interaction Structure

The Classification Browser provides the ability to browse senders and implementers in-place, that is, in the same browser window. In order to be backward compatible with the standard browser, the standard way of browsing senders and implementers is still supported by the Classification Browser.

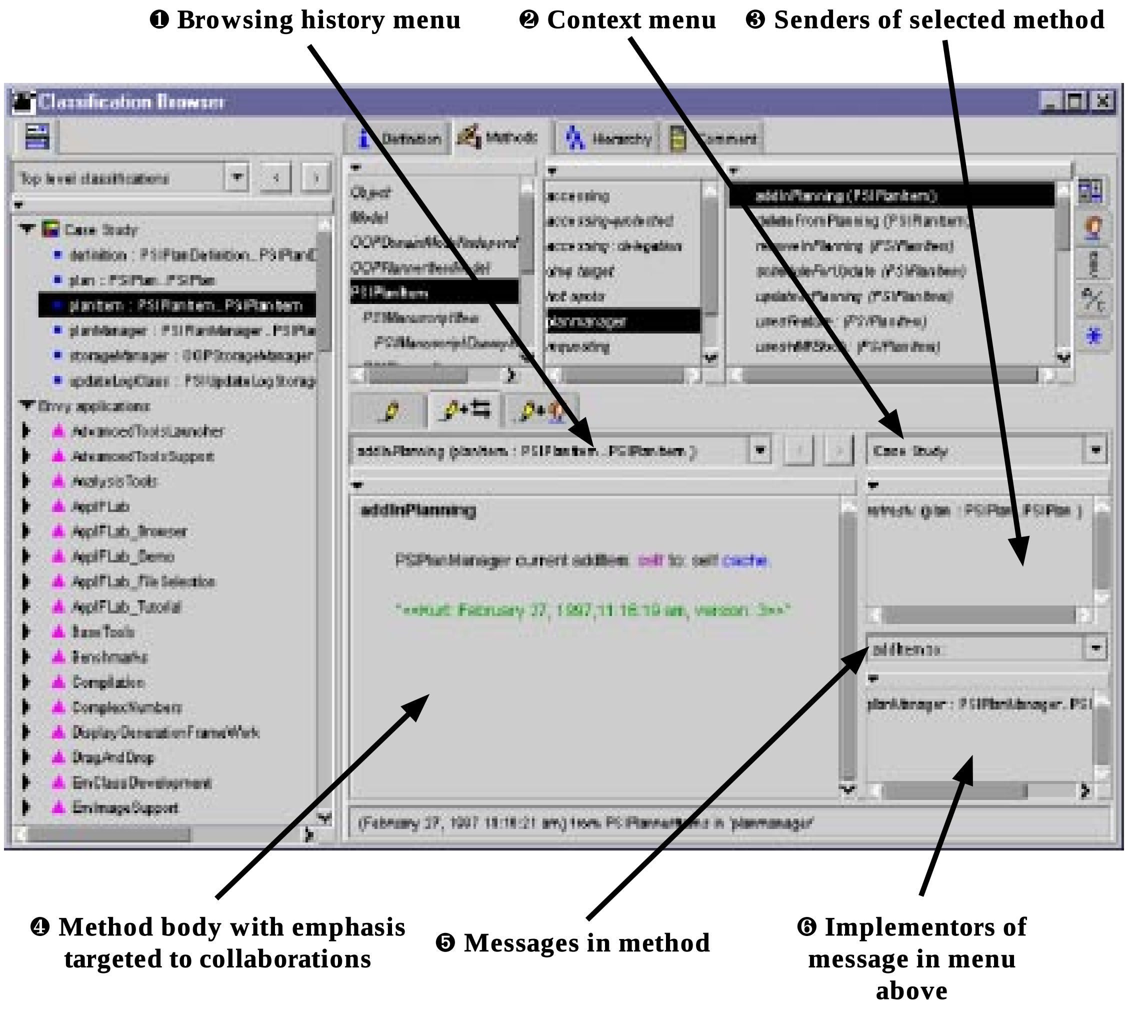
With standard senders/implementers browsers, the methods that result from a senders /implementers browsing action are shown in isolation. The Classification Browser displays the resulting methods in the context of their class. Browsing senders and implementers (sometimes called 'lateral browsing' or 'horizontal browsing') is thus tightly integrated with browsing class hierarchies (sometimes called 'vertical browsing').

The Classification Browser keeps a history of places that are visited when senders and implementers are browsed, so that the software engineer has a trace of his browsing actions and is able to backtrack at any time.

Figure 9.4 shows the Classification Browser in action. The point of interest here is the notebook in the bottom-right corner of the browser window. The text field (4 in the figure) displays the body of the selected method with special emphasis on grammatical entities that makes reading and browsing the source code easier. Self sends, super sends, the receiver ('self') as argument of messages, assignments and comments are displayed in different colours, so that they are spotted easily. Besides the body of the selected method, the notebook displays additional information that relates to senders and implementers, the context (classification) in which they are computed, and the browsing history.

The Classification Browser has a *scope reduction* feature. Browsing senders and implementers always happens in the context of a classification. The selection in the context menu (2 in the figure) defines the scope in which browsing takes place. The context menu holds the Favourites and the Recent classifications. Since these classifications are listed in the classification selector on the left-hand side of the browser window, the software engineer can easily manipulate the list of classifications shown in the context menu. He typically adds the classifications he is working with to the Favourites classification, so that he can select his own classifications as the browsing scope. The context menu always includes a special classification named 'All Classes'. When this special classification is selected, browsing happens Smalltalk-system-wide, that is, all classes in the Smalltalk image are queried to compute senders and implementers. A special classification named 'None' can be selected to turn off computation of senders and implementers.

This scope reduction feature is very helpful for narrowing the scope in which software is browsed. It is a practical way to focus on the classes that are under consideration. Irrelevant classes for the development task at hand can be kept out of view.


❶ Browsing history menu
❷ Context menu
❸ Senders of selected method
Classification Browser
Top level classifications
Definition
Methods
Hierarchy
Comment
Case Study
❹ Method body with emphasis targeted to collaborations
❺ Messages in method
❻ Implementors of message in menu above

```
addInPlanning

    PSIPlanManager current addItem: self to: self cache.

    "<<<kurt: February 27, 1997,11:16:10 am, version: 3>>"
```

Figure 9.4: Senders and implementers facilities in the Classification Browser

The senders list (3 in the figure) updates itself each time another method or another

context is selected. The implementers list (6 in the figure) updates itself when a message in the messages menu (5 in the figure) is selected. The implementers are thus not the implementers of the selected method, but the implementers of a message sent by the selected method, referred to by Smalltalk developers as the *message implementers*.
Double clicking an item in the senders or implementers list makes the browser change its focus to the method indicated by that item. The browser saves the old focus of attention in a history. The history is kept in a history menu (1 in the figure) that can be used by the software engineer to go back and forward in the history of browsing actions as desired.

9.1.5 Browsing Acquaintance Relationships

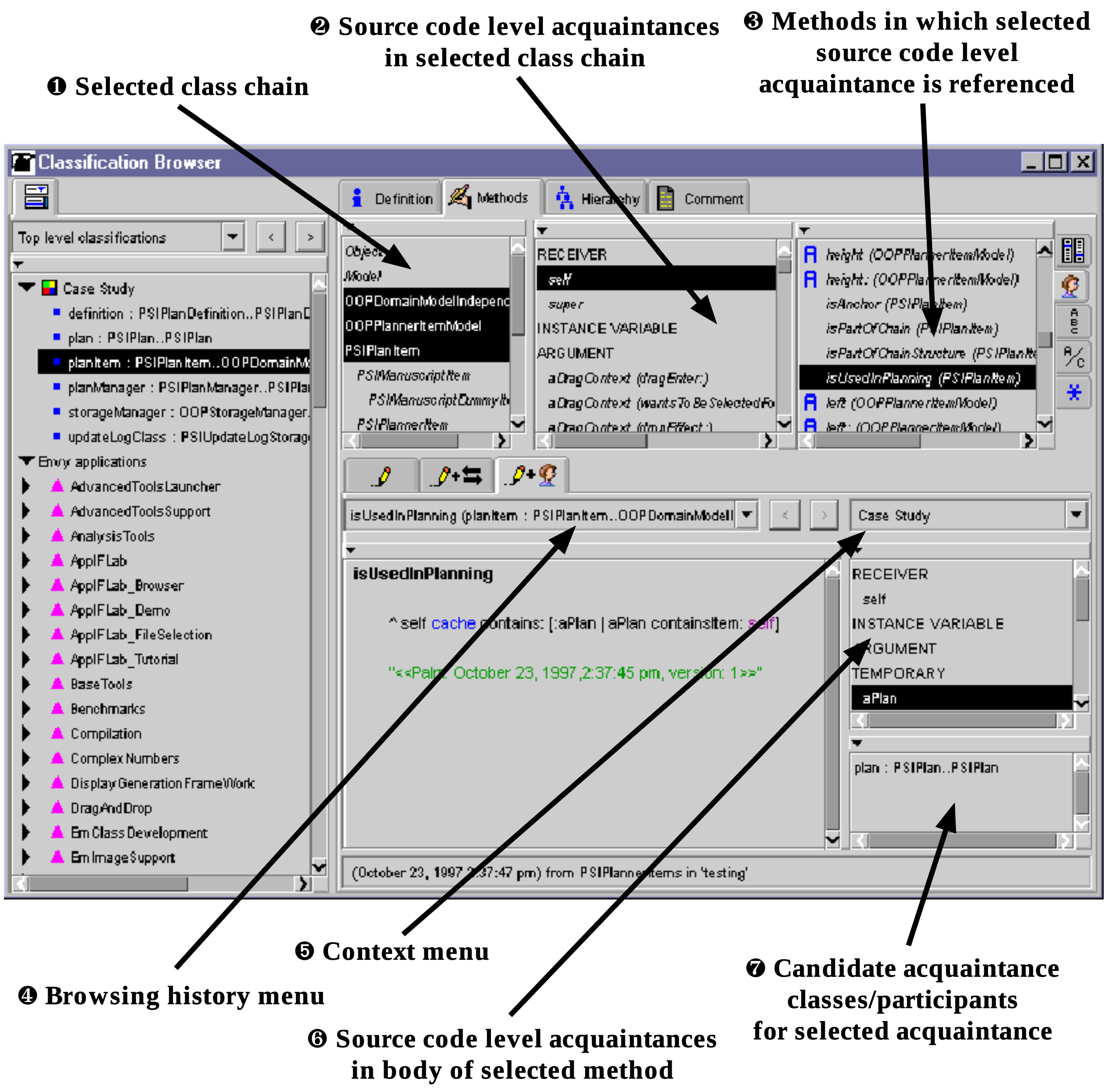
Besides integrated browsing of senders and implementers, the Classification Browser supports browsing based on the acquaintances found in the source code. On top of that, it is able to determine candidate acquaintance classes for source code level acquaintances.

Figure 9.5 shows the Classification Browser's facilities for browsing source code level acquaintances. These facilities are available when the acquaintance-related notebook pages (with icons that look like heads) are in view. The method selector notebook in the top-right corner of the browser window includes a page with an acquaintance and a method list. The method body editor in the lower half of the browser window includes a page with an acquaintance list and an acquaintance class list.
The browser displays the source code level acquaintances in the acquaintance list (2 in the figure). The list contains all source code level acquaintances found in the flattened class that corresponds with the selected class chain in the class hierarchy list (1 in the figure). The list displays acquaintance names. These names have a form as discussed in Section 7.8.4 and given in Table 7.3 (page 101), except for source code level acquaintances with method scope. Instead of prefixing their names with the signature of their containing method, the name is displayed with the method signature between parentheses behind it.

When a source code level acquaintance is selected, the method list (3 in the figure) shows all methods in which the selected acquaintance is referenced. For source code level acquaintances with method scope, only one method will be shown.

When a method is selected, the method body editor displays the method body as usual. In addition, it shows the source code level acquaintances found in the method body in the acquaintance list (6 in the figure).

A selection in that acquaintance list results in the computation of the candidate acquaintance classes for the selected source code level acquaintance. Computing the candidate acquaintance classes is done as explained in Section 7.10. The computation is restricted to the classification selected in the context menu (5 in the figure). This restriction corresponds to the introduction of a name space to enhance the search of conforming classes, as discussed in Section 7.10.5.
The results of the computation are shown in the list (7 in the figure) in the bottom-right


❶ Selected class chain
❷ Source code level acquaintances in selected class chain
❸ Methods in which selected source code level acquaintance is referenced
❹ Browsing history menu
❺ Context menu
❻ Source code level acquaintances in body of selected method
❼ Candidate acquaintance classes/participants for selected acquaintance
Classification Browser
Definition
Methods
Hierarchy
Comment
Top level classifications
Case Study
Envy applications
RECEIVER
INSTANCE VARIABLE
ARGUMENT
TEMPORARY
Case Study

```
isUsedInPlanning

    ^ self cache contains: [:aPlan | aPlan containsItem: self]

    "<<Palm: October 23, 1997 2:37:45 pm, version: 1>>"
```

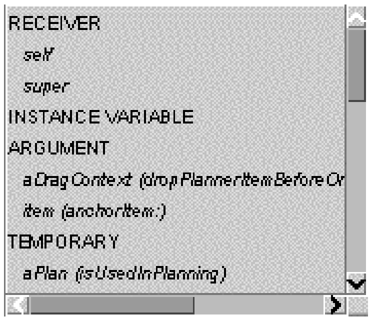
(October 23, 1997 2:37:47 pm) from PSIPlannerItems in 'testing'

Figure 9.5: Acquaintance facilities in the Classification Browser

corner of the browser window. The list may remain empty, indicating that no candidate acquaintance classes can be found in the selected context classification. Choosing a broader context may produce better results.

Double clicking a candidate acquaintance class makes the browser focus on that class. The old focus is saved in the history (4 in the figure), in the same way message implementers are browsed (see Section 9.1.4).

Note that the Classification Browser displays all source code level acquaintances, including the possibly uninteresting ones for inclusion in collaboration contracts, as explained in Section 7.8.3. This way, the software engineer gets the whole picture. However, the browser displays the source code level acquaintances ordered by implementation stereotype (see Figure 50), as found in Definition 2 and Definition 3 (see Section 7.8.1). Uninteresting acquaintances are thus spotted easily.



```
RECEIVER
  self
  super
INSTANCE VARIABLE
ARGUMENT
  aDragContext (dropPlannerItemBeforeOr
  item (anchorItem:)
TEMPORARY
  aPlan (isUsedInPlanning)
```

Figure 9.6: Source code level acquaintances ordered by stereotype in the browser

The ability to browse methods based on acquaintances found in their body opens the door to a wealth of information that is hard to obtain with standard browsers. The acquaintance list in the method selector shows all objects that receive a message. By examining this list, the software engineer gains an understanding of which acquaintances are used in the selected class chain. He immediately knows whether self sends are performed, whether the superclass is reused through super sends, whether instance variables are accessed directly, and whether classes, pool dictionaries, class variables and global variables are used. Moreover, he can quickly assess whether the Law of Demeter is respected, because intermediate acquaintances, not found when the Law is respected, show up in the list as well.

The set of methods in which an acquaintance is referenced is just one click away. This means that the software engineer has instant access to information about the self-reliance of the target class chain (methods performing self sends), the dependency on the superclass (the methods performing super sends), and the like. This quality is not only useful for reverse engineering collaboration contracts, it is useful for browsing in general.

9.2 Application in Architectural Recovery

9.2.1 Recovery of Collaboration Contracts

The Classification Browser supports the incremental approach to reverse engineering based on classifications, as presented in Section 7.1. According to the stated procedure, the reverse engineer determines the participants and classifies methods and acquaintances in those participants. Editing specialisation clauses is not necessary.
The following steps should be taken to reverse engineer collaboration contracts with the Classification Browser. The transitions of incremental classification discussed in Section 7.1 are indicated for reference.

1. **Prepare a global browsing context.** The purpose of this step is to set up the context (a set of classes and/or other classifiable items) that defines the global scope in which browsing will take place. This browser context can be selected in the Classification Browser's context menu to restrict the scope in which browsing senders and implementers, and computation of acquaintance classes will take place.

   A new classification is created and all classes under consideration are classified in there. It is not necessary to classify all classes one by one. If other classifications hold the desired classes (such as Smalltalk categories or Envy applications), these classifications can be classified in the new classification. All containing classes are then part of the global browsing context. If the browsing context appears to be too small or too large later, it can be changed easily to fit new needs.

   For example, an installation of the broadcast management software for a television or radio station consists of three Envy applications. The first holds the classes of the general application framework, the second contains the whats'On framework classes and the third contains the station-specific classes. When these applications are loaded into the Smalltalk image, they show up as classifications in the Classification Browser. When a classification `All Whats'On classes` is created and when the three Envy applications are classified in that classification, the global browsing context for whats'On has been prepared. Figure 9.7 shows part of the Classification Browser with a classification called `Whats'On` that contains the global browsing context with the three Envy applications. The pink triangular icon in front of the name of a classification indicates that it is an Envy application.

2. **Identify interesting subcontexts.** The purpose of this step is to create classifications that hold parts of the software that are interesting to consider as a whole. Examples are classifications that correspond to different views on the software, and classifications that hold important class hierarchies. This step is not required, but the resulting classifications often come in handy later during browsing. Interesting subcontexts may emerge during browsing, which means that this step can be taken at any time.

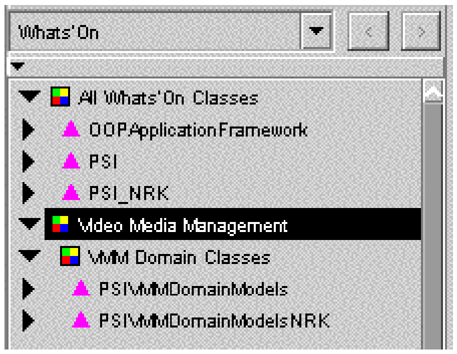


Figure 9.7: Context classifications created during incremental recovery

For example, when a software engineer searches for collaboration contracts in the Video Media Management domain classes, the creation of a classification `VMM domain classes` is useful, because it delimits the classes of interest. Figure 9.7 shows the interesting subcontext for the recovery of a collaboration contract in the Video Media Management Module. It is nested in a classification `Video Media Management`.

3. **Browse the software and search for key classes and key collaborations.** The purpose of this step is to identify classes for which collaboration contracts should be set up.

   The clues, guidelines and heuristics discussed in Sections 7.5, 7.6 and 7.7 are used to identify key classes and key collaborations. The Classification Browser works to the user's advantage in that it indicates some clues in the user interface so that they are spotted easily. For instance, the abstractness attribute of a method is displayed in the method list. Method bodies are enhanced with colours to indicate important grammatical clues, such as self sends, super sends, receiver arguments, and source comments. The hierarchy in which a class resides is displayed in the user interface, so that an assessment of the number of subclasses can be made quickly. Methods of subclasses and superclasses can be displayed in the method list by selecting the desired classes in the class hierarchy list.

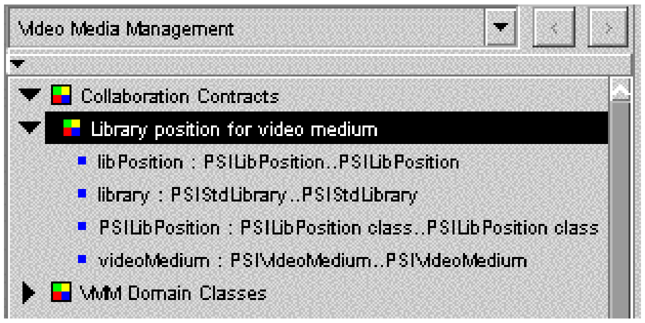
   The integrated senders, implementers, and acquaintances facilities help to track collaborations between classes. The history facilities can be used for backtracking if necessary.

   For example, using the provided guidelines, four classes are identified that collaborate to find a free library position for a video medium: `PSILibPosition`, `PSILibPosition class` (the metaclass of the first class), `PSIStdLibrary`, and `PSIVideoMedium`.

4. **Create a classification for a target collaboration contract (transition to**

stage 1). This step is the first step towards recording participants in a collaboration contract. The new classification serves as a container to hold participants that correspond to key classes found in the software. Later, this classification will be turned into a real collaboration contract. This step requires the reverse engineer to choose a name for the classification. A good name for the target collaboration contract may not be available at this time, but that does not pose a problem. The name of the classification can be changed at any time.

In the example, a subclassification `Library position for video medium` is created in the `Video Media Management` classification. Figure 9.8 shows this new classification.



Video Media Management

Collaboration Contracts
Library position for video medium
libPosition : PSILibPosition..PSILibPosition
library : PSIStdLibrary..PSIStdLibrary
PSILibPosition : PSILibPosition class..PSILibPosition class
videoMedium : PSIVideoMedium..PSIVideoMedium
VMM Domain Classes

Figure 9.8: Classes as participants in the target classification

5. **Classify key classes as participants in the target classification (transition 1 – 2 of Section 7.4).**

   If a class has been identified as a key class for the desired collaboration contract, it can be recorded as a participant by classifying it in the target classification. As already explained earlier, identifying key classes and identifying key collaborations is usually performed together, so this step and the next one are often done alternately.

   When the key classes are identified and when they are classified as participants in the target classification, it is a good idea to set the browsing context to the target classification. Doing so helps the identification of key collaborations, since senders, implementers, and acquaintance classes are only determined in that small context.

   In the example, the classes identified in step 3 are added as participants to the classification `Library position for video medium`. Using the browsing facilities, the appropriate class chain is selected for each participant. All participants are assigned a name. Figure 9.8 shows that the class chains of all participants are restricted to the subclasses themselves (`PSIStdLibrary..PSIStdLibrary` for instance).

6. **Classify key methods (transition 1 – 2).** The purpose of this step is to reduce the interfaces of the participants in the target classification to the methods that are actually important for the target classification.

   The important methods are the methods that are invoked in key collaborations. These methods are found by browsing senders, implementers, and acquaintance classes, and by applying the guidelines and heuristics.

   Classification of methods is performed at method level or at protocol level. The protocol list and the method list supply menu commands to classify and unclassify protocols and methods. The Classification Browser uses different font styles to display classified and unclassified methods, so that they are distinguished easily.

```
PSILibPosition class.. PSILibPosition class
    newInLibrary:videoMedium:
PSILibPosition..PSILibPosition
    canBeUsedBy:
    code:
    free
    free:
    library:
    vmType
    vmType:
PSIVideoMedium..PSIVideoMedium
    nr
    vmType
PSIStdLibrary.. PSIStdLibrary
    createNewLibPositionForVideoMedium:
    firstFreeLibPositionForVideoMedium:
    getLibPositionForVideoMedium:
    libPositionForVideoMedium:throwException:
    nextLibPositionCode
    newLibPositionForVideoMedium:
```

Figure 9.9: Classified methods in the recovered participants

   In the example, the method `libPositionForVideoMedium:throwException:` is recognised as the method that initiates the interaction. It is classified in the participant `PSIStdLibrary.. PSIStdLibrary`. Other methods are classified in the corresponding participants. This results in participants with interfaces as shown in Figure 9.9.

7. **Classify key acquaintances (transition 2 – 3).** Besides determining the interfaces of the participants by classifying methods, the reverse engineer needs to

determine the acquaintance relationships between the participants by classifying acquaintances and associating acquaintance names with participants in the classification.

The lightweight algorithm to compute the acquaintance classes (see Section 7.10) is used by the browser to determine the acquaintance participants (not classes) in the context of the target classification. The browser provides menu commands to set up the actual acquaintance relationship. The software engineer explicitly associates an acquaintance name with a candidate acquaintance participant.

In the example, browsing and determining the acquaintance relations produces the results depicted in Figure 9.10. The format of the acquaintance relationships is `(<stereotype>) <acquaintance name>` → `<participant name>`.

```
libPosition : PSILibPosition class.. PSILibPosition class
    (argument) aLibrary → library
    (argument) aVideoMedium → videoMedium
    (temporary) pos → libPosition
PSILibPosition  : PSILibPosition..PSILibPosition
    (argument) aVideoMedium → videoMedium
    (receiver) self → PSILibPosition
videoMedium : PSIVideoMedium..PSIVideoMedium
    <no acquaintance relations>
library : PSIStdLibrary.. PSIStdLibrary
    (tempoarary) position → libPosition
    (navigation) PSILibPosition.siteClass → PSILibPosition
    (receiver) self → library
```

Figure 9.10: Classified acquaintances in the recovered participants

8. **Repeat from step 2 until a collaboration contract emerges.** Steps 2 through 7 are repeated until the classification holds participants of which the acquaintances are also in the classification.

   The emerging collaboration contract will be tangible in the Classification Browser. Browsing the method invocations with senders and implementers in the context of the classification results in participants of the classification only. The browser then acts as a collaboration contract browser whereby the method invocations between the participants of the target collaboration contract can be browsed.

9. **Convert the classification into a collaboration contract (transition 3 – 4).**

> This step completes the incremental recovery of a collaboration contract.
>
> The Classification Browser provides a menu command to convert a selected classification into a collaboration contract. The browser computes the specialisation interfaces of all methods in the participants in the target classification. It also checks the well-formedness of the resulting collaboration contract. If the well-formedness check produces negative results, the participants in the classification need further examination. Otherwise, the classification represents a well-formed collaboration contract.

As mentioned in Section 7.4, the incremental recovery process cannot be strictly divided into the steps given above. Classification of methods and acquaintances, and setting up acquaintance relationships may be done at any time. The well-formedness check at the end ensures that no methods and acquaintance relationships are forgotten.

9.2.2 Interoperability with UML

A picture says more than a thousand words. A diagram is easier to read than a textual representation of a collaboration contract. Therefore, the Classification Browser is able to translate collaboration contracts to scripts that can be read and executed by a UML diagramming tool[5].

In UML, the collaboration contract is modelled by a class diagram and a collaboration diagram [MLS98]. The former models the static structure: the participants, their interfaces and the acquaintance relationships. The latter models the interaction structure: the participants and the messages they send among them.
The UML class diagram that is generated for the collaboration contract recovered in the previous section is depicted in Figure 9.11. The UML collaboration diagram is shown in Figure 9.12. The generated script is too large to show here. It is included in Appendix C.

The class names in the UML class diagram are the names of the partially flattened classes for which the collaboration contract was recovered. The acquaintance names are used as role names in the class diagram. Note that `PSILibPosition.siteCLass` is not a usual role name. It corresponds with an acquaintance relationship established through a message `siteClass` to class `PSILibPosition`.

9.3 Application in Software Evolution

9.3.1 Recovery of Reuse Contracts

As presented in Chapter 8, the recovery of reuse contracts is a 5-step process. The first step is the recovery of an initial collaboration contract. The second step is the recovery of

[5]Rational Rose [Cor96].

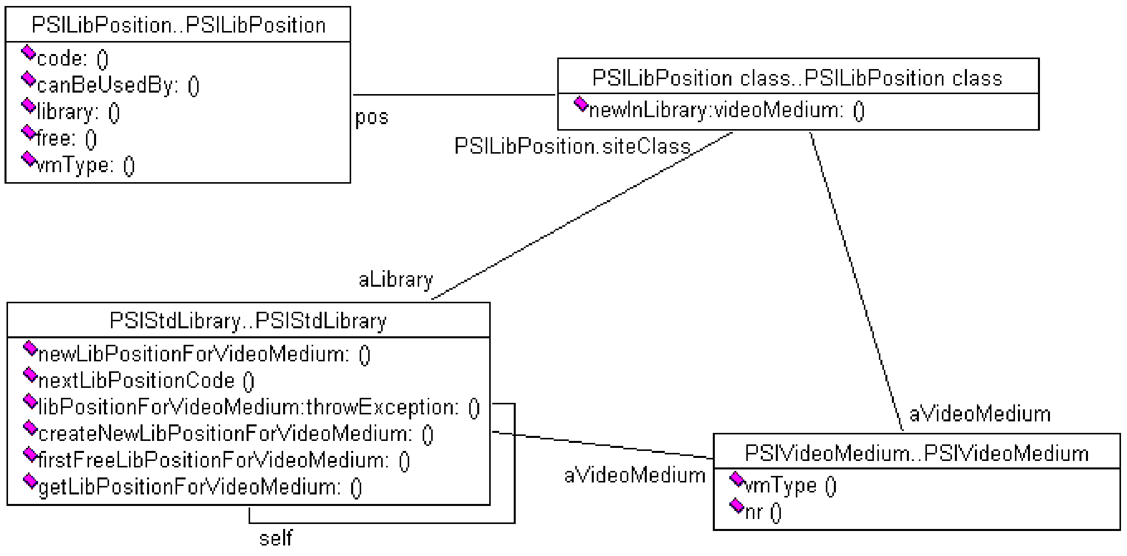

PSILibPosition..PSILibPosition
code: ()
canBeUsedBy: ()
library: ()
free: ()
vmType: ()
pos
PSILibPosition class..PSILibPosition class
newInLibrary:videoMedium: ()
PSILibPosition.siteClass
aLibrary
PSIStdLibrary..PSIStdLibrary
newLibPositionForVideoMedium: ()
nextLibPositionCode ()
libPositionForVideoMedium:throwException: ()
createNewLibPositionForVideoMedium: ()
firstFreeLibPositionForVideoMedium: ()
getLibPositionForVideoMedium: ()
self
aVideoMedium
aVideoMedium
PSIVideoMedium..PSIVideoMedium
vmType ()
nr ()

Figure 9.11: UML class diagram generated from a collaboration contract

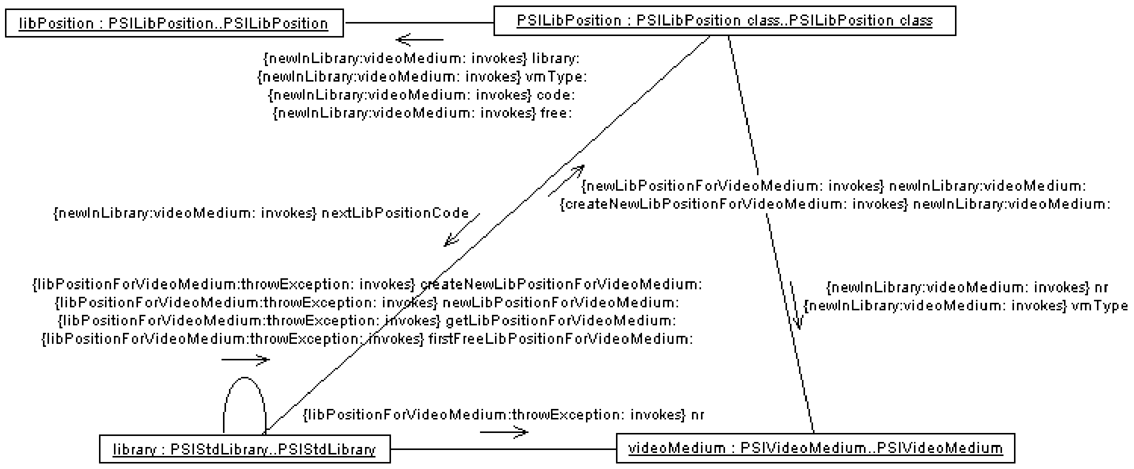

libPosition : PSILibPosition..PSILibPosition
PSILibPosition : PSILibPosition class..PSILibPosition class
{newInLibrary:videoMedium: invokes} library:
{newInLibrary:videoMedium: invokes} vmType:
{newInLibrary:videoMedium: invokes} code:
{newInLibrary:videoMedium: invokes} free:
{newLibPositionForVideoMedium: invokes} newInLibrary:videoMedium:
{createNewLibPositionForVideoMedium: invokes} newInLibrary:videoMedium:
{newInLibrary:videoMedium: invokes} nextLibPositionCode
{libPositionForVideoMedium:throwException: invokes} createNewLibPositionForVideoMedium:
{libPositionForVideoMedium:throwException: invokes} newLibPositionForVideoMedium:
{libPositionForVideoMedium:throwException: invokes} getLibPositionForVideoMedium:
{libPositionForVideoMedium:throwException: invokes} firstFreeLibPositionForVideoMedium:
{newInLibrary:videoMedium: invokes} nr
{newInLibrary:videoMedium: invokes} vmType
{libPositionForVideoMedium:throwException: invokes} nr
library : PSIStdLibrary..PSIStdLibrary
videoMedium : PSIVideoMedium..PSIVideoMedium

Figure 9.12: UML collaboration diagram generated from a collaboration contract

a derived collaboration contract. The three other steps determine the reuser clauses and the associated contract types of the reuse contract that describes the evolution from the initial to the derived collaboration contract. These steps can be performed automatically.

The recovery of reuse contracts thus relies on the recovery of collaboration contracts, as described in the previous section. Therefore, recovery of reuse contracts can be considered an indirect application of the software classification strategy presented in this chapter.

9.3.2 Keeping Collaboration Contracts in Sync

When collaboration contracts have been recovered, they can be invalidated when the source code is changed. In the current state of the Classification Browser, the inconsistency between the collaboration contract (the model) and the source code is not reflected in the browser, but to some extent the inconsistencies can be reflected.

Several changes affect the consistency of a collaboration contract. Method invocations can be added to and removed from method bodies. This change affects the specialisation clauses of methods. Source code level acquaintances can be added to and removed from method bodies. This change affects the acquaintance clauses of participants. Methods can be added to and removed from classes. This change affects the interfaces of participants. Classes can be added to and removed from the software system. This affects the collaboration contract as a whole.

Since collaboration contracts record what is in the source code, it is possible to check whether the entities in a collaboration contract are still present in the source code after a change has been made. When the entity is not there anymore, the Classification Browser can display that entity in an eye-catching way to indicate that that part of the collaboration contract is invalid. This works fine for removals, but it does not work well for additions. Since collaboration contracts (or better participants) are views on the source code, they do not include all what is found in method bodies. The Classification Browser cannot decide automatically whether an added entity in the source code should be included in the collaboration contracts. Therefore, it cannot indicate whether an addition makes a collaboration contract inconsistent with the source code. However, if it is assumed that all development tools use the (Smalltalk) repository (see Figure 9.1), additions of methods and method invocations to classes that participate in collaboration contracts can be signalled.

When the Classification Browser indicates invalidation in a collaboration contract after removals, or when additions are signalled, the software engineer does not have to start from scratch to update the affected collaboration contract to the new version of the source code. He can start from the collaboration contract. The update is restricted to the affected parts of the collaboration contracts; the other parts can be maintained.

This property is a consequence of the incremental approach to recovery of collaboration contracts. Consider a collaboration contract at stage 4 (see Figure 7.1). Additions and removals of method invocations bring parts of the affected collaboration contract to stage 3. Additions and removals of source code level acquaintances bring parts of the affected contract to stage 2, as do additions and removals of methods. Additions and removals of classes bring parts of the affected collaboration contract to stage 1. In order to get the affected collaboration contract back to stage 4, the recovery for the update of the contract starts at the level to which the contract fell back.

9.4 Experiment: Recovery of Design Documentation

Parts of the broadcast management system are documented with class diagrams and textual explanations. In this experiment, one of those class diagrams is subjected to careful examination by browsing the corresponding source code with the Classification Browser. The goal is to assess how complete and how correct the design documentation is, compared to what can be found in the source code, and to set up design documentation in the form of a collaboration contract.

The report given here is based on notes taken during the recovery process.

9.4.1 The Provided Design Documentation

The provided design documentation consisted of several class diagrams. Four class diagrams only included inheritance relationships, one class diagram described dependency relationships between classes, and one class diagram described association relationships between classes. The latter class diagram is shown in Figure 9.13. The others are of less importance for this experiment and were not used. In addition to the design diagrams, a written document "The internal workings of the Plan Manager" gave a loose description of how the so-called *plan manager* responds to a request for a plan according to a given *plan definition*.

The informal textual documentation mentioned that the method `getPlanFor:withRefresh:kind:withProgressBlock:` is a very important method in class `PSIPlanManager`. The sole instance of `PSIPlanManager` is responsible for the management of `PSIPlan` instances, which are windows on the global planning of a television or radio station. The method retrieves a `PSIPlan` that conforms to a `PSIPlanDefinition`, passed as first argument. A `PSIPlanDefinition` defines the conditions under which a `PSIPlanItem` belongs to a `PSIPlan`. It defines a time window on the global planning of a station. For example, when a plan is opened for a television station for a period from June 1 until June 5, and from 8 am until 11 am, the plan definition defines that period. Programmes planned in the global planning of a station are included in the plan only if they lie inside the time window defined by the plan definition.

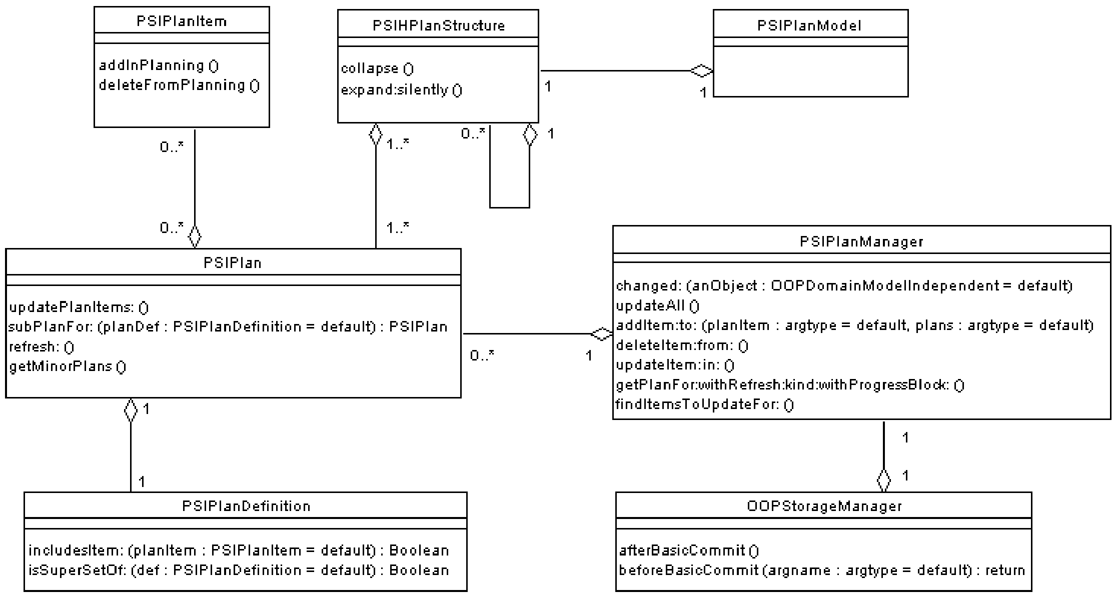

PSIPlanItem
addInPlanning ()
deleteFromPlanning ()
PSIHPlanStructure
collapse ()
expand:silently ()
PSIPlanModel
0..*
0..*
1..*
1..*
0..*
1
1
1
PSIPlan
updatePlanItems: ()
subPlanFor: (planDef : PSIPlanDefinition = default) : PSIPlan
refresh: ()
getMinorPlans ()
PSIPlanManager
changed: (anObject : OOPDomainModelIndependent = default)
updateAll ()
addItem:to: (planItem : argtype = default, plans : argtype = default)
deleteItem:from: ()
updateItem:in: ()
getPlanFor:withRefresh:kind:withProgressBlock: ()
findItemsToUpdateFor: ()
0..*
1
1
1
1
1
PSIPlanDefinition
includesItem: (planItem : PSIPlanItem = default) : Boolean
isSuperSetOf: (def : PSIPlanDefinition = default) : Boolean
OOPStorageManager
afterBasicCommit ()
beforeBasicCommit (argname : argtype = default) : return

Figure 9.13: The provided class diagram

9.4.2 Preparing the Browser Context

The global scope for this experiment are all whats'On classes. A classification called `All Whats'On Classes` was created to hold these classes (see Figure 9.7 on page 136). This context classification was used when narrower browsing scopes were not yet created or when browsing in narrower scopes produced no results.

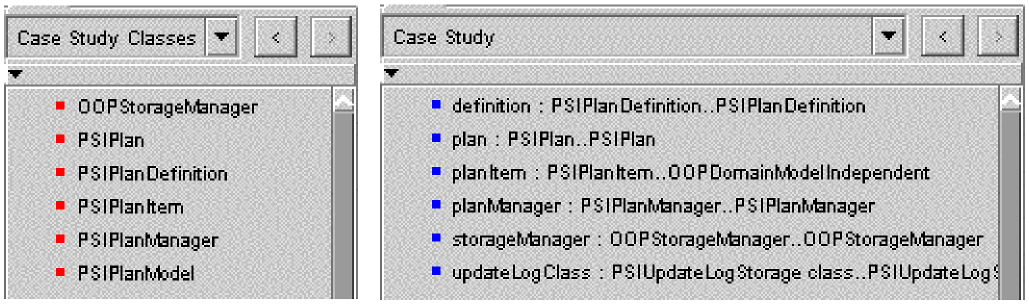
9.4.3 Identifying Concerns

The provided textual documentation was consulted to determine the concern that must be considered. The concern could be read from the documentation: *retrieving a plan according to a given plan definition.*

9.4.4 Identifying Key Classes

The identification of key classes was based on the provided design documentation. All classes in the provided class diagram were looked up. Browsing the classes in the diagram with the Classification Browser immediately revealed that class `PSIHPlanStructure` did not exist. This also invalidated the two expected associations with other classes (`PSIPlanModel` and `PSIPlan`).

A classification 'Case Study Classes' was created and the existing classes were classified in there. This classification defines the scope in which the experiment took place. A second classification 'Case Study' was created and the classes from the first classification were classified in there *as participants*. The purpose of this classification was to gradually evolve into a collaboration contract. The participants were renamed (participants carry the name of the corresponding class by default). Initially, the participants did not have classified methods. Figure 9.14 shows the two classifications.



- OOPStorageManager
- PSIPlan
- PSIPlanDefinition
- PSIPlanItem
- PSIPlanManager
- PSIPlanModel

- definition : PSIPlanDefinition..PSIPlanDefinition
- plan : PSIPlan..PSIPlan
- planItem : PSIPlanItem..OOPDomainModelIndependent
- planManager : PSIPlanManager..PSIPlanManager
- storageManager : OOPStorageManager..OOPStorageManager
- updateLogClass : PSIUpdateLogStorage class..PSIUpdateLog

Figure 9.14: Two classifications, one with classes, one with participants

The alphabetic method view of the browser was used to spot methods stated in the class diagram rapidly. They were classified in the corresponding participants. This browsing and classification step revealed that method `getMinorPlans` on class `PSIPlan` did not exist, and that method `beforeBasicCommit` took no arguments, contrary to what the class diagram stated. Browsing also showed that the methods `includesItem:` and `isSuperSetOf:` on class `PSIPlanDefinition` were abstract methods. Moreover, it showed that all classes except `PSIPlanManager` were abstract.

Figure 9.15 shows the method view of the Classification Browser when participant `plan` is selected (the selection is not shown). The list on the left is the hierarchy of the corresponding flattened class (in this case just `PSIPlan`). The list in the middle shows the protocols, and the list on the right shows the methods. Methods displayed in plain text are classified methods; methods in italic are not classified. Protocols in italic do not include classified methods, while protocols in plain text do.
This figure clearly shows that a participant is a view on a (flattened) class. The elements in italic are out of view.

9.4.5 Identifying Key Collaborations

Browsing senders and implementers with the Classification Browser in the context of the classes pointed out that the collaboration between the different classes was complex. Many methods were involved in retrieving a plan for a given plan definition. Moreover, some methods were very long (`refresh:` and `getPlanFor:withRefresh:kind:withProgress-Block:` for instance).

While browsing senders and implementers in the context of the classification `Case Study`

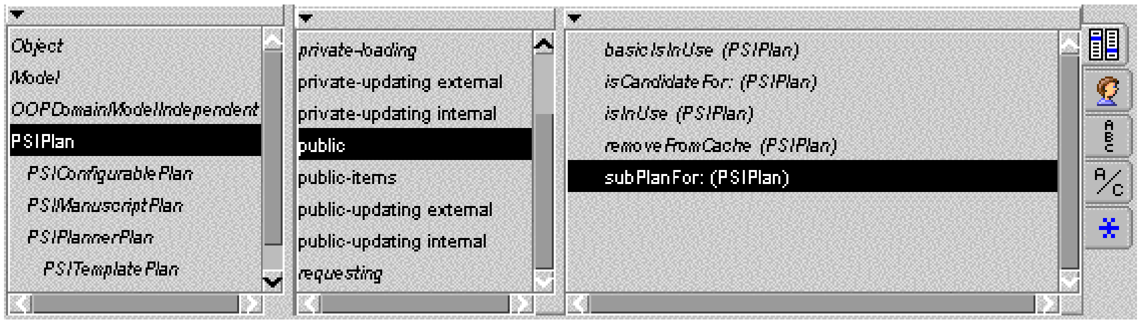


Figure 9.15: Classified and unclassified methods

`Classes`, interaction patterns started to emerge. Browsing was guided by three guidelines from Chapter 7. Spotting abstract methods to find template methods via senders resulted in two abstract methods and three template methods. Spotting message sends that take the receiver of the sending method as argument resulted in three methods.

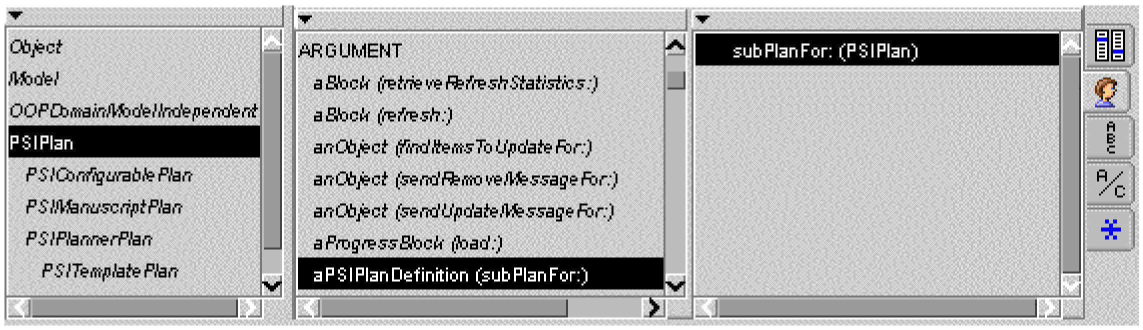
Using the browsing facilities for acquaintance relationships, five methods were found where the acquaintance `PSIPlanManager current` was used. Since `PSIPlanManager` plays an important role in the examined source code, these methods were examined by browsing senders and implementers.

9.4.6 Identifying Acquaintance Relationships

Classification of methods was alternated with classification of acquaintance relationships. When a message send was considered important for the collaboration contract, the acquaintance relationship corresponding to the receiving object and the method corresponding to the message were classified.

Browsing and classifying acquaintance relationships indicated that some aggregation relationships in the class diagram were wrong. It also became clear that the class `PSIPlanModel` did not play a role in the collaboration, but another class did: `PSIUpdateLogStorage class` (a metaclass). This class was added to the classification `Case Study Classes` and it was classified as participant in classification `Case Study`.

Figure 9.16 shows the acquaintance view of the Classification Browser when the participant `plan` is selected (this selection is not shown). The list in the middle displays all acquaintance relationships ordered by implementation stereotype (only arguments are in view). The list on the right displays the methods in which the selected acquaintance relationship is realised (only one method, since the acquaintance object is a method argument). Classified acquaintance relationships are displayed in plain text. Unclassified acquaintance relationships are displayed in italic. Again, this figure shows that a participant is a view on a (flattened) class.



Object
Model
OOPDomainModelIndependent
PSIPlan
PSIConfigurablePlan
PSIManuscriptPlan
PSIPlannerPlan
PSITemplatePlan

ARGUMENT
aBlock (retrieveRefreshStatistics:)
aBlock (refresh:)
anObject (findItemsToUpdateFor:)
anObject (sendRemoveMessageFor:)
anObject (sendUpdateMessageFor:)
aProgressBlock (load:)
aPSIPlanDefinition (subPlanFor:)

subPlanFor: (PSIPlan)

Figure 9.16: Classified and unclassified acquaintances names

9.4.7 Determining Acquainted Participants

When acquaintance relationships were classified, the acquainted participants were determined immediately. In the context of the `Case Study` classification, for each acquaintance relationship a single candidate acquainted participants was found. That was no surprise, regarding the small context in which they were computed.

Figure 9.17 shows part of the Classification Browser used to browse acquaintance relationships in a method body and to classify acquainted participants. The figure shows the acquaintance relationships in the method `subPlanFor:` on participant `plan:PSIPlan..PSIPlan`. All acquaintance relationships in view are displayed in plain text, which means that they are classified in the participant. Italic display is used to indicate that an acquaintance relationship is not classified. For the selected acquaintance relationship, identified by the acquaintance name `aPSIPlanDefinition`, one candidate acquainted participant is found (displayed at the bottom): `definition:PSIPlanDefinition..PSIPlanDefinition`. The browser displays it in plain text, indicating that the candidate acquainted participant has been classified. This means that the containing participant plan refers to participant `definition` with acquaintance name `aPSIPlanDefinition`.

9.4.8 Setting up the Collaboration Contract

When all methods involved in retrieving the plan were classified and when all involved acquaintance relationships were classified, the classification was turned into a collaboration contract. The Classification Browser computed the specialisation clauses of the methods, and performed a well-formedness check to see whether the collaboration contract was consistent. The consistency check turned up some problems that were fixed by classifying extra acquaintance classes.

The resulting collaboration contract is depicted in Figure 9.18 (static diagram) and Figure 9.19 (interaction diagram). Note the acquaintance relationship mentioned in the previous section. `PSIPlan..PSIPlan` refers to `PSIPlanDefinition..PSIPlanDefinition` with

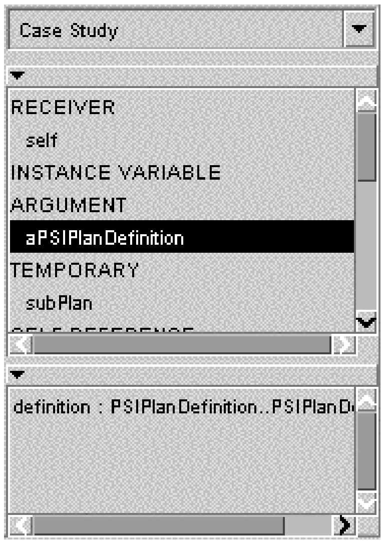


Figure 9.17: Determining acquainted participants

`aPSIPlanDefinition`.

9.4.9 Results

The resulting collaboration contract shows how a plan for a given plan definition is retrieved. The collaboration contract is rather large. It is probably a good idea to split it up along the different concerns addressed: determining a subplan, refreshing a plan according to the update log, and notifying dependents.

The comparison of the provided design documentation and the recovered collaboration contract shows that the class diagram was incomplete and inaccurate. Although some inconsistencies with the source code were expected, the result deviates considerably from the expected architecture. First, one class in the class diagram was not found in the software (`PSIHPlanStructure`). Another one does not take part in the examined collaboration. One undocumented (meta)class, unexpected to collaborate, appeared to play a role after all (`PSIUpdateLogStorage class`). Second, ignoring the non-existing and non-collaborating classes, the class diagram describes four class dependencies (associations). For the recovered collaboration contract, already ten inter-class dependencies are found. The recovery of more collaboration contracts for these classes would produce even more dependencies. Third, the textual documentation loosely describes how the classes collaborate. The collaboration contract documents a lot more details.

On the other hand, specifics about acquaintance relationships would enhance the collaboration contract considerably. Multiplicity and aggregation would improve the understanding of the (shared) aggregation relation between `PSIPlanManager` and `PSIPlan`.

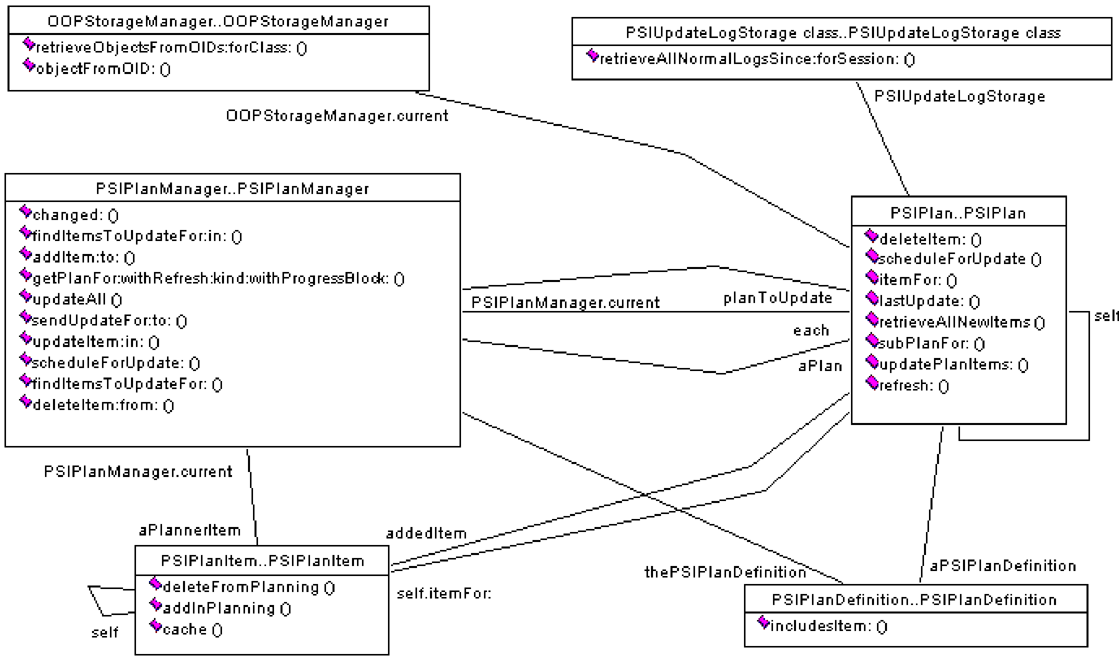


Figure 9.18: Static structure of the recovered collaboration contract

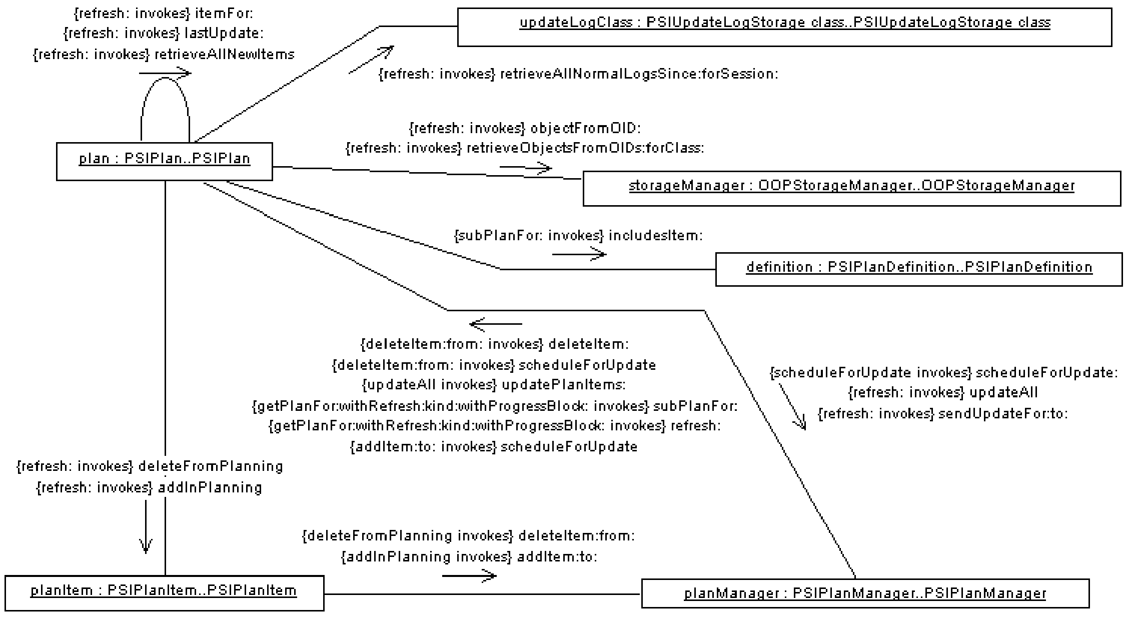

{refresh: invokes} itemFor:
{refresh: invokes} lastUpdate:
{refresh: invokes} retrieveAllNewItems
updateLogClass : PSIUpdateLogStorage class..PSIUpdateLogStorage class
{refresh: invokes} retrieveAllNormalLogsSince:forSession:
{refresh: invokes} objectFromOID:
{refresh: invokes} retrieveObjectsFromOIDs:forClass:
plan : PSIPlan..PSIPlan
storageManager : OOPStorageManager..OOPStorageManager
{subPlanFor: invokes} includesItem:
definition : PSIPlanDefinition..PSIPlanDefinition
{deleteItem:from: invokes} deleteItem:
{deleteItem:from: invokes} scheduleForUpdate
{updateAll invokes} updatePlanItems:
{getPlanFor:withRefresh:kind:withProgressBlock: invokes} subPlanFor:
{getPlanFor:withRefresh:kind:withProgressBlock: invokes} refresh:
{addItem:to: invokes} scheduleForUpdate
{scheduleForUpdate invokes} scheduleForUpdate:
{refresh: invokes} updateAll
{refresh: invokes} sendUpdateFor:to:
{refresh: invokes} deleteFromPlanning
{refresh: invokes} addInPlanning
{deleteFromPlanning invokes} deleteItem:from:
{addInPlanning invokes} addItem:to:
planItem : PSIPlanItem..PSIPlanItem
planManager : PSIPlanManager..PSIPlanManager

Figure 9.19: Interaction structure of the recovered collaboration contract

This experiment shows that the recovery of collaboration contracts can be useful to iterate over a *reflexion model* (also see Section 12.5.2). A reflexion model is an expected model of the source code to reflect on. The initial reflexion model used here is the provided class diagram (and the additional textual documentation). Through classification of participants, methods, and acquaintance relationships, this model was refined until the resulting collaboration contract emerged.

9.5 Summary

This chapter has presented *classification with advanced navigation tools*, a software classification strategy to recover classifications of software entities with an interaction structure. The proposed classification strategy is based on the availability of advanced tools to browse the interaction structures in a software system.

The advanced navigation tool presented and used here is the Classification Browser. The Classification Browser also supports two other software classification strategies: manual classification and virtual classification. Manual classification is the most basic means to set up classifications: creating classifications and putting items in classifications by hand. Virtual classification is a means to create classifications for software entities that exist in the software development environment, such as Smalltalk categories. Although virtual classifications have more purposes (see Section 13.3.6), in its current state the Classification Browser uses virtual classification only to provide starting points for browsing.

The recovery of collaboration contracts has been presented as a direct application of classification with advanced navigation tools. The Classification Browser supports the incremental approach to the recovery of collaboration contracts, as presented in Chapter 7. The recovery of a collaboration contract is based on incremental classification. An initial classification of classes passes through four stages of increasingly formal nature until it reaches a state in which it represents a collaboration contract. In an incremental way, starting from a set of participants that map on classes, methods, acquaintance relationships and method invocations are classified in those participants. To aid the software engineer in his recovery activities, the Classification Browser provides in-place browsing of senders and implementers, browsing of source code level acquaintances, scope reduction, and keeping a history of browsing actions. Indications for suggested exploration are given in the user interface, such as abstract methods, class hierarchy, and coloured text in method bodies.

The scope reduction property has a nice side effect. When browsing is restricted to a collaboration contract, the Classification Browser acts as collaboration contract browser. The software engineer can then use the browser to explore only the interactions in the collaboration contract, without interference of any other messages in the software system. The collaboration contract is then tangible in the software development environment.

Chapter 10

Automatic Classification Through Method Tagging

This chapter presents the software classification strategy *automatic classification through method tagging*, as introduced in Section 6.3.4. This software classification strategy is based on the provision of classification information during forward engineering in the form of method tags. Other automatic classification strategies are considered future work. One is briefly discussed in Section 13.3.6.

This chapter explains how method tagging is employed in the development of the broadcast management software. Results of method tagging during a period of almost two months are presented. Three applications of the software classification strategy are discussed: one in architectural recovery and two in software evolution.

10.1 Tagging Methods in "whats'On"

The method tagging approach to automatic classification has been adopted to classify the broadcast management software. The broadcast management software is developed with Envy/Developer. Envy has a versioning system. It keeps a history of all changes to methods, classes and applications. In particular, each time a method is compiled, the changed source code is stored in the Envy library. Envy stores a time stamp and the name of the developer along with the source.
When the development of the broadcast management software started years ago, this feature has been supplemented with a tag in the source code. This structured tag consisted of a piece of text, usually the developer's name, a timestamp, and a version number indicating the number of changes ever made to the method. The software engineer's name was asked when the development image was opened. Each time a method was compiled, the tag was changed and the version number was increased.

For this research, the tagging mechanism has been enhanced to include all desired tagging (classification) information. The following information is given by the software engineer, except for the first piece of information, which is provided by the development environment each time a method is compiled.

Time stamp. The date and time when the change was made.

Developer. The name of the software engineer that changed the method.

Activity. A brief description of the activity that gave rise to this change.

Site. The television or radio station for which this change was made.

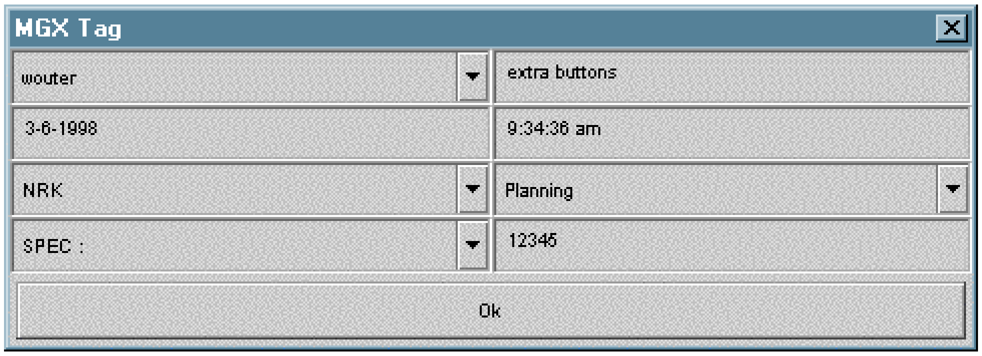
Module. The module in which this change was made, such as planning, video media management, programmes, etc.

Task. The task for which the change was made, such as new development, implementation of a specification, bug fix, code review, testing, etc.

Intention. For some tasks, more information must be given, such as the identification number of the specification or the bug fix (these identification numbers are assigned to specification and bug reports when they are received from a customer or a software engineer).

This information per method provides answers to several questions: Who changed the method? When was it last changed? For which specification or bug fix? For which station? For which module? And what was the reason to change the method?

In order to integrate the tagging in the development process, the development environment has been adapted so that tagging can be entered. All tagging information is requested each time a new version/edition of a class is made. Since software engineers usually carry out a given development task and many tagging parameters are fixed for a task, the tagging information has to be supplied only once per task.



wouter	extra buttons
3-6-1998	9:34:36 am
NRK	Planning
SPEC :	12345

Figure 10.1: Method tagging dialog for incremental classification

The software development environment pops up the tagging dialog window shown in Figure 10.1. It includes fields for all required information. Most tags are selected from pre-configured menus, so that selecting them can be done fast and easy. Other tags have to be entered textually. The date and time fields cannot be changed. They change each time a method is compiled. The other fields keep their values until the development task is finished.

The information supplied in the tagging dialog window is stored as a structured comment in the source code of the compiled method. The information provided in Figure 10.1 is stored as shown in Figure 10.2. The version number at the beginning is not shown in the tagging dialog window.

```
"<< 1.00 || wouter || 3-6-1998 || 9:34:36 am ||
extra buttons || NRK || Planning || SPEC: || 12345 >>"
```

Figure 10.2: Example tag with classification information

Tags can be parsed and interpreted so that they can be processed to produce classifications.

10.2 Processing the Method Tags

The classification information tagged on to the methods must be processed to make it available in actual classifications.

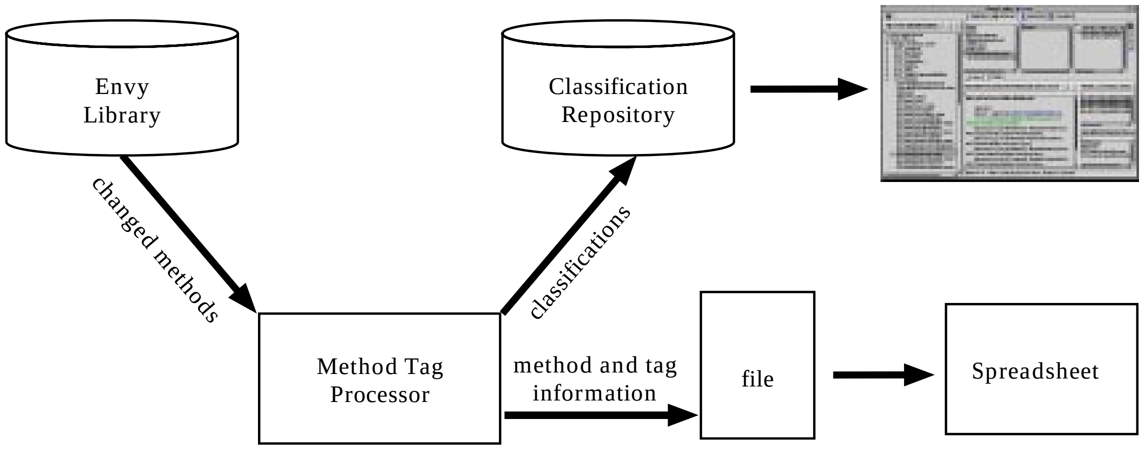


Figure 10.3: Processing method tags to produce classifications

Figure 10.3 shows how tag processing is done. The method tags are processed by the *Method Tag Processor*, a Smalltalk application that enumerates all changes during a given period, typically a day or a week[1]. All changed methods are collected and their tagging information is interpreted to produce classifications. These classifications are stored in a classification repository, so that they can be explored with the Classification Browser. The Method Tag Processor also stores the methods with their tags in a file that can be read by a spreadsheet application. The spreadsheet application can be used to analyse the tag information. This is useful for cross-referencing classification information and producing graphs of the results[2].

The Method Tag Processor generates ten main classifications (see Figure 10.4). Six classifications are simple classifications that map directly on the tags provided: changes per date, per developer, per intention, per module, per site, and per development task type (`Works` in the figure). The four other classifications combine the simple classifications. They contain cross-referenced information.

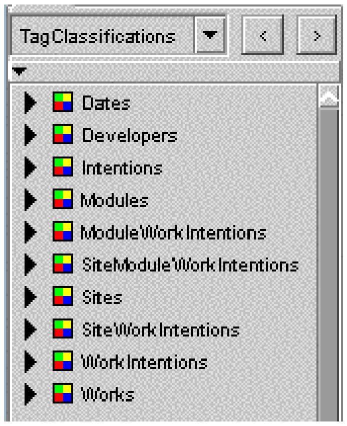


Figure 10.4: Classifications generated from method tags

10.3 Steady Increase in Classified Methods

In a period of almost two months, from April 20, 1998 until June 12, 1998, ten developers made 6807 method changes. Figure 10.5 shows the evolution in the number of classified methods per module. The tagging tool provides a choice between 23 modules to classify a method. In order not to clutter the graph, the figure shows only the seven modules with the most changes. The graph clearly shows that the number of classified methods per module steadily grows. The `Planning` module and the `Products` module were subject to many changes, around 1000 each. Modules `Programs` and `VMM` follow with nearly 600

[1]For this research, the method tags have been processed in batch. In the future, the method tag processor should be activated after each development task. This requires more changes to software development environment.

[2]The graphs shown in the following sections are created with a spreadsheet application based on a file generated by the Method Tag Processor.

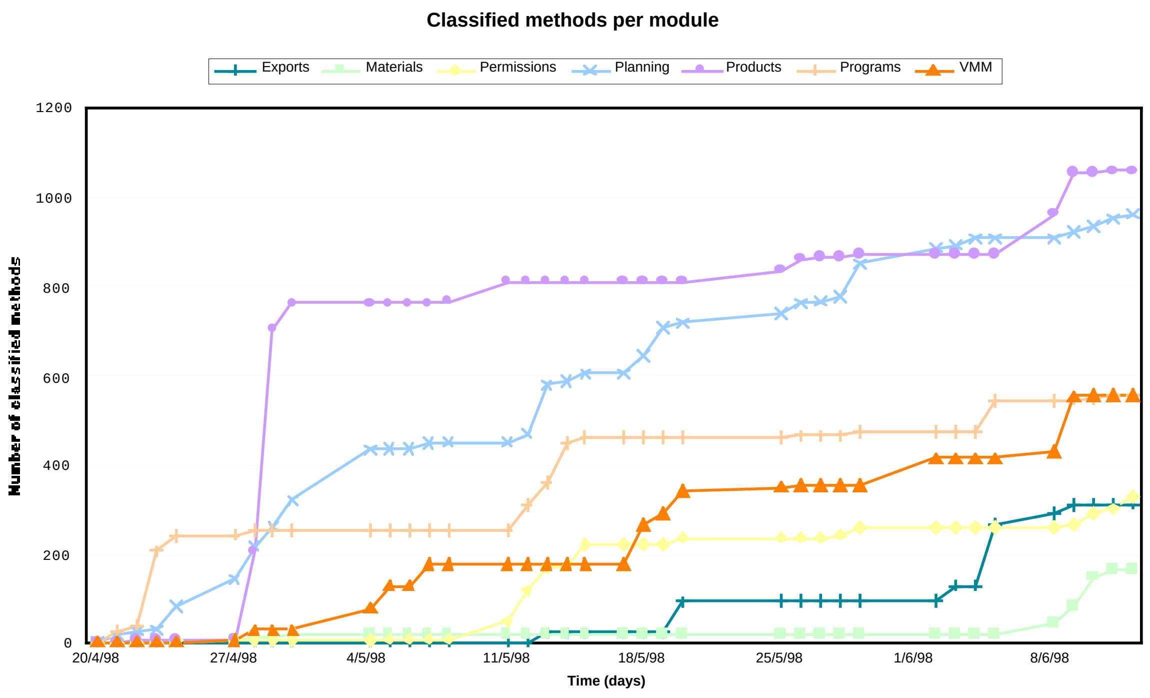


Figure 10.5: Number of incrementally classified methods over a two-month period

changes each.

The steep increment in the `Products` module at the beginning of the second week reflects major changes to the user interface layer of the `Products` module. The large amount of changes is also transparent in Figure 10.6, which shows the method changes per developer (on the left) and per module (on the right). The high value on the right is the number of changes made to the `Products` module by developer `Wouter`.

The chart in Figure 10.6 clearly shows that several developers change the same modules. It also shows that modules are tied to developers. The largest part of the changes to some modules is carried out by one developer. This reflects the expertise developers have in subareas of the broadcast management software. Development tasks are assigned to developers based on their expertise.

Feedback from the software engineers indicates that method tagging is not considered a burden and that it has advantages. From the software engineers:

- "Without too much additional effort, by entering tagging information once in a while, methods are supplied with structured comments."
- "The reference to the specification or bug fix is extremely useful to know what

the intention of a change was. Most of the time, that information is enough to understand why something has been changed."

- "The reference to the developer tells me who to ask for an explanation. Without the developer's name, the whole team must be bothered with my questions."
- "When methods have to be merged, the tagging information sometimes helps to solve conflicts. By checking why two conflicting changes were made, you can find a good solution to solve the conflict."

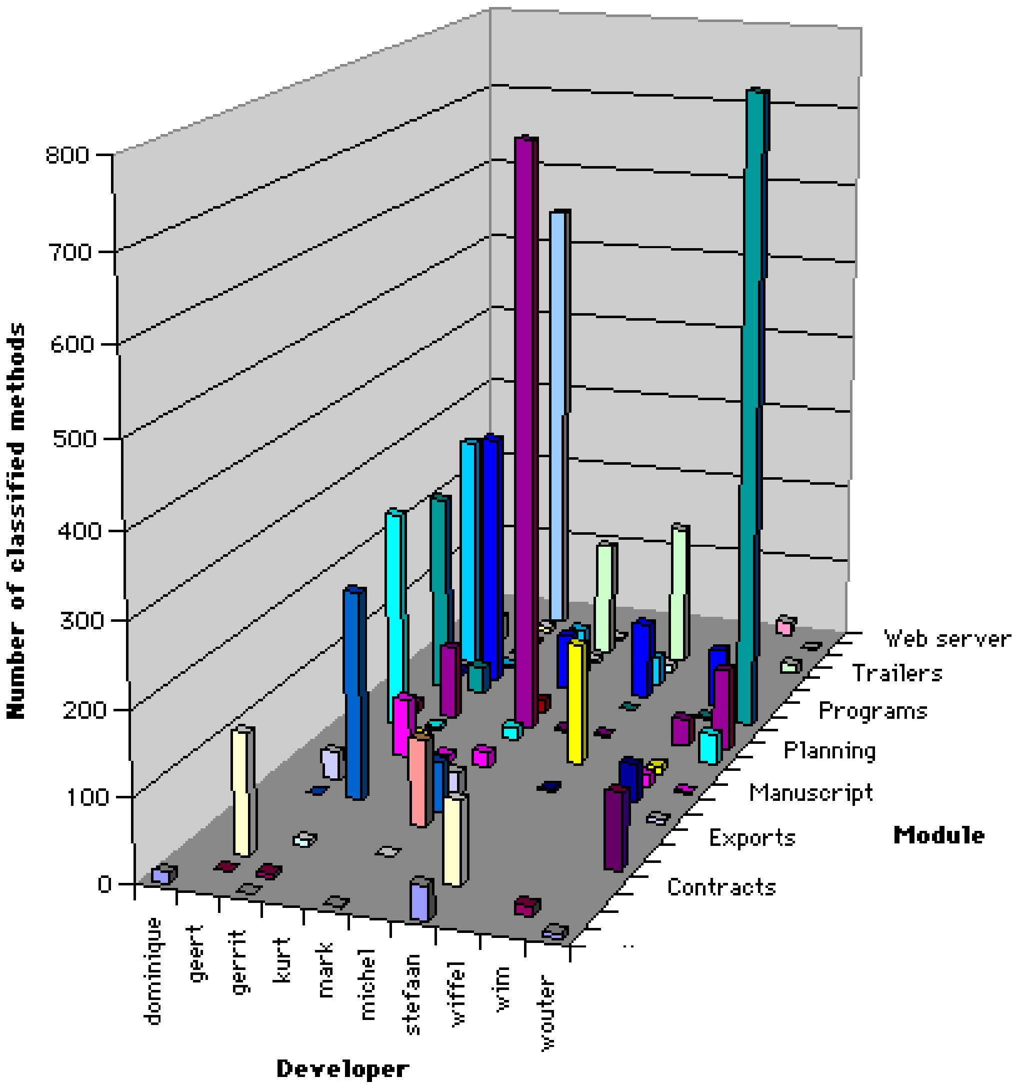


Figure 10.6: Number of incrementally classified methods per developer and per module

10.4 Application in Architectural Recovery: Recovery of Architectural Components

The provided tags for the subject software (see Section 10.1) include two entries that correspond to architectural components. The first one is the `Site` entry. Its value is selected from a pre-configured menu that includes all sites (television and radio stations). Each site corresponds to a framework customisation of the software. The second one is the `Module` entry. Its value is also selected from a pre-configured menu that includes all modules that the development team considers to be in the software.
When the software is being changed, each method is tagged with the site and the module it is part of. The Method Tag Processor generates classifications for each framework customisation and for each module (see Figure 10.7). These classifications hold partial classes, not single methods. The Method Tag Processor groups tagged methods according to the class they belong to. It creates a participant item to collect these methods. Each participant item in a classification is thus a view on a class that holds all methods that are changed in the context of the site or module represented by the containing classification.

Figure 10.7 shows part of the classifications that are generated by the Method Tag Processor. On the left, a classification `Sites` holds all framework customisations in the subject software. Each framework customisation is represented by a classification that holds participant items. The `TV2` classification is expanded to show its items[3]. The methods in a participant item can be browsed with the Classification Browser as usual. On the right, a classification `Modules` holds all modules in the subject software. The `Planning` module is expanded to show some of its items.

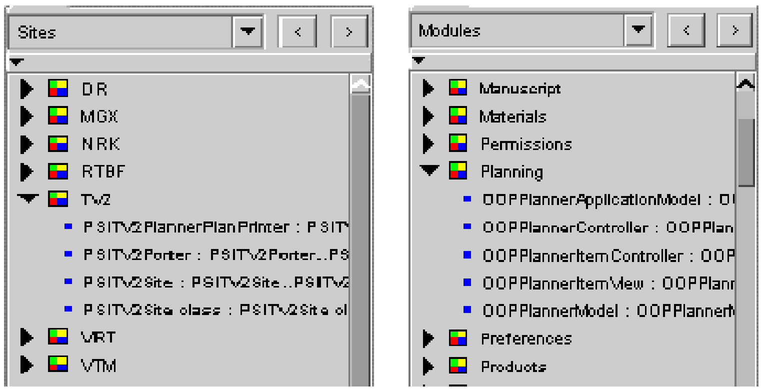

Sites
DR
MGX
NRK
RTBF
TV2
PSITV2PlannerPlanPrinter : PSIT
PSITV2Porter : PSITV2Porter..PS
PSITV2Site : PSITV2Site..PSITV2
PSITV2Site class : PSITV2Site cl
VRT
VTM
Modules
Manuscript
Materials
Permissions
Planning
OOPPlannerApplicationModel : O
OOPPlannerController : OOPPlan
OOPPlannerItemController : OOP
OOPPlannerItemView : OOPPlann
OOPPlannerModel : OOPPlanner
Preferences
Products

Figure 10.7: Recovered framework customisations and modules

The recovery of architectural components with the presented automatic classification strategy results in multiple architectural views (see Section 4.1) on software. By browsing the

[3]Actually, the classification holds a lot more items than shown here. The number of items has been limited to keep the figure clear.

generated classifications, the software engineers get a picture of the software in terms of architectural components. The conceptual components that live in their heads are now physical entities in the software development environment.

Although the tagging technique is used at the fine-grained level of methods, changing one method of a class is enough to classify a class in a classification that represents an architectural component that consists of classes. This suggests such classifications are recovered rapidly. The classifications are however not complete until all classes of the target software have been changed. The software engineers have to take that possible incompleteness into account when they use the generated classifications before all classes have been classified.

10.5 Applications in Software Evolution

The method tags include information about the changes made to the methods. The `time stamp` part records the date and time when a method has been changed. The `task` part describes why a method has been changed. The `intention` part records more information about the task, such as the identification number of a specification or bug report.
This information about the evolution of a method is provided to exploit it afterwards. This section presents two applications of automatic classification through method tagging that exploit the information about evolution: management of changes, and methods and classes in multiple classifications.

10.5.1 Management of Changes

In Section 3.1.8, an example request/defect system was discussed. The top part of Figure 10.8 repeats Figure 3.2 of Section 3.1.8 to show the differences between a request/defect system when method tagging is not used, and a request/defect system when method tagging is used and tag-based classifications are available in the Classification Browser.
When method tagging is not used (see the top part of Figure 10.8), the software engineer and the code reviewer have to log the changes in the request/defect database manually. The code reviewer fetches the changes from the request/defect database and queries the software system for source code that corresponds to the changes. Each changed method is possibly part of another class, so that the code reviewer has to browse several classes, usually using several browsers.

When method tagging is used (see the bottom part of Figure 10.8), the software engineer and the code reviewer do not need to log the changes anymore, but they provide method tag information instead. They provide that information in the method-tagging dialog only once for the implementation of one specification or one bug fix. When the dialog pops up when a new version of a class is made, that information is still available in the method-tagging dialog. Confirming the dialog is all that has to be done. The overhead of logging the changes in the request/defect database is thus reduced considerably.

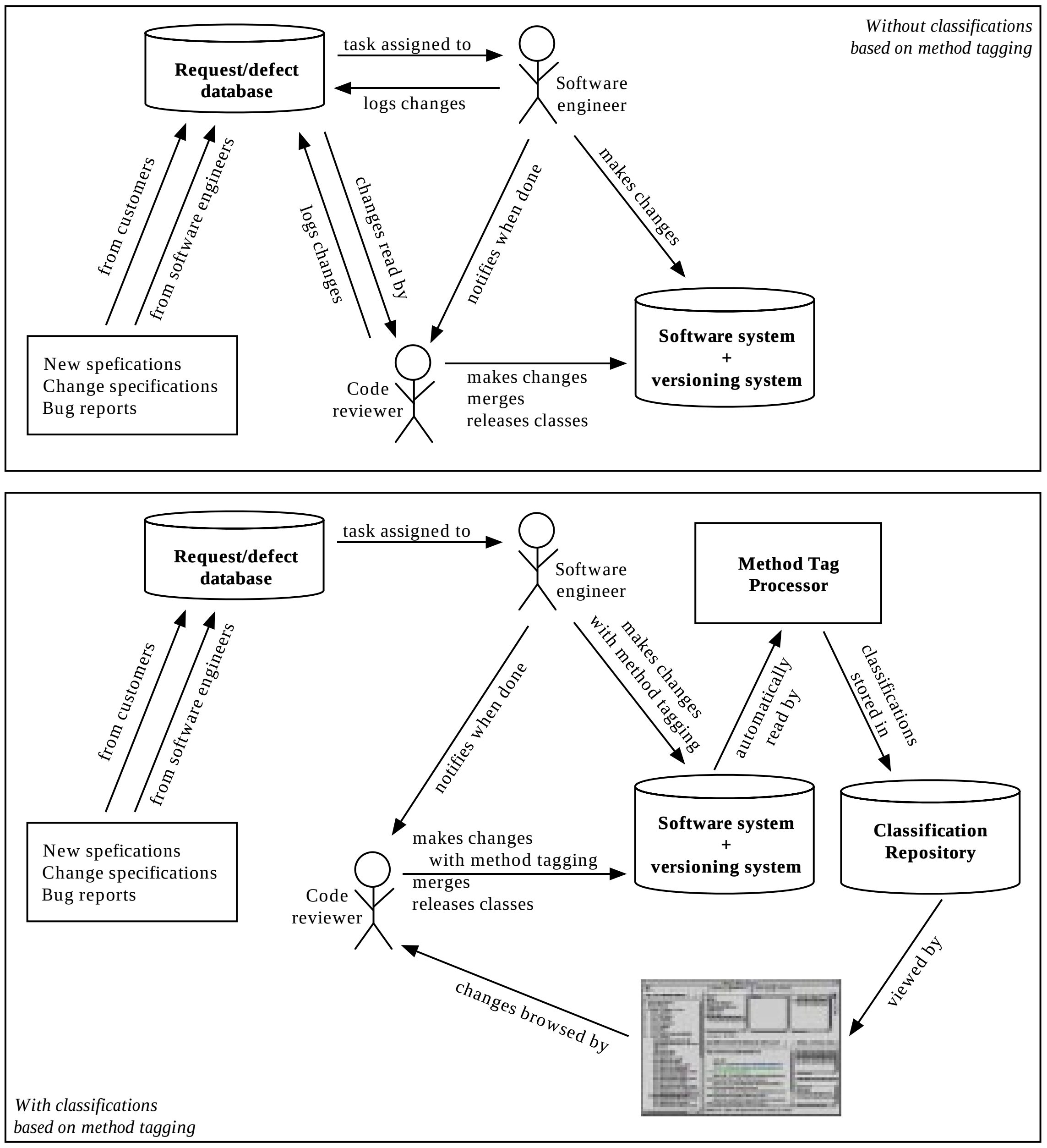
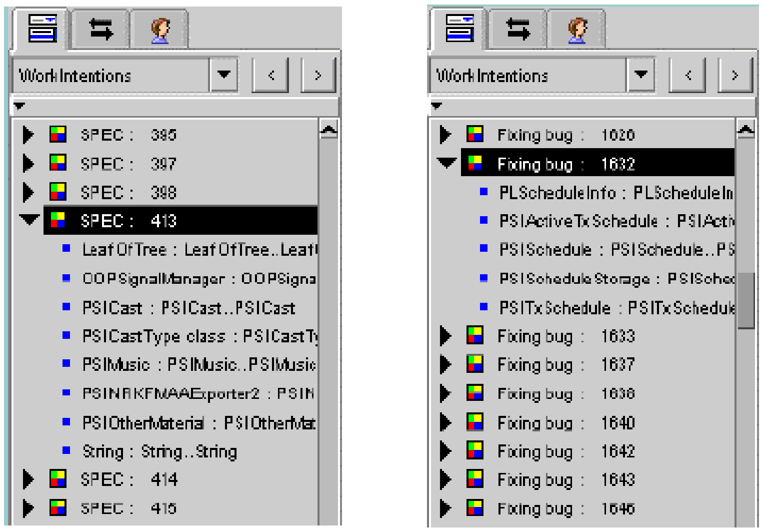


Figure 10.8: Making, logging and tracking changes with tag-based classifications

The method tags are processed automatically by the method tag processor. The resulting classifications are stored in the classification repository, which makes them available in the Classification Browser (see Figure 10.4). Based on the identification number supplied by the software engineer, the code reviewer is able to locate the corresponding specification or bug report easily. The Classification Browser lists a classification 'WorkIntentions' that holds all specification implementations and bug fixes, sorted by identification number. All specification implementations and bug fixes show up as classifications with items that represent class changes (see Figure 10.9). Each item is in fact a participant that holds the methods of a class that was changed for the specification or bug report under consideration. The code reviewer has an overview of all changes. They can be browsed immediately. The Classification Browser shows the changed methods in their context (i.e. their class), as usual. No special or additional browsers are necessary.

Figure 10.9 shows two parts of the `WorkIntentions` classification. On the left, the changes corresponding to specification 413 are displayed. Eight classes were changed for this specification. On the right, the content of bug fix 1632 is displayed. Five classes were changed for this bug fix.



```
WorkIntentions
SPEC :  395
SPEC :  397
SPEC :  398
SPEC :  413
    LeafOfTree : LeafOfTree..Leaf
    OOPSignalManager : OOPSigna
    PSICast : PSICast..PSICast
    PSICastType class : PSICastT
    PSIMusic : PSIMusic..PSIMusic
    PSINRKFMAAExporter2 : PSIN
    PSIOtherMaterial : PSIOtherMat
    String : String..String
SPEC :  414
SPEC :  415
```

```
WorkIntentions
Fixing bug :  1620
Fixing bug :  1632
    PLScheduleInfo : PLScheduleIn
    PSIActiveTxSchedule : PSIActi
    PSISchedule : PSISchedule..PS
    PSIScheduleStorage : PSISched
    PSITxSchedule : PSITxSchedule
Fixing bug :  1633
Fixing bug :  1637
Fixing bug :  1638
Fixing bug :  1640
Fixing bug :  1642
Fixing bug :  1643
Fixing bug :  1646
```

Figure 10.9: Changes corresponding to specifications and bug fixes in the Classification Browser

The `WorkIntentions` classification supplies a global overview of all changes made in the context of specifications and bug reports. The Method Tag Processor also generates classifications for site specific specifications and bug reports. The `SiteWorkIntentions` classification consists of subclassifications that hold changes that were made in the context of a site (television or radio station) and some development task. These classifications carry the site, the development task, and the intention in their names. Figure 10.10 shows part of the `SiteWorkIntentions` classification. Only bug fixes are shown. Some subclassifications carry a numerical description of the intention (the number of the bug report) in their name, while other classifications carry a textual description of the intention. The subclassifications include participant items that hold all methods that were changed in the corresponding class in the context of the bug fix. Classification `DR - Fixing bug: 1853` contains four participants (large bug fix). `DR - Fixing bug: Close of contract` contains only one participant (small bug fix).

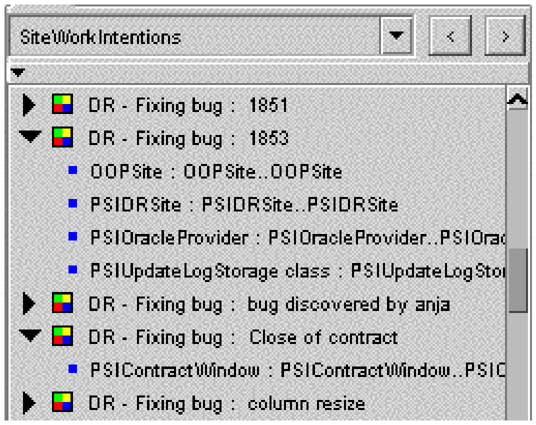

SiteWorkIntentions
DR - Fixing bug : 1851
DR - Fixing bug : 1853
OOPSite : OOPSite..OOPSite
PSIDRSite : PSIDRSite..PSIDRSite
PSIOracleProvider : PSIOracleProvider..PSIOrac
PSIUpdateLogStorage class : PSIUpdateLogStor
DR - Fixing bug : bug discovered by anja
DR - Fixing bug : Close of contract
PSIContractWindow : PSIContractWindow..PSIC
DR - Fixing bug : column resize

Figure 10.10: Exploring method changes in context

The classifications used here provide a traceability view (see Section 4.2.2) on the software. Before tagging of the broadcast management software started, the information provided by the `WorkIntentions` and `SiteWorkIntentions` classifications was impossible to obtain. Now, this information is readily available in the software development environment. It is a major asset to the project managers. They are now able to assess why a change was made, who made it, and which methods were changed for a given development task. The classifications presented in this section are currently used by the code reviewer of the development team that develops the broadcast management software.

10.5.2 Methods and Classes in Multiple Classifications

As shown in Figure 10.6, developers may change the same module. They may even change the same methods. Such multiple changes to methods are potential causes for inconsistencies or bugs in the software. This problem has actually a larger scope than modules. Methods can be changed for different modules, for different framework customisations, or for different specifications or bug fixes.

Based on the tags in the methods of the broadcast management software, multiple classification of methods provides insight into the overlap of modules. Figure 10.11 shows the number of classes that is classified in each couple of modules. A class is classified in a module if at least one of its methods is tagged with that module's name. Figure 10.12 shows the same information, looking down from the top of Figure 10.11.

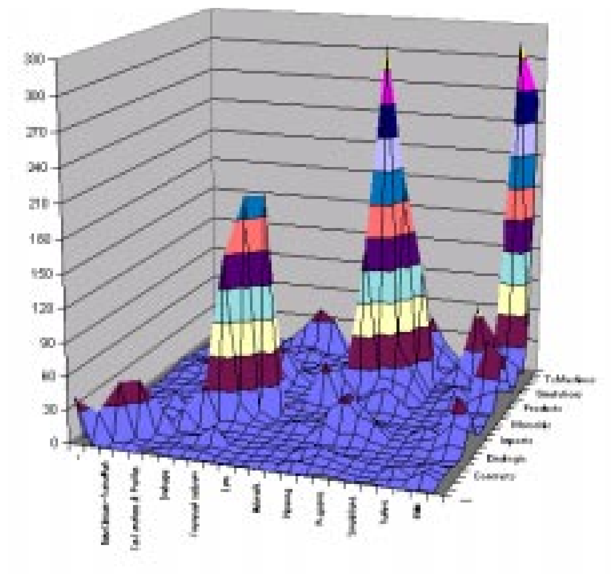


Figure 10.11: Number of classified classes in two different modules

Ideally, that is, when the module architecture is not breached, the graph should be a diagonal line. Each class is then part of one module only. Figure 10.11 and Figure 10.12 show clearly that this is not the case, although the diagonal is observable. The highest peaks are located on the diagonal, which means that many classes are part of only one module. Figure 10.12 shows some islands of classes beside the diagonal, including the arms of the turtle-like figure in the middle of the graph. These islands include classes that are classified in multiple modules.
The figures indicate that hard module boundaries cannot be defined yet. The multiple classification of classes must be eliminated first. The problem is, however, that maybe a clear separation of classes in modules is impossible because the wrong modules may have been chosen. A careful investigation of the reasons why classes are classified in multiple modules must be conducted to shed light on this issue. Each method that is responsible for the multiple classification of a class must be investigated and its tag must be reconsidered. Maybe the method's tag is wrong or maybe the method inherently belongs to more than one module. Solving the former problem may result in less multiple classifications. Solving the latter problem may result in the reconsideration of the modules currently in use. Such investigation takes a lot of time and many resources, but it could result in a better architecture.

Multiple classification of methods also gives useful information for specifications and bug fixes. Multiple changes to the same method are solved by the code reviewer, who decides which version of the method should be kept. Three cases can be distinguished. One, the original version is kept when the method should not have been changed. The developers have to implement their specifications or bug fixes otherwise. Two, one of the two new

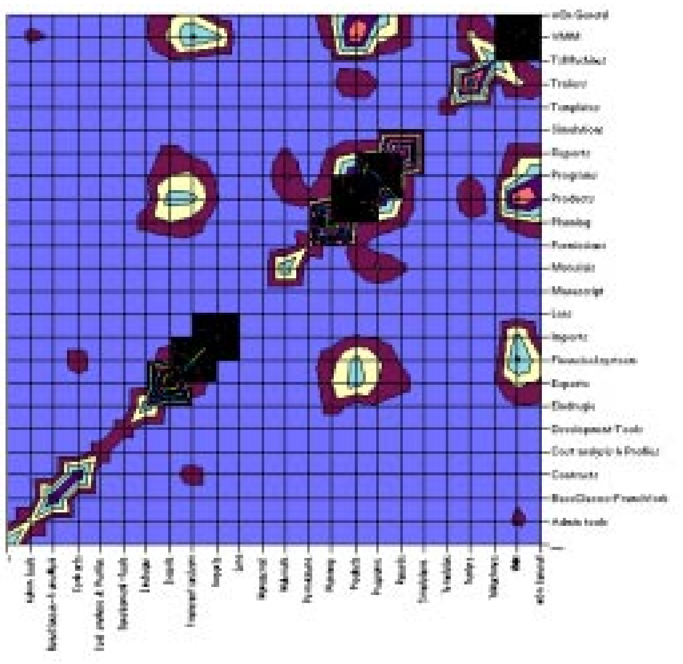


Figure 10.12: Classes classified in two different modules

versions is kept, so that one of the two developers has to provide another implementation. Three, the two versions are merged.

Classification information helps the code reviewer to assess whether problems may be expected. For example, the following two generated subclassifications from the `SiteModule-WorkIntentions` classification (see Figure 10.4) were found to include the same method (real station names suppressed): `STATION1 - Planning - SPEC : 23` and `STATION2 - Planning - SPEC : 395`. The method that is changed by both classifications is `redButtonPressedEvent:` in class `PSIPlannerItemController`. As can be read from the names of the classifications, these methods were changed for different sites (`STATION1` and `STATION2`), for the same module (Planning), and for different specifications (`23` and `395`). Further investigation reveals that the changed method is part of the framework of the software, not part of the customisations for `STATION1` and `STATION2`. This information is very helpful for the code reviewer who can identify two problems immediately. First, changes to the framework affect all sites, not only `STATION1` and `STATION2`. Therefore, these changes may be dangerous and must be further investigated by looking at the source code. Second, the two method changes were made for different specifications. That means that the specifications are dependent and should be compared.

In general, when a method is classified in two specifications, the method appears to play a role in both specifications. It is possible, however, that the version of the method for the specification implemented last does not conform to the specification implemented first. The new version may even introduce bugs for the other specification. Multiple classification of a method thus indicates that the method should be tested for all specifications in which it plays a role. This suggests that multiple classification of methods may be useful to steer regression tests. How and when regression tests should be done remains an open question at this time. More research must be devoted to this issue in order to determine the relation between multiple classification and testing.

10.6 Summary

This chapter has presented *automatic classification through method tagging*, a software classification strategy to recover classifications of large software systems. The proposed classification strategy is based on method tagging during forward engineering. Each time a method is compiled, it is tagged with classification information, such as the developer, the identification number of the specification or bug report for which the changes was made, the module, the framework customisation, the reason why the changes were made, etc. These tags are stored in the source code of the methods, so that the software engineers can read them and use them during future development activities. The tags are processed by the Method Tag Processor. It reads the method tags and produces classifications based on the classification information found in the tags. These classifications are stored in the classification repository, so that they can be consulted by the software engineers and in particular by the code reviewer.

Two applications of the classification strategy were presented: architectural recovery and software evolution.

The application in architectural recovery requires that information about the architectural components is provided in the method tags. Based on the tagging information about the architectural components a method is part of, classifications for modules and framework customisations can be generated by the Method Tag Processor. In the examples used, two architectural components were considered: modules and framework customisations. Software engineers now have direct access to the organisation of classes in modules and framework customisations. The conceptual components that lived in their heads before are now tangible entities in the software development environment.

The application in software evolution requires that evolution information is tagged onto methods. Automatic classification through method tagging integrates well with a request/defect tracking system. Changes to the software are automatically logged and the generated classifications provide traceability views on the software that can be used by the code reviewer to rapidly find out what changes were made for a given specification or bug report.

10.7 Considerations for Introduction of the Technique in a SDE

The proposed software classification strategy has been (and is) successfully used in the context of the broadcast management software. For a large part, this success can be ascribed to the support and the commitment of the software provider's managers. That support was essential, because automatic classification through method tagging requires changes to the development tools and to the software development process. Due to these inevitable requirements, in other contexts management may not be inclined to adopt this automatic classification strategy.

Because of the required changes to the software development environment, this classification strategy cannot be used in black-box software development environments. This technique requires at least an intervention in the compilation process. In order to integrate the Classification Browser in the development environment, the environment must support the addition of extra tools, be it by providing a plug-in interface (such as in SniFF [Sof98]), or by opening up the implementation of the environment (such as in VisualAge for Java [IBM98]).

Automatic classification through method tagging, as the name suggests, is targeted to software development environments where object-oriented systems are developed on a method-by-method basis. However, the tagging technique can be used in a more coarse-grained way, by tagging classes (or files in some object-oriented languages) with classification information. The coarse-grained tagging would then result in more coarse-grained classifications. The Method Tag Processor would generate classifications with only class changes, not method changes.
Many consider method-by-method development a unique characteristic of Smalltalk environments. Note, however, that some modern software development environments for other object-oriented languages, such as VisualAge for Java [IBM98], also support method-by-method development. If these environments are open enough, the method tagging technique proposed here can be ported as-is.

The success of the applications of automatic classification through method tagging can be completely ascribed to the choice of information that must be included in the method tags. After all, the generated classifications can only include information that has been tagged. For example, the tags used to automatically classify the broadcast management software do not include information about the software layer in which a changed method resides. Consequently, there are no generated classifications for the software layers in the software.
Therefore, it is very important to consider carefully what information must be tagged. Not only the parts that make up the method tag must be chosen in advance; the values that can be entered for each part of the tag must be known in advance as well. When automatic classification through method tagging is considered for adoption, the development team

must determine the tags and the values. Although more research on this issue is necessary, the conjecture is that the information about evolution will parallel the information used for the broadcast management software. For architectural views, much more information can be provided, depending on the architectural and other views that exist in the heads of the software engineers. When the development team determines the values that can be entered for architectural views, it makes explicit the conceptual views on the software that exist in the heads of the software engineers. Through method tagging, these conceptual views are mapped onto physical views that describe how physical methods and classes map onto conceptual architectural components.

Chapter 11

How the Proposed Solution Meets the Requirements

Chapter 5 collected requirements that should be met in order to make the model, the recovery process, and the recovery tools practical. This chapter discusses how the stated requirements are met by the proposed software classification model, the software classification strategies, the recovery process as a whole, and the tool support.

11.1 Take Multiple Views on the Software into Account

The software classification model meets this requirement excellently. Indeed, the model has been conceived with multiple views in mind. The decision to support multiple views was driven by the observation that there is not one predominant software architecture. Object-oriented systems often have a layered architecture to separate the different software layers, and a framework architecture to describe the stable and the variable parts of the software. On top of supporting multiple views on the architecture, software engineers like to organise the software in a user-defined way, so that they can take a view on the software that best matches the development task at hand.

The software classification model allows for multiple views by stating that items can reside in multiple classifications. By grouping classes in multiple classifications, the software engineer is able to create different views on the software. The examples and the experiments described in the previous chapters show classifications of classes that represent software layers, modules, framework customisations, etc.

Multiple classifications are also useful for describing the different collaboration contracts in which a class participates. Each collaboration contract defines a different role played by the class, so the collaboration contracts provide multiple views on the roles played by the class.

Although multiple views are very useful to organise classes and to model the software as compositions of components, some enhancements concerning the combination of classifications would be useful. For instance, in Section 4.1.2, a combined modules/layers view expressed the relationship of modules and software layers (see Figure 4.3). In the combined view, the user interface layer of the planning module could be identified. It is however not possible to create a classification that represents the user interface layer of the planning module based on two classifications that represent the user interface layer and the planning module, unless by manual classification. For this example, it would be advantageous to have set-theoretic operations on classifications, so that the intersection of classifications can be expressed. More research is necessary to determine which combination of operators on classifications are desirable and useful.

11.2 The Recovery Process Should be Incremental

The recovery of collaboration contracts is incremental by definition. Collaboration contracts are recovered by incremental classification. The proposed method for the recovery of collaboration contracts is based on the identification of classes, methods, and acquaintance relationships that are candidates for inclusion in a collaboration contract. Clues, guidelines and heuristics help the identification process. Inclusion in the collaboration contract is performed by incrementally classifying participants, methods, acquaintance relationships, and method invocations.

The recovery process is incremental in another aspect as well. Since the results of recovery are recorded in the development environment, the software engineer can easily interrupt an ongoing recovery. Classifications hold the state of a recovery, so that the software engineer can continue with the recovery later, even much later. The partial classification then serves as a mnemonic device.

The recovery of architectural components based on method tagging is incremental because at a certain point in time, the architectural components of the software system under consideration are only known partially, depending on the number of methods (and consequently classes) that has been tagged so far.

The Classification Browser does not introduce modes. The software engineer is able to switch seamlessly between programming, browsing and recovery.

11.3 Motivate the Software Engineer

The introduction of software classification in the development process employed to build the subject software has not been simple. In a prototypical manner, software classification, the classification tools, and the recovery tools have evolved from models and tools that were not accepted at all, to a model, a method tagging tool, the Classification Browser, and a process for collaboration contract recovery of which the software engineers and the project leaders benefit daily in their development activities.

The method tag at the end of a method body, as simple as it may be, is experienced as information that can not be missed anymore. The method tag informs the reader about the developer who created or last changed the method, the module and the framework customisation the method belongs to, and the specification or bug report the method was changed for.

The free-text field in the method-tagging tool has been used in interesting ways. Software engineers use this field to provide more information on the reason why they change a class. The software engineers appear to use this field frequently, although it is not mandatory. Although this positive result was not expected, it shows the motivation of the software engineers. The small amount of extra work necessary to provide tagging information pays off afterwards.

The extra work also pays off when the classifications generated by the method tag processor are available in the development environment. Those classifications supply information that is hard to obtain when classifications or similar structures are not available. The availability of this information in the development environment results in a reduction of the communication overhead among developers. Some questions can be answered by consulting (browsing) the generated classifications, instead of appealing to other members of the development team for help.

The recovery of collaboration contracts requires little extra effort compared with the browsing activities that are required to identify invocation dependencies anyway. The additional effort is necessary to classify participants, methods, and acquaintance relationships. The software engineer is motivated to do the extra work for three reasons. First, the overhead is minimal. The Classification Browser has good browsing and classification capabilities that help in the extra classification effort. Second, the extra effort results in software documentation that is available online in the development environment, so that it can be used for subsequent development tasks. Third, the software engineer is able to generate diagrams that can be used as external documentation during group activities, such as design and code reviews.

11.4 Keep the Model Simple

The software classification model is extremely simple. There are only two entities: item and classification. When an item is part of a classification, we say that the item is classified in that classification. Items may be classified in multiple classifications. In this dissertation, items are mainly classes, but anything that is available in the software development environment qualifies as candidate item. To support hierarchical composition of classifications, the model allows classifications as items.
Due to its simplicity, the software classification model is easy to understand and learn.

Moreover, its uniform nature facilitates the construction of classification browsing tools.

11.5 Keep the Recovery Process Lightweight

Collaboration contract recovery is lightweight in two aspects. First, recovery of a collaboration contract involves a small extra effort compared to the effort needed to browse a software system. The extra effort is needed to classify classes, methods and acquaintance relationships. The Classification Browser provides classification commands in all parts of the browser, so that the overhead is minimised.

Second, the process relies on automatic extraction whenever possible and feasible. Automatic extraction is used to compute the class of a source-level acquaintance and to compute the specialisation clauses of participants. The algorithm to compute acquaintance classes is simple: a candidate acquaintance class is a class that implements the required interface of a source-level acquaintance. The required interface of a source-level acquaintance can be extracted from the source code. The computation of the acquaintance classes is based on functionality that is available in most object-oriented development environments: senders and implementers. With efficient lookup strategies, the computation of acquaintance classes is instantaneous. Computing acquaintance classes is lightweight because it is based on a simple but fast algorithm that computes a conservative approximation. In general, many candidate acquaintance classes are found. However, through scope reduction with classifications, in many cases a single acquaintance class is found.

The recovery of reuse contracts is also lightweight, because the computation of the reuse contract does not involve the software engineer. Only an initial and a derived collaboration contract must be provided. The reuse contract that describes how the latter is derived from the former can be computed automatically.

One important reason to set up reuse contracts is to use them for software change impact analysis. The ability to compute reuse contracts automatically is particularly useful in that respect. Impact analysis can now be performed when initial and derived collaboration contracts are provided. The software engineer is relieved from manually setting up reuse contracts, an activity that is tedious and error-prone. That means that the software engineer can focus on setting up collaboration contracts.

The recovery of architectural components through method tagging is lightweight because it requires little effort from the software engineers. They need to provide tagging information only once per development task. The system takes care of processing the tags and producing tag-based classifications.

11.6 Integrate the Model and the Recovery Process in the SDE

At this time, the software classification model has been integrated in one tool, not in the development environment as a whole. For instance, classifications have no influence on debugging and classifications cannot be versioned. More research must be committed to the integration of classifications in other developments tools (see Section 13.3.7). Due to the limited transparency and openness of the employed development environment (Envy/Developer [OTI95]), it appeared to be easier to build a new development tool based on the software classification model that is backward compatible with the main development tools. The resulting Classification Browser is an enhanced version of its standard counterpart, and provides extensive support for recovery of collaboration contracts and architectural components. At this time, reuse contracts are not integrated in the Classification Browser[1].

Method tagging has been integrated in the software development process. The standard development tools have been adapted to integrate the method tagging dialog. It pops up each time a software engineer makes a new version of a class.

11.7 Integrate Existing Software Entities in the Model

The software classification model is simple, but it is an open model. Virtual classifications allow the introduction of classifications of which the items are computed. This feature has been successfully used to draw existing software entities into the model. The subject software is developed with Envy/Developer, which includes categories of classes, and applications of classes (sort of hierarchical categories with extra semantics). For both entities, special virtual classifications were developed. These virtual classifications compute their items (i.e. classes for Smalltalk categories and classes and subapplications for Envy applications) based on information provided by the development environment itself. Category classifications and Envy application classifications query the development environment for the classes that reside in categories and Envy applications respectively. The result is that software entities familiar to the software engineers are available in the software classification model, and thus in the Classification Browser. Categories and Envy applications can be manipulated as any other classification, while the usual commands to work with categories and Envy applications are still available.

11.8 Make the Results of Recovery Tangible in the SDE

The software classification model is integrated in the software development environment through the Classification Browser. Recovered classifications, including collaboration contracts, can be consulted on-line in the development environment. At this time, reuse

[1]There are no technical reasons that inhibit integration of reuse contracts, however.

contracts are not integrated.

Besides being tangible in the development environment, the results of recovery can be exported to files that can be read by other applications. Classifications can be exported to text files, and collaboration contracts can be exported to script files for external diagramming tools. Results of recovery can thus be included in the paper software documentation and can be shared among software engineers during design reviews and other group activities.

11.9 Cope with an Obscure Software System

The software classification model, the software classification technique, and the recovery of architectural components make no assumptions about the target software. The recovery of collaboration contracts, and consequently the recovery of reuse contracts, also make no assumptions about the software on which they are applied, but the algorithm to compute acquaintance classes does not work well on source code that includes metalevel code. The algorithm is able to give an indication when an acquaintance class cannot be determined. In that case, the software engineer has to determine the acquaintance class by hand.

Chapter 12

Related Work

This work relates to work in many areas: software evolution, design models, software architecture, reverse engineering, and software development environments. This chapter makes a selection of related work in the following fields: reuse contracts, classification, determining acquaintance classes, reverse engineering, and program visualisation.

12.1 Reuse Contracts

Reuse contracts were introduced by Steyaert et al [SLMD96]. The emphasis was put on reuse by inheritance. Evolution problems showed up as inconsistencies introduced in subclasses after a change in their superclasses. Reuse contracts were further developed and formalised by Lucas [Luc97], who extended the model with inter-class relationships (reuse by composition or delegation). Reuse contracts have been discussed in the context of composability issues [LSM96], component-oriented programming [HLS97], systematic reuse [SLM97], adaptable systems [MSL96], and managing software evolution [LSM97]. The latest report on reuse contracts comes from Mens et al who report on a version of the reuse contract model that is more conform to the UML [MLS98]. This work is the first to treat development environment issues concerning collaboration contracts and reuse contracts.

12.2 Models

Several models are related to collaboration contracts. Many object-oriented design methodologies have a model for object interaction. Booch [Boo93] has object diagrams, OOA/OOD [CY91] has message connections, OMT [RBP$^+$91] has sequence diagrams, CRC [WBW89] has collaboration graphs, Fusion [CAB$^+$94] has interaction graphs, and UML [OTI97] has collaboration diagrams (and equivalent sequence diagrams). These models and collaboration contracts differ in two important respects. First, these models are not targeted to reuse and evolution. Second, these models describe the dynamic behaviour of components, while collaboration contracts describe the static behaviour. This quality accounts for the fact that collaboration contracts can be reverse engineered from source code.

Helm, Holland and Gangopadhyay's interaction contracts [HHG90] come close to collaboration contracts, as they are formal descriptions of object interactions. There are no reports on tools and the practical use of interaction contracts, however.

12.3 Classification of Reusable Components

Classification of software components has already been examined in the context of libraries of reusable components. Prieto-Díaz and Freeman [PDF87] (also see [PD91]) propose a faceted classification scheme for cataloguing reusable software components. The classification scheme helps with locating and retrieving software components from a large collection of components.

The purpose of a classification scheme is to produce systematic order based on a controlled and structured index vocabulary. The index vocabulary lists names of concepts and classes in a systematic way to show the relationships between classes (see the left column of Table 12.1). Classification schemes express hierarchical and syntactical relationships. Syntactical relationships express relationships between concepts in different hierarchies. Classification schemes come in two flavours: enumerative and faceted.

Enumerative classification is a hierarchical classification in which a class at a certain level narrows the class of the level above. Compound classes are used to express syntactical relationships. For example, the right column of Table 7 shows an enumerative classification based on the element classes in the left column of the table.
Strict hierarchical classification suffers from problems also encountered when modelling multiple inheritance in single-inheritance object-oriented programming languages. When a class can be placed at two (or more) positions in the classification hierarchy, compound classes need to be created to express the two (or more) relationships with other classes. In the worst case, a compound class must be created for each combination of the elemental classes.

Maintenance	Maintenance
Cleaning	Cleaning
Repairing	Repairing
Vehicles	Vehicles
Land vehicles	Land vehicles
Water vehicles	Cleaning land vehicles
	Repairing land vehicles
	Water vehicles
	Cleaning water vehicles
	Repairing water vehicles

Table 12.1: Elemental classes and compound classes in enumerative classification

Faceted classification takes another approach. It relies on synthesis of terms that make up subject statements of documents. The index vocabulary of a faceted classification scheme includes the elemental classes found in the subject statements. A facet is a group of elemental classes in a classification scheme. In the example, `Maintenance` and `Vehicle` are facets. Facets have a citation order according to their relevance to users. For example, when the citation order is `Maintenance` $\rightarrow$ `Vehicle` and the title to classify is `The Repair of Cars`, the title would be classified in classification `Repairing/Cars`. If the order were the other way around, the title would have been classified in classification `Cars/Repairing`. In both cases, `Cars` should be added under `Land Vehicles`. In an enumerative classification, in absence of a class `Repairing cars` (as is the case in the example), the librarian must decide whether to classify `The Repair of Cars` under `Cars` or under `Repairing`. Faceted classification is thus more flexible.

Prieto-Díaz and Freeman employ a facetted classification scheme for reusable software components. They propose a component description format based on a standard vocabulary of terms and they impose a citation order for the facets. The component description format is a triple `<function, object, medium>`. `Function` is the name of a function, `object` is the collection of objects manipulated by the program, and `medium` is the locale where the function is executed. Examples are [PDF87]: `<input, character, buffer>`, `<search, root, B-tree>`, and `<compress, lines, file>`. The facets used for these examples are:

- Function: add, append, close, compare, compress, insert, join, ...
- Objects: arrays, blanks, buffers, directories, files, lists, pages, ...
- Medium: buffer, cards, file, printer, stack, tape, tree, ...

The component description is the classification of the component. Retrieval of a component is based on input of terms. A metric for conceptual distance between terms in each facet helps to find related components.

A major difference between faceted classification and software classification proposed here is that reusable components in faceted classification have a unique component description (lookup key). Each component belongs to one class, while our approach allows components to reside in more than one classification. This quality is essential to provide multiple views on software.

Another major difference is the motivation: faceted classification is concerned with cataloguing reusable software components in a predefined manner. This work is concerned with the organisation of software in a flexible, user-defined way. Our classifications can be used to steer the browsing process, while faceted classifications cannot be navigated easily.

12.4 Determining Acquaintance Classes

12.4.1 Type Inference and Type Systems

Determining the class of a source code level acquaintance sounds like a typing problem. In that view, one solution to this problem is a type inference engine. In the last 18 years, several people have tried to retrofit a type system to Smalltalk, but most of them failed to deliver practicable results. Many solutions work nicely for a subset of the Smalltalk language only. This is no surprise since type inference is harder for object-oriented languages than for traditional languages, because the control-flow graph is only known in advance up to an approximation due to late binding. Type inference for object-oriented languages is thus a combined control-flow and data-flow problem [AH95]. Although many papers have been published on the subject [BI82], [Joh86], [JGZ88], [PS91], [BG93], [AH95], many of them do not mention the performance of the type inference algorithms, which is a very important factor. It seems that the best results come from Graver's work [Gra89]. His type inference algorithm is able to type close to all Smalltalk code, but it requires explicit type declarations, which makes his algorithm inadequate for existent Smalltalk code.

Apart from the technical and performance problems involved with implementing a type inference engine for an object-oriented language, type inference approaches struggle with the definition of a type. This problem exists on a broader scale as well. Typing in object-oriented languages is an active research domain [FM96, BPF97, DV98]. Current type systems are not yet mature enough for the construction of object-oriented programs in a flexible way. Therefore, it is harder to build frameworks in statically typed object-oriented languages than in dynamically typed object-oriented languages. The large amount of inevitable typecasts that are necessary when a typed framework is customised is a barometer for the typing problems that still need to be solved.

12.4.2 Runtime Statistical Typing

Brown uses runtime statistical typing [Bro96] in the context of reverse engineering design patterns [GHJV94] (also see the next section). Brown executes a Smalltalk program and interrupts it every few seconds with a high-priority process. This process enumerates all classes for which type information is required and uses the `allInstances` (meta-)message to determine the class of each instance variable of these instances. These classes are recorded in a type representation for each class.

The major disadvantage of his approach and dynamic (runtime-based) type inference in general is that the obtained types highly depend on the control flow of the target program. It is possible that some types are incomplete.
Brown acknowledges that his approach is not perfect. Therefore, he allows the reverse engineer to change the type of an instance variable on the fly. Such an approach requires that the reverse engineer has knowledge about the system he is reverse engineering, which is not always the case.

12.5 Reverse Engineering

12.5.1 Recovering Design Patterns

Brown performs design reverse engineering for automated design pattern detection in Smalltalk [Bro96]. His approach makes extensive use of the Smalltalk metalevel facilities to extract is-kind-of, has-a, and message flow diagrams. Based on the extracted diagrams, he is able to automatically detect the following design patterns: Composite and Decorator (these are closely related), Template Method (considered trivial by most programmers), and Chain of Responsibility. Brown's approach apparently does not handle the recovery of other design patterns.

12.5.2 Reflexion Models

Murphy proposes lightweight *structural summarisation* as an aid for more effectively assessing, planning and executing a software change task [Mur96]. Summarisation is a technique to deal with the large quantities of information. Summaries are overviews that help to comprehend the information without forcing the reader to examine all the information in detail. Murphy describes two techniques to apply summaries in the understanding of the structure of a software system: the software reflexion model technique and the lexical source model extraction technique.

The *software reflexion model technique* is used to compare a software engineer's mental model of the subject software with the model found in the source code.
The software engineer starts by selecting a high-level structural model of a collection of software entities with interactions (calling dependencies) between them. With an extraction tool, the engineer extracts structural information from the artefacts of the system: the *source model*. Extraction can be done statically or dynamically. The source model consists of relations between source components, such as relations between classes and message sends between them. The software engineer then defines a mapping between the source model and the high-level structural model. The mapping consists of mapping entries, each describing how a source model component maps to an entity in the high-level structural model. The employed mapping language can be extended in case it is not expressive enough to specify the entities in the source model at hand. Finally, a tool computes the *software reflexion model*. It provides the comparison between the source model and the high-level structural model. The reflexion model shows interactions between entities from the high-level structural model as solid, dashed, or dotted lines. Solid lines, called *convergences*, represent discovered interactions that were expected by the software engineer. Dashed lines, called *divergences*, are discovered interactions that were not expected, and dotted lines, called *absences*, represent expected interactions that were not found. Each line in the diagram indicates the discovered number of calls in the source code.
The engineer examines the reflexion model to assess whether it provides enough information to carry out the change task at hand. If the provided summary does not contain enough information for this assessment, the engineer can access the calls associated with

an interaction line in the diagram to obtain a more detailed view (i.e. the source code). If the reflexion model does not provide enough information for the given change task, it can be refined incrementally until it does. The engineer can refine the high-level structural model, use a better source model extraction tool, or enhance the mapping between the two models to compute a new reflexion model. Murphy claims that specifying the inputs and computing the software reflexion model has often been accomplished in about an hour. Murphy's technique scales well. It has been used on an artefact with more than a million lines of source code.

The *lexical source model extraction technique* is used to extract structural information for the source model, or for other reverse engineering activities, such as program visualisation. Murphy defines her own lexical specifications and she developed her own lexical analyser because current tools do not fit her needs. Current lexical analyser tools are too sensitive to the condition of the source code (written in language dialects or not yet in a form that can be compiled). Moreover, they cannot be used to scan other system artefacts than program source files, and the lexical specifications are too complex. Murphy's lexical source model extraction technique permits engineers to describe regular expressions with additional actions that must be executed when a desired pattern has been found in an artefact. These actions can be used to create an output of matches or to combine matched information into structural interactions. The software engineer starts with a coarse specification and refines it until it the lexical analyser produces the desired results.

The reflexion model technique is related to incremental recovery of collaboration contracts, in motivation (planning software changes) as well as in spirit. Both techniques are lightweight. They are developed to produce results rapidly, without heavy and time-consuming computations. Both techniques are incremental. Iteration and incremental refinement are important qualities. Both techniques are semi-automatic. Input from the software engineer is crucial. Extraction tools are used when appropriate, but the engineer decides whether the extracted information is useful.

A classification holding collaborating participants can be considered a reflexion model during incremental collaboration contract recovery. As long as the software engineer decides that a classification of participants is not yet complete, the current state of the classification is a model to reflect on. The comparison of the partial collaboration contract with the source code is easily made with the Classification Browser. When a participant is being browsed, it is presented as a view on a (partially flattened) class. The browser displays all methods of the class and highlights the methods that are classified in the participant. In the same vein, the interaction structure (through senders and implementers) can be explored in the scope of the classification, as well as in the scope of larger parts or the whole system. The engineer knows exactly what the difference between the partial collaboration contracts and the source code is.

12.6 Program Visualisation

Program visualisation is another technique to create models of software. An interesting approach to the visualisation of the interaction structure of an object-oriented program comes from De Pauw et al [PHKV93]. An instrumented version of a C++ program produces output that is transformed into graphical output.

Initially, when no messages are sent yet, the objects are placed near the edge of the window that displays the graphical output. Message sends are displayed as lines from one object to another. They remain on screen as long as the corresponding method is being executed. When objects interact a lot, they are displayed closer to each other. The result is that after a while clusters of objects appear on screen. Objects drift apart when the number of message sends between them decreases. Ultimately, non-interacting objects are displayed near the edge of the window again.

A software engineer watching the graphical output is able to assess the interaction patterns between objects. The clustering of objects is an important indication of the importance of an interaction pattern.

Program visualisation is a technique based on dynamic information, that is, information collected when the program is running. This contrasts this work, which is based on static information that can be extracted from the source code. The results differ accordingly: program visualisation produces event traces while recovery through software classification produces collaboration contracts.

Although the collection of information at runtime scales up, the visualisation of the results suffers from scaling problems. When many objects are involved in an interaction, the screen is quickly filled with an overwhelming amount of graphical data. The abundance of graphical output may be a seriously impediment to readability and interpretation of the results. Advanced zoom in/zoom out and clustering facilities may reduce the viewing problems considerably.

Chapter 13

Conclusion

The objective of this work has been architectural recovery in evolving object-oriented systems. Since the architecture of a software system comprises too many aspects to handle in a dissertation, the scope of this work has been narrowed to the recovery of components and their interaction structure. The roots of this research, being the research on reuse contracts, have driven further scope reduction. The result is that this work has three foci of attention. First, the recovery of collaboration contracts that describe the interaction structure of a set of classes (the components). Second, the recovery of reuse contracts that describe the evolution of collaboration contracts. Third, the recovery of architectural components that are larger than classes, such as software modules, software layers, and framework customisations. The latter focus is only concerned with the recovery of components, not their interaction structure. Therefore, the last focus of attention is to be considered the first step towards the recovery of collaboration contracts with architectural components as participants.

13.1 Summary

Instead of looking at recovery of collaboration contracts, reuse contracts and architectural components in isolation, and providing ad-hoc solutions for the three recovery problems, this research has focussed on finding a generic model to treat the three recovery problems under the same heading. This dissertation has introduced *software classification* as a very flexible means to recover architectural elements in an incremental way.

Software classification has two major aspects: the software classification *model* and the software classification *technique*. The software classification model has been conceived as a uniform structure to organise software entities. Although the model is very simple — classifications are containers of items and items can be classified in multiple classifications —, this work suggests that the model can take us very far. The openness of the model, achieved through the introduction of virtual classifications, opens the door to items that go beyond the traditional ones like classes and views on classes (the latter represented here by participants). Virtual classifications compute their items or retrieve them from external media. In this work, only items with an interface have been used, but the software classification model does not impose restrictions on items. Any kind of software entity is a candidate item.

When software classification is being performed, software classification strategies are applied to create new or update existing classifications. The choice of classification strategy depends on the goal of the classification activity. Although further research on classification strategies is required (see Section 13.3.5), some strategies have already been used successfully: manual classification, virtual classification, classification with advanced navigation tools, and automatic classification through method tagging.
Manual classification is the simplest form of classification: the software engineer puts items in classifications according to his wishes. Virtual classification is a classification strategy to draw software entities of the software development environment into the software classification model. In a Smalltalk environment, for instance, existing Smalltalk categories show up as classifications in which classes are virtually classified. Classification with advanced navigation tools is typically used to set up classifications of items with an interaction structure or dependency relationship. The recovery of collaboration contracts is based on this classification strategy. One automatic classification strategy has been investigated in-depth: classification based on method tagging. This classification strategy is based on mandatory information provided by the software engineer during forward engineering. The classifications are set up automatically based on the method tags provided. This automatic classification strategy produces promising results, which suggests that more research must be devoted to automatic classification strategies.

Five applications of software classification have been studied and discussed in detail: expressing multiple views on software, in particular on the software architecture, recovery of collaboration contracts, recovery of reuse contracts, recovery of large architectural components, and the management of changes.

Expressing multiple views on software. Expressing multiple views on software is an application of the software classification *model*. The application is based on the property that items can be classified in multiple classifications. Multiple classification of classes provides the software engineer with a powerful means to organise the classes of an object-oriented software system. Each classification can be used to express an abstraction that is otherwise hard to make tangible in a software development environment.

Besides multiple views on the software, considered as collections of classes, software classification provides a means to express multiple roles played by classes in collaboration with other classes. By observing that a collaboration contract can be considered a classification of participants that are views on classes, participants as items and collaboration contracts as classifications integrate well in the software classification model. When a class participates in several collaboration contracts, each collaboration contract expresses a different role played by that class. The software classification model thus not only supports multiple views on a set of classes; it also supports multiple views on the collaboration of a class with other classes.

Recovery of collaboration contracts. The recovery of collaboration contracts is an application of the software classification *model* and the software classification *technique*. It is an application of the *model* because collaboration contracts are represented by classifications of participants. Seen from the software classification model, a participant is a special kind of item (a view on a partially flattened class). Seen from the reuse contract model, a participant is an actor in a collaboration contract. It bears a partial interface of the corresponding (partially flattened) class, and it has an acquaintance clause. Methods in a participant's interface are annotated with an abstractness attribute and a specialisation interface. The participant item in the classification model is thus a straightforward representation of the concept of participant in the reuse contract model.

The recovery of collaboration contracts is also an application of the software classification *technique*. Collaboration contracts are recovered through *incremental classification*. Recovery starts with the creation of a classification that ultimately will represent a collaboration contract. Recovery proceeds with the classification of classes as participants and the classification in those participants of methods, acquaintance relationships and method invocations. Conceptually, the initial classification of classes passes through four stages, each of which is a more formal representation of the target collaboration contract. In the first stage, the classification is just a classification of classes. In the second stage, the classification holds participants with partial interfaces of those classes. In the third stage, the classification holds participants that are acquainted with each other. Finally in the fourth stage, the classification represents a collaboration contract, being a set of acquainted participants with an interaction structure. Each transition from one stage to the next involves classification activities. The transition from stage 1 to stage 2 first requires the classification of participants in a classification. After that, methods are classified in those participants. To get from stage 2 to stage 3, acquaintance relationships are classified in the participants. The transition from stage 3 to stage 4, method invocations are classified (this transition is actually done automatically based on the interfaces and the acquaintance relationships of the participants). Although participants are not really classifications, the term 'classification' is used nevertheless to refer to the act of putting methods, acquaintance relationships and method invocation into participants.

The proposed recovery process is not an automatic recovery process. The software engineer's input is required with regard to decisions for inclusion (classification) and exclusion of participants, methods and acquaintance relationships. The recovery of collaboration contracts relies on browsing activities alternated with classification activities. In practice, the conceptual stages are not as clear-cut. Since the recovery process relies on browsing, and browsing is typically done in a non-sequential manner, the transitions from one stage to another are strongly intertwined.

Recovery of reuse contracts. The recovery of reuse contracts is an indirect application of the software classification *model*. Actually, reuse contracts are not recovered by employing classifications. Rather, classification techniques are employed to set up an initial and a derived collaboration contract. The reuse contract between the two can be computed automatically.

The computation of the reuse contract boils down to the computation of the different basic reuser clauses, together with their associated contract types, that make up the change from the initial to the derived collaboration contract. The algorithm to determine the reuser clauses computes the differences between the interfaces, acquaintance clauses and specialisation clauses of the corresponding participants, and represents the differences by reuser clauses and contract types. The algorithm produces a reuse contract with a provider clause that contains the initial collaboration contract, and a combined reuser clause that includes all the basic reuser clauses, each with their contract type.

The integration of reuse contracts in the software classification model is based on the observation that a reuse contract can be considered as a classification with two elements, the initial and the derived collaboration contract.

Recovery of architectural components. The recovery of architectural components is an application of the software classification *model* and the software classification *technique*. It is an application of the *model* because classifications are used to gather information about the software system. It is an application of the *technique* because methods (and consequently classes) are classified into multiple classifications, based on a fixed algorithm.

The recovery of architectural components is based on tagging methods with classification information. A method tag includes information about the module, the framework customisation, etc. in which a method resides. Tagging requires little effort from the software developers. They have to provide the mandatory tagging information only once per development task (specification or bug fix). A batch process extracts the tags and uses their contents to classify the methods in classifications. The generated classifications include classifications for modules, framework customisations and architectural components. The generated classifications thus provide information that can be exploited by the software engineers and the project managers in further development activities.

Management of changes. The management of changes is also an application of the software classification *technique* that is based on method tagging. Besides information about the large architectural building blocks in which a class resides, the method tags also include a time stamp, a developer name, the development task, and a free-text field to log the reason why a change was made. This information is also processed and put into classifications. Classifications based on the development task help to assess why a set of methods has been changed. By cross-referencing the classifications, more information can be retrieved, such as the expertise of a developer based on the number of changes made to a given part of the software. The classifications are available in the Classification Browser, so that the software engineers and the code reviewer in particular can consult and browse these classifications. Since the classifications hold onto the changes that were made in the context of a specification or a bug report, the code reviewer can rapidly track changes based on the identification numbers of specifications and bug reports. Information that previously had to be retrieved from a request/defect tracking system is now available in the software development environment.

The software classification model is the foundation of the Classification Browser. This browser employs the software classification model as a means to organise classes, methods and other software entities, such as collaboration contracts, reuse contracts, and classifications. The browser currently supports three software classification techniques: manual classification, virtual classification, and classification with advanced navigation tools. The latter is supported by supplementing standard browsing features with support for in-place browsing of senders and implementers, and for browsing of acquaintance relationships.
Classifications are not only used for the organisation of the software; they are also used to define scopes in which browsing takes place. When browsing is restricted to the scope of a collaboration contract, the Classification Browser acts as collaboration contract browser.

The software classification model, the classification strategies, and the methods for recovery have been developed according to the industry-as-laboratory approach. In close co-operation with the software engineers of a software company, the software classification model, the recovery process, and the classification browser were conceived, designed, evaluated and implemented in an incremental and prototypical way. The subject software system, a continuous evolving complex software system for broadcast management, served as an excellent case study. It is representative of the complex evolving object-oriented software systems in use today.
The commitment of the software provider to this research has been crucial to conduct experiments on real world evolving object-oriented software, because the proposed classification strategy based on method tagging requires changes to the software development environment and the software development process. Without that commitment, parts of this research would not have been possible. After all, not many software providers are inclined to change their way of working for experimental research such as this.

The experiments conducted in the context of the subject software suggest that software classification, as a model, and as a technique, are valuable in software development. The fact that the recovered classifications can be consulted in the development environment is considered an asset by the software engineers that have used (and still use) the Classification Browser. The initial feedback from software engineers is very encouraging and

indicates that more and larger experiments should be conducted to assess the value of software classification. In particular, controlled experiments are required to assess the impact of the availability of collaboration contracts on the productivity of the software engineers during software evolution.
The automatic classification of the subject software system provides insight into the organisation of the software with respect to modules, framework customisations and specifications. It also spawns crucial insight in the evolution of the software that is very helpful for the code reviewer.

13.2 Contributions

The main contribution of this dissertation is the presentation of a generic model and strategies for incremental recovery of the architecture of evolving object-oriented systems. Expressing multiple views on software, the recovery of large architectural building blocks, collaboration contracts and reuse contracts, and the management of changes are applications of the software classification model and the software classification technique.

The software classification *model* provides simple concepts to organise large software systems and their evolution in manageable units (classifications). The software classification *technique* provides strategies to set up and recover those manageable units. The handful of classification strategies used in this dissertation — manual classification, virtual classification, classification with advanced navigation tools, and automatic classification through method tagging — suggest that software classification is a valid approach to reverse engineering of collaboration contracts and architectural components. This dissertation does not provide an answer to all questions raised, but it indicates that software classification has potential that must be investigated further.

Very important in this work is the integration of software classification in the software development environment and in the software development process. The results of recovery are tangible in the software development environment and they can be used in subsequent software development activities. Therefore, this work also makes the following contributions: the Classification Browser, the method tagging tool and method tag processor, the exploitation of different views on software, scope reduction as a browsing aid, and an algorithm to automatically compute the reuse contract between an initial and a derived collaboration contract.

Besides contributing to software evolution, software architecture, and reverse engineering, this work is also beneficial for the following research areas:

Reuse contracts. The lack of tool support has been acknowledged as a major problem to validate the claims made about the model and to carry out larger-scale experiments [Luc97]. This work addresses this problem by supplying methods and tools to set up collaboration contracts and reuse contracts. In addition, it presents the classification model as a means to organise software entities, and shows how collaboration contracts and reuse contracts fit in that model.

Object-oriented software development environments. The ideas behind the major artefact that results from this work, the Classification Browser, advances the state-of-the-art of object-oriented software development environments in three important aspects. First, having a flexible organisational structure to organise software entities in a user-defined way is a substantial aid to structure and comprehend the large and complex software in use today. Feedback from software engineers suggests that the availability of classifications increases productivity because classifications can be used as on-line software documentation. Second, the Classification Browser provides very powerful browsing capabilities. The focus on reuse, class collaboration and evolution is a great asset to a software engineer. Third, the Classification Browser is able to handle collaboration contracts, an important abstraction in object-orientation that is not tangible in conventional development tools.

Therefore, this work is useful for:

- Researchers who want to further enhance the reuse contract model, in theory as well as in practice.
- Software engineers who want to document a software system with classifications, collaboration contracts and reuse contracts. The Classification Browser supplements the directions for recovery given in this document.
- Practitioners who consider introducing classifications, collaboration contacts and reuse contracts in their development process. They gain an insight into the benefits of applying software classification and collaboration contracts, and are supplied with tools that give an idea of how software classification and collaboration contracts integrate in a software development environment and in the software development process.

13.3 Future Work

Many enhancements to the classification model, the classification browser, and the recovery process are conceivable. A handful is considered here.

13.3.1 Architectural Components as Participants

This dissertation focuses on classes as participants in collaboration contracts. However, other software entities have an interface too, and it is interesting to know how they collaborate.
For example, software layers are architectural components that collaborate. If the separation of the layers is not broken, the user interface layer only interacts with the domain model layer, and the domain model layer only interacts with the persistency layer (both

in one direction, except for call-backs). Collaboration contracts between the user interface layer and the domain model layer would be helpful to assess how the user interface layer uses and relies on the domain model layer. These collaboration contracts would provide insight into the use of domain model objects by application objects.
Another example is frameworks. Collaboration contracts with a framework and its customisations as participants would provide very interesting information about how a framework customisation relies on the framework and how both parts work together.

As shown in this work, the classification model provides a way to represent larger software entities. If such an entity is a grouping of whole classes, the entity can be represented by a classification that holds all those classes. What's missing for considering such entities as participants in a collaboration contract, is an interface. In the current state of the classification model, a classification has no interface. So in order to use classifications to represent large software entities that participate in a collaboration contract, a means must be devised for associating an interface with a classification. The same technique could be used as employed to consider classes (or objects) as participants. In the currect state, a participant is a view on one class: it lists (classifies) a selection of methods, acquaintance relationships and method invocations of one class. The concept of a participant could be extended to list (classify) a selection of methods, acquaintance relationships and method invocations of multiple classes. Research on this subject is in progress.

There is another interesting aspect of larger software entities as participants. When larger software entities consist of classes, it is interesting to know how the collaboration contracts for the large software entities relate to collaboration contracts for the classes that make up the large software entities. Further research is necessary to determine the relationships among the collaboration contracts at different levels of granularity.

13.3.2 Generic Collaboration Contracts

Experiments with collaboration contract recovery have shown that some collaboration contracts are the same except for participants names, acquaintance names, or method signatures. This observation led to the need for generic collaboration contracts in which participant names, acquaintance names, and method signatures are placeholders. An instantiation mechanism could then be used to fill in the names and the signatures.

Generic collaboration contracts may be useful to describe the 'Structure' and the 'Collaborations' sections of the design pattern form, because design patterns express the structure and the collaborations in an abstract (generic) manner.

For example, consider the collaboration contract at the top of Figure 13.1. It describes part of the Composite design pattern: the collaboration of the Component and the Composite participants. The design pattern lays down that when a Composite object receives a message `operation`, it responds to the message by sending the same message to all the Component objects that make up the Composite object. The Composite design pattern

describes the collaboration in a generic way. It does not state any specifics about the message. So the signature of the operation is to be considered a placeholder for the actual method signature used when the design pattern is applied.
The collaboration contract in the figure describes this collaboration. The `Component` object has a method called `operation`, which is abstract (note that it is written in italic). The `Composite` object has a method with the same name. The `Composite` object refers to each of its (sub)components by the acquaintance name `subComponent`. The invocation on the left-pointing arrow indicates that `operation` on `Composite` invokes `operation` on `Component`.

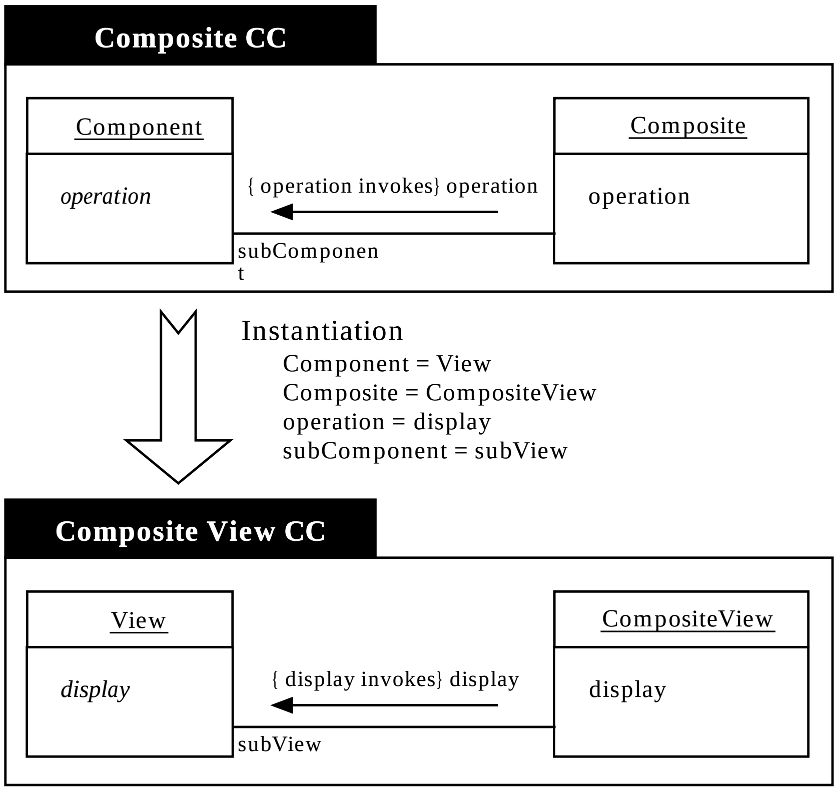


Figure 13.1: Instantiation of a generic collaboration contract

When the composite design pattern is applied, the generic names used in the design pattern are substituted for 'real' names. In the context of collaboration contracts, application of the design pattern implies that the generic collaboration contract is instantiated. The figure shows how instantiation could be achieved. All generic names are replaced by the real names. In the figure, the design pattern is applied in the design of views and composite views. The information provided next to the instantiation arrow includes a replacement for each generic name. Besides replacing `Component` by `View` and `Composite` by `CompositeView`, the generic name for message `operation` is replaced by `display` and the generic acquaintance name `subComponent` is replaced by `subView`, a more appropriate name in the context of views.

More research is necessary to define the role and the usefulness of generic collaboration contracts in general, and in the description of design patterns in particular.

13.3.3 Keeping Classifications In Sync

As traditional software documentation, classifications and collaboration contracts may become out of sync with the source code. Having them in the development environment is advantageous in that respect. Classifications and collaboration contracts can be checked for consistency by comparing them with the source code. The development environment can take inconsistencies into account and notify the problems, or display the inconsistent software entities in an eye-catching way. The Classification Browser already takes into account inconsistencies in classifications. When a classified class is not available in the Smalltalk image, the Classification Browser marks the class as unknown in the user interface.

Checking inconsistencies during development can be helpful in order to detect architectural drift early. The development environment is able to check whether a new version of a method still complies with the collaboration contracts in which it is involved. If it does not comply, either a collaboration contract has been broken, or either the collaboration contract needs revision. Whether such inconsistency checking is desirable, how and when it must be performed, and how inconsistencies should be presented to the software engineer, are questions that still need careful investigation.

13.3.4 Enforcing Collaboration Contracts

A question that rises frequently when people are introduced to collaboration contracts, and in particular to tools based on collaboration contracts, is whether collaboration contracts can be used to enforce object interactions that may not be broken. As already stated by Lucas [Luc97], collaboration contracts can be used to enforce a design. In the context of the Classification Browser, enforcing a collaboration contract can be done in a preventive and in a corrective way.

The preventive way of enforcement is based on preventing the software engineer to make changes to a class that would invalidate existing collaboration contracts of which it is known that they are not allowed to change. We will use the term 'stable collaboration contracts' to refer to collaboration contracts that are not intended to change. Since a collaboration contract does not include information about permitted changes (the reuse contract model assumes that anything may change, as long as the change is documented), the model needs to be extended to add such information. Stating that a stable collaboration contract may not be changed implies that the contract itself is not allowed to evolve, and that the collaboration contract is not allowed to be subject to given reuser clauses. For example, when a certain method invocation is mandatory, the collaboration contract may not be reused through a participant coarsening that removes the invocation. In general, the stable collaboration contract must be annotated with reuser clauses that express which changes are not allowed. More research is necessary to investigate what is the best way to model disallowed changes.

In order to enforce collaboration contracts, the development tool has to check whether the source code of the changed classes still conforms to the information included in the stable collaboration contracts. With software classification, these checks are easily performed. Since collaboration contracts are classifications, and since the classification repository maintains inverse classifications, a simple query returns all collaboration contracts a class is involved in. Based on the extra information mentioned earlier, the returned set can be filtered, so that only the stable collaboration contracts remain. The change can be accepted when the source code of the class still conforms to all these collaboration contracts. Otherwise, the software engineer must be informed that the intended change is not allowed.

While the preventive way to enforce collaboration contracts avoids that a disallowed change is made, the corrective way of enforcement, as the name suggests, is based on correcting or rejecting disallowed changes after they are made. The corrective approach to enforcement is a consequence of the observation that preventive approach does not work in practice, due to the fine granularity of changes. Since changes for a development task are made in a sequential manner, the source code is often in an inconsistent state during the execution of the development task. After execution of all required changes, however, the source code is expected to be in a consistent state again. Triggering the conformance checks each time a change is made may result in many reported problems that in fact are not real problems. They are only real problems if they persist after the development task has been performed. This observation indicates that conformance checks should be conducted after a development task, not after each change required for the task. The observation further suggests that a development task is a kind of transaction that can be rolled-back (as in database terminology) when the changes result in source code that does not conform to the stable collaboration contracts.

Without collaboration contracts and software classification, corrective enforcement is usually used in software development processes in which a code reviewer is present. Based on his knowledge of the software system, the code reviewer accepts or rejects changes made by the development team. When software classification and collaboration contracts are available in the development environment, the code reviewer's work can be partly automated. Conformance checking can also be conducted by members of the development team, before they submit the changes to the code reviewer. By doing so, they receive early feedback, and the net result is that the code reviewer will reject less disallowed changes.

The code reviewer is not only a consumer of the collaboration contracts; he is a producer of collaboration contracts as well. The code reviewer is in an ideal position to add new collaboration contracts to the system, and to review existing ones. When he receives changes to the software system, the changed classes may be covered by collaboration

contracts, or they may not yet have been documented. In the former case, the code reviewer can perform conformance checks. If the conformance checks fail, the code reviewer can decide to reject the changes, or to accept them after updating the relevant collaboration contracts. In the latter case, the code reviewer may decide to enter new collaboration contracts to the system, based on knowledge gained through investigation and review of the provided changes. Based on his knowledge of the software system, the code reviewer can also decide whether new or updated collaboration contracts should be stable collaboration contracts or not.
This way of working is an incremental approach to setting up collaboration contracts for a software system. The code reviewer relies on collaboration contracts that exist, while he relies on his knowledge when no collaboration contracts are available. In the latter case, the code reviewer is in the best position to extend the set of available collaboration contracts. By adding collaboration contracts, the code reviewer increases the number of collaboration contracts he can rely on during subsequent code reviews.

13.3.5 Expressing Architectural Constraints

This dissertation has restricted architectural recovery to the recovery of collaboration contracts and the recovery of architectural components. There are, however, other interesting aspects of the architectural that can be subject to recovery. One aspect has already been discussed in Section 13.3.1: recovery of the interaction structure of architectural components. Here expressing architectural constraints is discussed.

Architectural constraints that are concerned with the interaction structure of the architectural components are candidates for expression with collaboration contracts and classifications. For example, consider a layered architecture with a user interface layer, a domain model layer, and a persistency layer. The constraints imposed by the layered architecture state that classes in the user interface layer may invoke methods of classes in the domain model layer, and that classes in the domain model layer may invoke methods of classes in the persistency layer, but not the other way around. A possible solution for expressing this architectural constraints is based on other suggested enhancements of the reuse contract model and the classification model as discussed earlier in this future work section: architectural components as participants in collaboration contracts, generic collaboration contracts, and enforcing collaboration contracts.

When collaboration contracts with architectural components as participants are available, the collaboration contract to express the collaboration of the user interface layer and the domain model layer can be described as depicted in Figure 13.2. The collaboration contract is a generic contract that expresses that the user interface layer sends messages to the domain model layer. Any collaboration contract between a class of the user interface layer and a class of the domain model layer should be an instantiation of this generic collaboration contract.
As mentioned in Section 13.3.4, the collaboration contract model is not equipped with a means to describe that no messages may be sent from the domain model layer to the

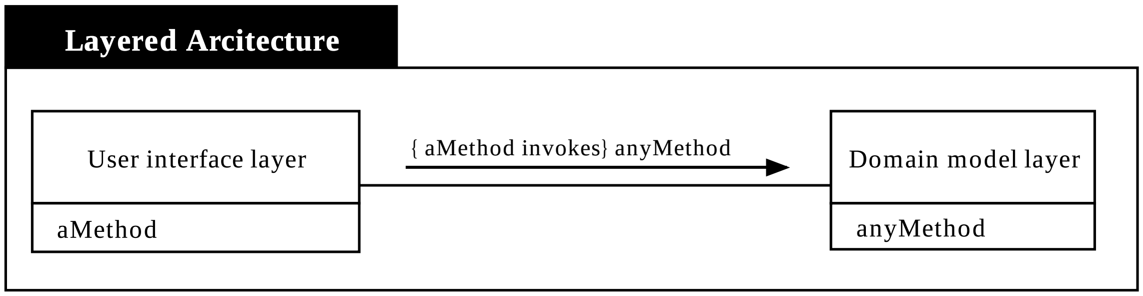


Figure 13.2: Collaboration contract for layered architecture

user interface layer, but it is conceivable to supply negative information in the form of reuser clauses that are not allowed to be applied on the collaboration contract. In this case, the constraint "no messages should be sent from the domain model layer to the user interface layer" can be expressed by a participant refinement reuser clause that includes such message send. Figure 13.3 shows this reuser clause. When interpreted as negative information, it expresses that `someMethod` on the domain model layer participant is not allowed to invoke `badMethod` on the user interface layer participant. Note that this reuser clause is a generic version of the participant refinement reuser clause normally used in reuse contracts.

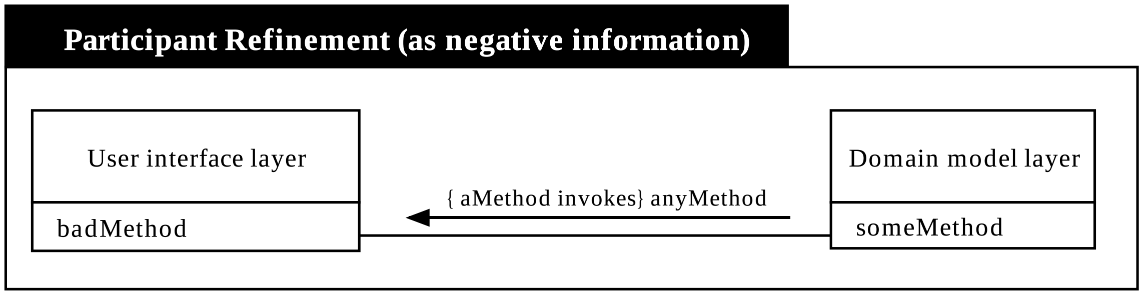

Participant Refinement (as negative information)
User interface layer
badMethod
{ aMethod invokes} anyMethod
Domain model layer
someMethod

Figure 13.3: Reuser clause to be interpreted as negative information

The conclusion is thus that enhancements of, and additions to, the collaboration contract model may produce a version of collaboration contracts that can be used to express architectural constraints that are concerned with the interaction structure, but more research on the subject is necessary to determine the best approach to express constraints.

13.3.6 Software Classification Strategies

In this work, four software classification strategies have been presented: manual classification, virtual classification, classification with advanced navigation tools, and automatic classification through method tagging. Further research is necessary to identify other classification strategies and to evaluate their usefulness in software development.

The exploitation of virtual classifications promises to produce interesting results in the context of automatic classification strategies. When virtual classifications are connected with a logic metaprogramming language, they can be used to automatically classify software entities according to a given logic query. SOUL [Wuy98] is a candidate for integration in the Classification Browser. SOUL (Smalltalk Open Unification Language) is a logic language embedded in Smalltalk. Among others, it has been used to detect design patterns. Associating a new kind of virtual classification with SOUL is enough to associate a logic meta-programming language in the Classification Browser. Research on this topic is currently going on.

13.3.7 Classifications in Other Development Tools

For now, classifications are only used in the Classification Browser. For a full integration in the software development environment, classifications should be used in other development tools as well. Further research is necessary to assess the usefulness of classifications in other development tools.

One development tool in which the application of classifications seems useful is the debugger. Classifications can be used as filters that restrict the method invocations that are displayed on the invocation stack of the debugger. In that way, stepping through method invocations can be restricted to methods of classes that are included in a classification, or methods that are included in a collaboration contract. Like reducing the browsing scope in the Classification Browser, the software engineer would then be able to reduce the execution scope in the debugger. Similar restriction, although imposed by the system, is found in LearningWorks [GAL97], in which the debugger only shows method invocations on the stack that are part of a learning book, instead of all messages, including system messages, that are necessary to evaluate messages sent from within a learning book.

13.3.8 Classifications in non-Smalltalk Environments

This work has a bias towards Smalltalk. There are two reasons for that bias. First, Smalltalk is still the most flexible language and environment to prototype and build applications. Second, the context in which this research has been carried out and in which it has been validated uses Smalltalk as development platform.

This work is also valuable for other languages and environments, however. Two differences between Smalltalk and other popular object-oriented languages make it worthwhile to investigate the integration of classifications and collaboration contracts in development environments for other languages.
First, Smalltalk keeps all classes and methods in one image, while other languages, such as Java and C++, are file-based. This may strongly affect the performance of the navigation in the Classification Browser and the determination of acquaintance classes. On the other

hand, modern development environments for Java and C++ feature browsers with senders and implementers facilities as in Smalltalk browsers. Therefore, porting the Classification Browser to such environments should be possible.
Second, typing may have an effect on the determination of acquaintance classes. The algorithm for computing acquaintance classes, discussed in Section 7.10, can be used for typed languages, but the presence of typing information may have an influence on the algorithm. The use of Java interfaces in Java programs is interesting in that respect. More research is necessary to shed more light on these issues.

The recovery of collaboration contracts for Java is currently going on. The development of a Classification Browser for Java is planned.

Appendix A

Reuse Contracts

This work uses reuse contracts as defined by Lucas [Luc97], albeit in a slightly different form. In her dissertation, Carine Lucas introduced reuse contracts for arbitrary components that invoke each other's operations (methods). Later in that document, she adapted and extended the basic reuse contract model towards reuse contracts for the UML [BRJ97]. The adaptations and extensions concern classes, inheritance, super sends, and abstract methods. Since this document is concerned with object-oriented development, it uses those definitions. This chapter recapitulates all relevant definitions, so that it is clear what the foundation is on which this work elaborates, and for reference in this document.

Each definition below is an aggregation of several definitions taken from Carine Lucas' dissertation. Detailed explanations of these definitions are not included here. Refer to her dissertation for detailed discussions. One major difference between Lucas' work and this work is the change in terminology with respect to reuse contracts. The term 'reuse contract' in Lucas' work actually refers to a collaboration contract, as used here.

A.1 The Reuse Contract Model

The reuse contract model was conceived as structured documentation to support the evolution of reusable components. Its conception was driven by the observation that assumptions about the co-operation between software components remain implicit and that making them explicit is essential for change propagation and impact analysis.

In early work on reuse contracts [SLMD96], the term 'reuse contract' referred to an explicit contract between a provider and a reuser. The provider supplies reuse information about software components, and reusers state how these are actually reused. The notion of a contract was used as in the real world: two parties, called provider and reuser, make a contract to stipulate formally how reuse is achieved. The terms and conditions of the contract state that the provider has to supply all necessary information so that the software components that are the subject of the contract can be reused properly. This means that assumptions about, and dependencies between the software components be known. The terms and conditions of the contract also state that the reuser has to supply information about how the provided software components have been reused.

In the original definition of a (multi-component) reuse contract [Luc97], however, a reuse contract does not include the notion of a provider and a reuser. It only describes a collaboration between a number of components. Therefore, later work introduced the term 'collaboration contract' to refer to a formal description of a collaboration between components, and redefined the term 'reuse contract' to assign it the original meaning, as given above. Furthermore, the term 'reuse modifier', describing an adaptation of an old-style reuse contract, is obsolete and is now found as the combination of a contract type and a reuser clause.

So in order to avoid confusion due to the change in terminology: the term 'reuse contract' found in Lucas' work [Luc97] has been renamed to 'collaboration contract', and the term 'reuse contract' is now used to refer to a contract between a provider and a reuser. The relation between the two will be clear from the definitions given below: the provider (clause) of a reuse contract is a collaboration contract.

A.2 Collaboration Contracts

A collaboration contract formally describes a collaboration between participants. A participant can be any software entity that has an interface and that is able to invoke operations (methods) of other participants. Examples are classes, parts of classes, modules, components (in the sense of component-oriented programming), subsystems, software layers, etc. Each participant has a name, an interface and an acquaintance clause. The acquaintance clause states which participants a participant is acquainted with (possibly itself), the so-called acquaintance relationships. A collaboration contract has a name for reference and it should be well-formed, so that it does not reference participants or methods that do not exist in the contract.

A.2.1 Collaboration Contract

Definition 8 (Collaboration Contract)

A **collaboration contract** consists of:

1. a name;
2. a set of participants, each with a name that is unique within the collaboration contract.

Definition 9 (Participant)

A **participant** consists of:

1. a name;
2. an acquaintance clause;
3. an interface.

Definition 10 (Acquaintance clause)

An **acquaintance clause** is a set of acquaintance relationships $a.p$, associating acquaintance name a with a participant name p.

Definition 11 (Interface)

An **interface** is a set of methods each consisting of

1. a method signature that is unique within this interface;
2. an annotation *abstract* or *concrete*;
3. a specialisation clause.

Definition 12 (Specialisation clause)

A **specialisation clause** is a set of method invocations $a.m$, where a is an acquaintance name or the keyword *self* and m is a method signature.

The set of participants in a collaboration contract together with the acquaintance relationships between them is called the **context** of a collaboration contract.
When the acquaintance clause of a participant p contains $a.q$, we say that a on p **refers to** q or simply that p **refers to** q.

A.2.2 Well-Formedness

Definition 13 (Well-formedness of a collaboration contract)

A collaboration contract CC is **well-formed** if for each participant p in CC the following conditions hold:

1. for each acquaintance relationship $a.q$ in the acquaintance clause of p: a participant with name q exists in CC;
2. for each method invocation $a.m$ in a specialisation clause in p:
   (a) a is an acquaintance name in the acquaintance clause of p;
   (b) m is the signature of a method in the interface of the participant a refers to.

A.3 Reuse Contracts

A reuse contract is a contract between a provider and a reuser. The contract has two clauses, the provider clause and the reuser clause, and a contract type. The provider clause describes what is provided. In multi-component reuse contracts, the provider clause holds a collaboration contract that states how components collaborate to achieve some behaviour. The reuser clause describes how the provided components are reused. It can be considered as an amendment to the provider clause. The content of a reuser clause depends on the type of the contract. Several contract types are defined and each has its own form of reuser clause. Reuse contracts also have a name for reference.

A.3.1 Reuse Contract

Definition 14 (Reuse contract)

A **reuse contract** consists of:

1. a name;
2. a provider clause, which is a collaboration contract;
3. a contract type;
4. a reuser clause.

A.3.2 Reuse Contract Type

Definition 15 (Contract type)

A **contract type** is an annotation that describes the relationship between a provider clause and a reuser clause.

There are 11 basic contract types, as shown in Table A.1. The contract types correspond to the ways collaborations are adapted by a software engineer. The types can be divided into two groups: the participant types and the context types. The participant types indicate adaptation of the description of participants in a collaboration contract, while the context types indicate adaptation of the collaboration contract's context (the acquaintance relationships together with all participants as a whole).

A.4 Reuser Clauses

Each contract type has its associated reuser clause. Each reuser clause describes how a collaboration contract is adapted. In correspondence with the division of contract types, reuser clauses are also divided into two groups: the participant reuser clauses and the context reuser clauses. The former clauses describe changes to interfaces of participants and specialisation clauses of methods. The latter clauses describe changes to the context

Basic Contract Type	Meaning
Participant extension	Adding new methods to the interface of a participant
Participant cancellation	Removing methods from the interface of a participant
Participant refinement	Adding extra method invocations to a specialisation clause, while keeping the original method invocations
Participant coarsening	Removing method invocations from a specialisation clause
Participant specialisation	Adding extra method invocations to a specialisation clause, while referring to the original method invocations with 'super'
Participant concretisation	Making an abstract method concrete
Participant abstraction	Making a concrete method abstract
Context extension	Adding new participants
Context cancellation	Removing participants
Context refinement	Adding new acquaintance relationships
Context coarsening	Removing acquaintance relationships

Table A.1: Basic contract types

of a collaboration contract.

The changes described by a reuser clause can be applied to a collaboration contract in a provider clause to obtain a new collaboration contract that describes the adapted collaboration contract. Therefore, the following sections each consist of three definitions. The first defines a particular reuser clause. The second states when application of that reuser clause on a collaboration contract is allowed. The third defines the result collaboration contract after application of the reuser clause. The definitions ensure that the collaboration contract that results from applying a reuser clause on a well-formed collaboration contract is again well-formed.
The first definition is essential to integrate the reuser clause concept in a development environment. The first and the third definition together can be used inversely to extract a reuser clause by 'subtracting' two collaboration contracts, one being the original and the other being an adapted version of the original. These definitions will prove to be crucial in reverse engineering of reuse contracts, as discussed in Section 9.4.

A.5 Participant Reuser Clauses

Participant reuser clauses describe how participants in a collaboration contract are adapted. There are two kinds of participant adaptation: adaptation of participant interfaces and adaptation of specialisation clauses. The former involves addition and removal of methods, and alteration of a method's abstractness attribute. The latter involves addition and removal of method invocations.

There are seven participant reuser clauses: participant extension, cancellation, refinement, coarsening, concretisation, abstraction, and specialisation. Participant extension, refinement and concretisation are known as the design preserving participant reuser clauses, because they involve addition. Cancellation, coarsening and abstraction are their respective inverse reuser clauses. These are known as the design breaching participant reuser clauses, because they involve removal. The odd one out is participant specialisation. It is in fact a special kind of refinement to describe super sends in specialisation clauses. Its inverse reuser clause is also participant coarsening.

A.5.1 Participant Extension

A participant extension reuser clause describes how the interfaces of a set of participants are extended.

Definition 16 (Participant extension reuser clause)

A **participant extension reuser clause** is a reuser clause which is a set of pairs (p, int) each consisting of a participant name p and an interface int.

Definition 17 (Participant extendible)

A collaboration contract CC is **participant extendible** by a participant extension reuser clause C_{pe} if for each pair (p, int) in C_{pe}:

1. p is a participant name in CC;
2. no method signature in int appears in the interface of participant p in CC;
3. for each method invocation $a.m$ in a specialisation clause in int:
   (a) a is an acquaintance name in the acquaintance clause of p in CC;
   (b) if a on p refers to q then m is a method in the interface of q in C_{pe}.

This definition puts restrictions on a participant extension reuser clause. It can only add methods to existing participants (clause 1) and an added method cannot yet be present in the interface of the target participant (clause 2). Besides these obvious conditions, a participant extension reuser clause must be self-contained. This means that the specialisation clause of an added method is not allowed to refer an unknown acquaintance name (clause 3a). And it also means that method invocations in a specialisation clause can only refer methods that exist in the interface of the target acquaintance (clause 3b).

Definition 18 (Participant extension)

If a collaboration contract CC is participant extendible by a reuser clause C_{pe}, then the collaboration contract CC_{pe} is the participant extension of CC by C_{pe}, where:

1. CC_{pe} contains all participants of CC that are not mentioned in C_{pe};
2. for each (p, *int*) in C_{pe}: CC_{pe} contains a participant with the same name and acquaintance clause as p in CC and that contains all methods of p, plus *int*.

This definition states what the result is of applying a participant extension reuser clause on a collaboration contract. The resulting collaboration contract holds all participants of the original collaboration contract. Participants not referred to in the reuser clause are left untouched (clause 1). The other participants have an extended interface (clause 2).

A.5.2 Participant Cancellation

A participant cancellation reuser clause describes how the interfaces of a set of participants are reduced. Participant cancellation is thus the inverse operation of participant extension.

Definition 19 (Participant cancellation reuser clause)

A **participant cancellation reuser clause** is a reuser clause which is a set of pairs (p, *int*) each consisting of a participant name p and an interface *int*.

Definition 20 (Participant cancellable)

A collaboration contract CC is **participant cancellable** by a participant cancellation reuser clause C_{pc} if for each pair (p, *int*) in C_{pc}:

1. p is a participant name in CC and each method in *int* is identical to a method in this participant in CC;
2. for all methods m, n and for all participants q in CC, such that m on q invokes n on p: if n is an element of *int*, then m appears associated with q in C_{pc}.

This definition states when participant cancellation reuser clauses are applicable. They should only mention methods that exist in the target collaboration contract (clause 1). In addition, methods can only be removed if they are not referred to in any specialisation clause in the collaboration contract on which the reuse clause is applied (clause 2). The latter condition ensures well-formedness of the resulting collaboration contract.

Definition 21 (Participant cancellation)

If a collaboration contract CC is participant cancellable by a reuser clause C_{pc}, then the collaboration contract CC_{pc} is the **participant cancellation** of CC by C_{pc}, where:

1. CC_{pc} contains all participants of CC that are not named in C_{pc};
2. for each (p, *int*) in C_{pc}: CC_{pc} contains a participant with the same name and acquaintance clause as p in CC and that contains all methods of p, except for those in *int*.

The result of applying a participant cancellation reuser clause on an original collaboration contract is a collaboration contract with the same participants. Participants not referred to in the reuser clause are left untouched (clause 1). The other participants have reduced interfaces, according to the removal information found in the reuser clause (clause 2).

A.5.3 Participant Refinement

A participant refinement reuser clause describes additions of method invocations to specialisation clauses of methods.

Definition 22 (Participant refinement reuser clause)

A **participant refinement reuser clause** is a reuser clause containing pairs (p, *extint*) each consisting of a participant name p and an extended interface *extint*. An **extended interface** is a set of methods, each consisting of a method signature and two disjoint specialisation clauses. The first repeats the specialisation clause of the base contract, while the second describes the method invocations that need to be added.

Definition 23 (Participant refinable)

A collaboration contract CC is **participant refinable** by a participant refinement reuser clause C_{pr} if for each pair (p, *extint*):

1. p is a participant name in CC;
2. for each method signature m in *extint*: m appears in participant p in CC and m's first specialisation clause in *extint* is identical to the specialisation clause of m in p in CC;
3. for each method invocation $a.m$ in a second specialisation clause in *extint*:
   (a) a is an acquaintance name in the acquaintance clause of p in CC;
   (b) m is a method in the interface of the participant referred to by a in p in CC.

A participant refinement reuser clause is applicable on a collaboration contract only if several conditions are met. Only specialisation clauses of existing methods can be augmented

(clause 1), and these methods should have the same specialisation clause as stated in the extended interface (clause 2). Invocations mentioned in the reuser clause should refer to existing acquaintances of the target participant (clause 3a), and should refer to existing methods on the acquainted participants (clause 3b).
Clause 2 actually says that a participant refinement reuser clause is context sensitive. It can only be applied on collaboration contracts that hold methods with an expected specialisation clause. A participant specialisation reuser clause (see Section A.5.5) is the refinement's context insensitive counterpart.

Definition 24 (Participant refinement)

If a collaboration contract CC is participant refinable by a reuser clause C_{pr} then the collaboration contract CC_{pr} is the **participant refinement** of CC by C_{pr}, where:

1. CC_{pr} contains all participants of CC that are not mentioned in C_{pr};
2. for each (p, *extint*) in C_{pr}: CC_{pr} contains a participant
   (a) with name p and the same acquaintance clause as p in CC;
   (b) that contains all methods of p in CC not mentioned in *extint*;
   (c) that contains all methods in *extint* with as specialisation clause the union of their two specialisation clauses in *extint*.

The application of a participant refinement reuser clause produces a new collaboration contract that holds all participants of the original collaboration contract. All aspects of the participants are left untouched (clause 1, 2a and 2b), except for the specialisation clauses of the methods mentioned in the reuser clause, which are augmented with the method invocations found in the reuser clause (clause 2c).

A.5.4 Participant Coarsening

A participant coarsening reuser clause describes removals of method invocations from specialisation clauses of methods. Participant coarsening is thus the inverse operation of participant refinement.

Definition 25 (Participant coarsening reuser clause)

A **participant coarsening reuser clause** is a reuser clause containing pairs (p, *extint*) each consisting of a participant name p and an extended interface *extint*. The first specialisation clause of the extended interface denotes which invocations are retained, while the second denote the invocations that are removed.

Definition 26 (Participant coarsenable)

A collaboration contract CC is **participant coarsenable** by a participant coarsening reuser clause C_{pc} if for each pair (p, *extint*):

1. p is a participant name in CC;
2. for each method signature m in *extint*:
    (a) m appears in participant p in CC;
    (b) the union of m's specialisation clauses in *extint* is identical to the specialisation clause of m in CC.

A participant coarsening reuser clause is applicable on a collaboration contract only if it mentions existing participants (clause 1) and existing invocations in specialisation clauses of methods in interfaces of those participants (clause 2a). Moreover, a method's specialisation clause in the target collaboration contract should be equal to the union of the specialisation clauses found in the extended interface (clause 2b). The last condition makes a participant coarsening reuser clause context sensitive. It can only be applied on collaboration contracts that hold methods with an expected specialisation clause.

Definition 27 (Participant coarsening)

If a collaboration contract CC is participant coarsenable by a reuser clause C_{pc} then the collaboration contract CC_{pc} is the **participant coarsening** of CC by C_{pc} where:

1. CC_{pc} contains all participants of CC that are not mentioned in C_{pc};
2. for each pair (p, *extint*) mentioned in C_{pc}: CC_{pc} contains a participant
    (a) with name p and the same acquaintance clause as p in CC;
    (b) that contains all methods of p not mentioned in *extint*;
    (c) that contains all methods of *extint* with as specialisation clause the first of the specialisation clauses in *extint*.

The application of a participant coarsening reuser clause produces a new collaboration contract that holds all participants of the original collaboration contract. All aspects of the participants are left untouched (clause 1, 2a and 2b), except for the specialisation clauses of the methods mentioned in the reuser clause, which are reduced according to the specialisation clauses found in the reuser clause (clause 2c).

A.5.5 Participant Specialisation

A participant specialisation reuser clause is a special form of a participant refinement reuser clause. In contrast to a participant refinement reuser clause, a participant specialisation reuser clause is context insensitive. It extends the specialisation clause of methods regardless of the specialisation clause of the corresponding methods in a target collaboration contract.

Definition 28 (Participant specialisation reuser clause)

A **participant specialisation reuser clause** is a reuser clause containing pairs (p, int) each consisting of a participant name p and an interface int.

Definition 29 (Participant specialisable)

A collaboration contract CC is **participant specialisable** by a participant specialisation reuser clause C_{sp} if for each pair (p, int) in C_{sp}:

1. p is a participant name in CC;
2. for each method signature m in int:
   (a) m appears in participant p in CC;
   (b) m's specialisation clause is disjoint from the specialisation clause of m in p in CC;
3. for each method invocation $a.m$ in a specialisation clause in int:
   (a) a is an acquaintance name in the acquaintance clause of p;
   (b) m is the name of a method in the interface of the participant a refers to.

A participant specialisation reuser clause is applicable on a collaboration contract only if several conditions are met. Only specialisation clauses of existing methods can be augmented (clause 1 and 2a). The specialisation clauses of these methods and the corresponding specialisation clause in the collaboration contract should not overlap (clause 2b). Invocations mentioned in the reuser clause should refer to existing acquaintances of the target participant (clause 3a), and should refer to existing methods on the acquainted participants (clause 3b).

Note the difference with a participant refinement reuser clause: a participant specialisation reuser clause makes no assumptions about the collaboration contract on which it may be applied.

Definition 30 (Participant specialisation)

If a collaboration contract CC is participant specialisable by a reuser clause C_{sp}, then the collaboration contract CC_{sp} is the **participant specialisation** of CC by C_{sp}, where:

1. CC_{sp} contains all participants of CC that are not mentioned in C_{sp};
2. for each (p, *int*) in C_{sp}: CC_{sp} contains a participant
   (a) with name p and the same acquaintance clause as p in *int*;
   (b) that contains all methods of p in CC not mentioned in *int*;
   (c) that contains all methods of *int* with as specialisation clause the union of the specialisation clause of this method in CC and the specialisation clause of this method in *int*.

The application of a participant specialisation reuser clause produces a new collaboration contract that holds all participants of the original collaboration contract. All aspects of the participants are left untouched (clause 1, 2a and 2b), except for the specialisation clauses of the methods mentioned in the reuser clause, which are augmented with the method invocations found in the reuser clause (clause 2c).

A.5.6 Participant Concretisation

A participant concretisation reuser clause describes which abstract methods are made concrete.

Definition 31 (Participant concretisation reuser clause)

A **participant concretisation reuser clause** is a reuser clause containing pairs (p, *int*) each consisting of a participant name p and an interface *int*, in which all methods have the annotation *concrete*.

Definition 32 (Participant concretisable)

A collaboration contract CC is **participant concretisable** by a participant concretisation reuser clause C_{pc} if for each pair (p, *int*):

1. p is a participant name in CC;
2. for each method m in *int*:
   (a) m has the same name as an abstract method in p in CC;
   (b) m has the same specialisation clause as the corresponding method in p in CC.

A participant concretisation reuser clause can only be applied on a collaboration contract if it refers to existing abstract methods (clause 1 and 2a), and if it does not change the specialisation clauses of the abstract methods (clause 2b).

Definition 33 (Participant concretisation)

If a collaboration contract CC is participant concretisable by a reuser clause C_{pc}, then the collaboration contract CC_{pc} is the **participant concretisation** of CC by C_{pc}, where:

1. CC_{pc} contains all participants of CC that are not mentioned in C_{pc};
2. for each pair (p, int) in C_{pc}: CC_{pc} contains a participant
   (a) with name p and the same acquaintance clause as p in CC;
   (b) that contains all methods of p in CC not mentioned in int;
   (c) that contains all methods of int.

The result of applying a participant concretisation reuser clause on a collaboration contract is a new collaboration contract. It holds the same participants as the original (clause 1, 2a and 2b), but all methods mentioned by the reuser clause have the annotation 'concrete' (clause 2c and Definition 31).

A.5.7 Participant Abstraction

A participant abstraction reuser clause describes which abstract methods are made abstract.

Definition 34 (Participant abstraction reuser clause)

A **participant abstraction reuser clause** is a reuser clause containing pairs (p, int) each consisting of a participant name p and an interface int, in which all methods have the annotation *abstract*.

Definition 35 (Participant abstractable)

A collaboration contract CC is **participant abstractable** by a participant abstraction reuser clause C_{pc} if for each pair (p, int):

1. p is a participant name in CC;
2. for each method m in int:
   (a) m has the same name as a concrete method in p in CC;
   (b) m has the same specialisation clause as the corresponding method in p in CC.

A participant abstraction reuser clause can only be applied on a collaboration contract if it refers to existing concrete methods (clause 1 and 2a), and if it does not change the specialisation clauses of the concrete methods (clause 2b).

Definition 36 (Participant abstraction)

If a collaboration contract CC is participant abstractable by a reuser clause C_{pc}, then the collaboration contract CC_{pc} is the **participant abstraction** of CC by C_{pc}, where:

1. CC_{pc} contains all participants of CC that are not mentioned in C_{pc};
2. for each pair (p, int) in C_{pc}: CC_{pc} contains a participant
   (a) with name p and the same acquaintance clause as p in CC;
   (b) that contains all methods of p in CC not mentioned in int;
   (c) that contains all methods of int.

The result of applying a participant concretisation reuser clause on a collaboration contract is a new collaboration contract. It holds the same participants as the original (clause 1, 2a and 2b), but all methods mentioned by the reuser clause have the annotation 'abstract' (clause 2c and Definition 34).

A.6 Context Reuser Clauses

Context reuser clauses describe the adaptation of the context of a collaboration contract. There are two kinds of context adaptation: changes in the number of participants and changes to acquaintance clauses. The former involves addition and removal of participants as a whole. The latter involves addition and removal of acquaintance relationships.

There are four context reuser clauses: context extension, cancellation, refinement, and coarsening. Context extension and context refinement are the design preserving context reuser clauses. Context cancellation and context coarsening are their respective inverse reuser clauses. They are the design breaching context reuser clauses.

A.6.1 Context Extension

A context extension reuser clause holds participants that should be added to a collaboration contract.

Definition 37 (Context extension reuser clause)

A **context extension reuser clause** is a reuser clause which is a well-formed collaboration contract.

Definition 38 (Context extendible)

A collaboration contract CC is **context extendible** by a context extension reuser clause C_{ce} if for each participant p in C_{ce}:

- p's name is different from all participant names in CC.

This definition states that a context extension reuser clause and the collaboration contract on which it is applied should have disjoint sets of participants.

Definition 39 (Context extension)

If a collaboration contract CC is context extendible by a context extension reuser clause C_{ce}, then the collaboration contract CC_{ce} is the **context extension** of CC by C_{ce} where:

- CC_{ce} contains all participants of CC and all participants of CC_{ce}.

The result of applying a context extension reuser clause on a collaboration contract is a new collaboration contract that is the 'sum' of the two: the result holds all participants of the original and all participants found in the reuser clause.

A.6.2 Context Cancellation

A context cancellation reuser clause lists all participants that should be removed from a collaboration contract.

Definition 40 (Context cancellation reuser clause)

A **context cancellation reuser clause** is a reuser clause which is a well-formed collaboration contract.

Definition 41 (Context cancellable)

A collaboration contract CC is **context cancellable** by a context cancellation reuser clause C_{cc} if for each participant p in C_{cc}:

1. p is identical to a participant in CC;
2. p does not appear in the acquaintance clause of a participant in CC that is not in C_{cc}.

A context cancellation reuser clause can be applied on a collaboration contract if it lists participants that are equal to participants found in the collaboration contract (clause 1) and if the participants are not acquaintances of the remaining participants (clause 2).

Definition 42 (Context cancellation)

If a collaboration contract CC is context cancellable by a reuser clause C_{cc} then the collaboration contract CC_{cc} is the **context cancellation** of CC by C_{cc} where:

- CC_{cc} contains all participants of CC, except for those named in C_{cc}.

The application of a context cancellation reuser clause on a collaboration contract is thus a kind of subtraction of participants.

A.6.3 Context Refinement

A context refinement reuser clause lists the acquaintance relationships that should be added to acquaintance clauses of participants in a collaboration contract.

Definition 43 (Context refinement reuser clause)

A **context refinement reuser clause** is a reuser clause containing triples (p, *acq1*, *acq2*) each consisting of a participant name p and two disjoint acquaintance clauses.

Definition 44 (Context refinable)

A collaboration contract CC is **context refinable** by a context refinement reuser clause C_{cr} if for each triple (p, *acq1*, *acq2*):

1. p is a participant name in CC;
2. *acq1* is identical to the acquaintance clause of p in CC;
3. *acq2* contains acquaintance relationships $a.q$, where a is different from all acquaintance names in *acq1* and q is a participant name in CC.

A context refinement reuser clause is applicable on a collaboration contract only if satisfies three conditions. It should only refer existing participants (clause 1) and existing acquaintance clauses (clause 2). It should only add acquaintance relationships using new acquaintance names and referring to existing participants (clause 3).

Definition 45 (Context refinement)

If a collaboration contract CC is context refinable by a reuser clause C_{cr} then the collaboration contract CC_{cr} is the context refinement of CC by C_{cr}, where:

1. CC_{cr} contains all participants of CC that are not mentioned in C_{cr};
2. for each triple (p, *acq1*, *acq2*) in C_{cr}: CC_{cr} contains a participant with the same name and interface as p in CC and the union of *acq1* and *acq2* as acquaintance clause.

The result of applying a context refinement reuser clause on a collaboration contract is a new collaboration contract with the same participants. Participants not mentioned in the reuser clause are untouched (clause 1). The others have extended acquaintance clauses (clause 2).

A.6.4 Context Coarsening

A context coarsening reuser clause lists the acquaintance relationships that should be removed from acquaintance clauses of participants in a collaboration contract.

Definition 46 (Context coarsening reuser clause)

A **context coarsening reuser clause** is a reuser clause containing triples (*p*, *acq1*, *acq2*) each consisting of a participant name p and two disjoint acquaintance clauses.

Definition 47 (Context coarsenable)

A collaboration contract CC is **context coarsenable** by a context coarsening reuser clause C_{cc} if for each triple (*p*, *acq1*, *acq2*):

1. p is a participant name in CC;
2. the union of *acq1* and *acq2* is identical to the acquaintance clause of p in CC;
3. for all $a.q$ in *acq2*: no method in p has a in its specialisation clause.

A context coarsening reuser clause is applicable on a collaboration contract only if it satisfies three conditions. It should only refer existing participants (clause 1) and existing acquaintance clauses (clause 2). It should only remove acquaintances of a participant that are not referred to in the specialisation clause of any method in that participant's interface (clause 3).

Definition 48 (Context coarsening)

If a collaboration contract CC is context coarsenable by a reuser clause C_{cc} then the collaboration contract CC_{cc} is the context coarsening of CC by C_{cc}, where:

1. CC_{cc} contains all participants of CC that are not mentioned in C_{cc};
2. for each triple (*p*, *acq1*, *acq2*) in C_{cc}: CC_{cc} contains a participant with the same name and interface as p in CC and *acq1* as acquaintance clause.

The result of applying a context coarsening reuser clause on a collaboration contract is a new collaboration contract with the same participants. Participants not mentioned in the reuser clause are untouched (clause 1). The others have reduced acquaintance clauses (clause 2).

A.7 Combined Reuser Clauses

When collaboration contracts are reused or evolved, they are seldom subject to one reuser clause only. In general, a collaboration contract is subject to several reuser clauses applied successively. Some combinations of reuser clauses are frequently used to reuse or evolve a collaboration contract, as indicated by the following list.

Participant extension & participant refinement. This combination can be used to describe extra invocations of added methods. It corresponds to added method behaviour. These reuser clauses are combined with context extension and context refinement when the added methods are part of new participants.

Participant coarsening & participant cancellation. This combination describes the removal of methods, including the invocations of those methods. Only applying the cancellation could result in dangling references. Only applying the coarsening may result in unused methods if all corresponding method invocation are removed. The correct combination of the two reuser clauses avoids dangling references and redundant methods. These reuser clauses may be combined with context coarsening and/or context cancellation to reduce redundancies across participants.

Participant concretisation & participant refinement. This combination describes the concretisation of methods in which other methods are invoked, a typical change when abstract methods are overridden. In combination with participant extension, the three reuser clauses together describe the concretisation of methods in which added methods are invoked. The combination with context refinement may be necessary to set up new acquaintance relationships when the invoked methods reside on other participants.

Participant coarsening & participant refinement. This combination describes major changes to specialisation clauses: method invocations are removed and others are added. These reuser clauses may be combined with several other ones to add required methods or remove unused methods.

Participant coarsening & participant extension & participant refinement. This combination can be used to describe a method factorisation. In combination with context reuser clauses, the factorisation may extend across participants.

The importance of combined reuser clauses prompts the following definition.

Definition 49 (Combined reuser clause)

A **combined reuser clause** is a sequence of reuser clauses.

The corresponding contract type is ‘combined’. The word ‘sequence’ in the definition implies order, which is important for the correct application of the reuser clauses.

Definition 50 (Single reuser clause applicability and result of applying)

A reuser clause C **is applicable to** a collaboration contract CC, and CC_{result} is **the result of applying** C to CC in any of the following cases:

1. C is a participant extension reuser clause, CC is participant extendible by C, and CC_{result} is the participant extension of CC by C.
2. C is a participant cancellation reuser clause, CC is participant cancellable by C, and CC_{result} is the participant cancellation of CC by C.
3. C is a participant refinement reuser clause, CC is participant refinable by C, and CC_{result} is the participant refinement of CC by C.
4. C is a participant coarsening reuser clause, CC is participant coarsenable by C, and CC_{result} is the participant coarsening of CC by C.
5. C is a participant specialisation reuser clause, CC is participant specialisable by C, and CC_{result} is the participant specialisation of CC by C.
6. C is a participant concretisation reuser clause, CC is participant concretisable by C, and CC_{result} is the participant concretisation of CC by C.
7. C is a participant abstraction reuser clause, CC is participant abstractable by C, and CC_{result} is the participant abstraction of CC by C.
8. C is a context extension reuser clause, CC is context extendible by C, and CC_{result} is the context extension of CC by C.
9. C is a context cancellation reuser clause, CC is context cancellable by C, and CC_{result} is the context cancellation of CC by C.
10. C is a context refinement reuser clause, CC is context refinable by C, and CC_{result} is the context refinement of CC by C.
11. C is a context coarsening reuser clause, CC is context coarsenable by C, and CC_{result} is the context coarsening of CC by C.

Definition 51 (Combined reuser clause applicability and result of applying)

A combined reuser clause $C = (C_1, \ldots, C_n)$ **is applicable to** a collaboration contract CC_0 if

- $\forall$ i $\in$ [1, n]: C_i is applicable to CC_{i-1}, with CC_i as result

CC_n is then called the **result of applying** C to CC_0.

Application of a combined reuser clause is thus defined as successive application of the reuser clauses in the combined reuser clause.

Appendix B

Definition of Accessor Methods

Several types of accessor methods can be distinguished. Accessor methods are also referred to as 'get' methods. They usually have 'set' counterparts. In this document, the 'set' methods are called *mutator methods*. In the following sections, the mutator methods are not given; only the accessor methods are discussed.

B.1 Direct Accessor

A direct accessor method is a method that returns the contents of an instance variable and does nothing else. In Smalltalk, such an accessor typically looks like this: `a ^a`.

B.2 Direct Accessor with Lazy Initialisation

Usually instance variables are initialised in an initialisation (Smalltalk) or constructor (C++, Java) method. When instance variables are bound to `nil` (or null) at object creation time, however, and when `nil` has no special meaning for the containing object, and when instance variables are not referenced directly in the source code, but instead always fetched through accessor methods, there is a widely used alternative for instance variable initialisation. A direct accessor method with lazy initialisation initialises an instance variable when the accessor is invoked for the first time. The accessor knows that it is invoked for the first time when the instance variable holds `nil`, instead of any other object. In Smalltalk, such an accessor typically looks like this: `a ^a == nil ifTrue: [a := <lazy initialisation expression>] ifFalse: [a]`.

B.3 Direct Accessor with Default

The lazy initialisation approach does not work when `nil` has a special meaning for the containing object. When `nil` can be returned to senders of the accessor message, a direct accessor would satisfy. When `nil` is not to be returned to senders of the accessor message, however, the direct accessor method with default provides a way to return a meaningful default object instead of `nil`. In Smalltalk, such an accessor typically looks like this: `a ^a == nil ifTrue: [<default object expression>] ifFalse: [a]`.

B.4 Indirect Accessor

An indirect accessor method is a method that returns the result of sending an accessor message to an object held in one of its instance variables. It is typically used as an abstraction technique to avoid multiple occurrences of compound accessor messages. Consequently, a variation that is frequently encountered in practice is an indirect accessor that contains a compound accessor message, such as `a b c d`. In Smalltalk, an indirect accessor typically looks like this: `a ^self b c`.

B.5 Design-Specific Accessor

The implementation of some accessor methods may be influenced strongly by the software design, especially when the state of an object is held by another object. For instance, when a strict separation between the domain model layer and the persistency layer is established, as is the case when the Bridge design pattern is applied, the state of a domain model object is held by its associated persistency layer object. Accessor methods on the domain model classes typically delegate the accessing behaviour to the persistency layer object. In Smalltalk, a design-specific accessor for this example looks like this: `a ^self asPersistentObject a asDomainObject`.

The message `asPersistentObject` is in fact an accessor message to fetch the persistency layer object corresponding to the receiver. The object that is returned by the accessor method `a` on the class of the persistency layer object is converted back into a domain model object by `asDomainObject`, so that the domain model accessor method returns a domain model object, instead of a persistency layer object.

B.6 Developer Dependent Accessor

Developers tend to develop their own style for accessor methods. Such accessor methods often overlap in functionality with the other varieties of accessor methods. For instance, `a ^b ifNotNilDo: [:p | p c]` is an accessor method that returns `nil` when the accessed instance variable holds nil, or the result of the expression in the block when the instance variable does not hold `nil`. This is in fact an indirect accessor combined with the possibility that the desired object cannot be fetched (`nil` has special meaning). A variant is `a ^b ifNotNilDo: [:p | p c] default: [<expression>]` that returns the result of the expression in the default block when the desired object cannot be fetched (`nil` has special meaning for the object, but not for its clients).

B.7 Collection Accessor

While instance variable accessors typically have no arguments, collection accessors typically have one argument, being some key into to collection. In Smalltalk, collection accessors typically look like this: `at: key ^<expression based on key>`.

Appendix C

Example Generated Rose Script

This script is the Rational/Rose script that generates the UML class diagram and the UML collaboration diagram in Figure 9.11 and Figure 9.12. The script was generated by the Classification Browser.

```
'Script for Rational/Rose.
'Generated by the Classification Browser
 on June 12, 1998 12:08:09.000.


Sub Main
Const collaborationDiagramType = 2

Dim model As Model
Set model = RoseApp.CurrentModel
Dim rootCategory As Category
Set rootCategory = model.RootCategory
Dim mainClassDiagram As ClassDiagram
Set mainClassDiagram =
     rootCategory.AddClassDiagram("Generated Packages")

Dim newPackage As Category
Dim theClassDiagram As ClassDiagram
Dim theAssociation As Association
Dim cc As ScenarioDiagram
Dim theMessage As Message

Set newPackage = rootCategory.AddCategory
    ("Library position for video medium")

b = mainClassDiagram.AddCategory(newPackage)
```

```
Set theClassDiagram = newPackage.AddClassDiagram
    ("Class Diagram")
Set cc = newPackage.AddScenarioDiagram
    ("Collaboration Diagram", collaborationDiagramType)

Dim class3 As Class
Set class3 = newPackage.AddClass
    ("PSIVideoMedium..PSIVideoMedium")
Set object3 = cc.CreateObject
    ("videoMedium", "PSIVideoMedium..PSIVideoMedium")

Dim class4 As Class
Set class4 = newPackage.AddClass("PSIStdLibrary..PSIStdLibrary")
Set object4 = cc.CreateObject("library","PSIStdLibrary..PSIStdLibrary")

Dim class1 As Class
Set class1 =
    newPackage.AddClass("PSILibPosition class..PSILibPosition  class")
Set object1 = cc.CreateObject("PSILibPosition","PSILibPosition
    class..PSILibPosition class")

Dim class2 As Class
Set class2 = newPackage.AddClass("PSILibPosition..PSILibPosition")
Set object2 =
    cc.CreateObject("libPosition","PSILibPosition..PSILibPosition")

b = class1.AddOperation("newInLibrary:videoMedium:","")
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} nr",object1,object3,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} free:",object1,object2,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} vmType",object1,object3,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} code:",object1,object2,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} vmType:",object1,object2,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} nextLibPositionCode",object1,object4,0)
Set theMessage = cc.CreateMessage("{newInLibrary:videoMedium:
    invokes} library:",object1,object2,0)
Set theAssociation =
    class1.AddAssociation("pos",class2.Name)
Set theAssociation =
     class1.AddAssociation("aVideoMedium",class3.Name)
Set theAssociation =
     class1.AddAssociation("aLibrary",class4.Name)

b = class2.AddOperation("code:","")
b = class2.AddOperation("canBeUsedBy:","")
b = class2.AddOperation("library:","")
b = class2.AddOperation("free:","")
b = class2.AddOperation("vmType:","")

b = class3.AddOperation("vmType","")
b = class3.AddOperation("nr","")

b = class4.AddOperation("newLibPositionForVideoMedium:","")
Set theMessage =
    cc.CreateMessage("{newLibPositionForVideoMedium: invokes}
    newInLibrary:videoMedium:",object4,object1,0)
b = class4.AddOperation("nextLibPositionCode","")
b = class4.AddOperation("libPositionForVideoMedium:throwException:","")
Set theMessage =
    cc.CreateMessage("{libPositionForVideoMedium:throwException:
    invokes} createNewLibPositionForVideoMedium:",object4,object4,0)
Set theMessage =
     cc.CreateMessage("{libPositionForVideoMedium:throwException:
     invokes} nr",object4,object3,0)
Set theMessage =
     cc.CreateMessage("{libPositionForVideoMedium:throwException:
     invokes} newLibPositionForVideoMedium:",object4,object4,0)
Set theMessage =
     cc.CreateMessage("{libPositionForVideoMedium:throwException:
     invokes} getLibPositionForVideoMedium:",object4,object4,0)
Set theMessage =
     cc.CreateMessage("{libPositionForVideoMedium:throwException:
     invokes} firstFreeLibPositionForVideoMedium:",object4,object4,0)
b = class4.AddOperation("createNewLibPositionForVideoMedium:","")
Set theMessage =
     cc.CreateMessage("{createNewLibPositionForVideoMedium:
     invokes} newInLibrary:videoMedium:",object4,object1,0)
b = class4.AddOperation("firstFreeLibPositionForVideoMedium:","")
b = class4.AddOperation("getLibPositionForVideoMedium:","")
Set theAssociation = class4.AddAssociation("PSILibPosition.siteClass",
    class1.Name)
Set theAssociation = class4.AddAssociation("aVideoMedium",class3.Name)
Set theAssociation = class4.AddAssociation("self",class4.Name)
```

```
b = theClassDiagram.AddClass(class3)
b = theClassDiagram.AddClass(class4)
b = theClassDiagram.AddClass(class1)
b = theClassDiagram.AddClass(class2)

theClassDiagram.Layout
cc.Layout
mainClassDiagram.Layout
End Sub
```

Bibliography

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AH95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *Proceedings of OOPSLA'95*, pages 91–107. ACM/SIGPLAN, October 1995.

[Aji95] Samuel Ajila. Software Maintenance: An Approach to Impact Analysis of Object Change. *Software – Practice and Experience*, 25(10):1155–1181, October 1995.

[Art87] Lowell Jay Arthur. *Software Evolution. The Software Maintenance Challenge.* John Wiley & Sons, 1987.

[BA96a] Shawn A. Bohner and Robert S. Arnold. An Introduction to Software Change Impact Analysis. In *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[BA96b] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[BA96c] Shawn A. Bohner and Robert S. Arnold. Software Maintenance. In *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[Ben97] Keith H. Bennett. Software Maintenance: A Tutorial. In *Software Engineering.* IEEE Computer Society Press, 1997.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of OOPSLA'93 (Sep. 26 – Oct. 1, Washington, DC)*, pages 215–230. ACM/SIGPLAN, 1993.

[BI82] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 133–139, 1982.

[Big89] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[Boo93] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1993.

[Boo94] Grady Booch. Coming of Age in an Object-Oriented World. *IEEE Software*, 11(6):33–41, November 1994.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. Unified Method Language 1.0. Technical report, Rational, 1997.

[Bro96] Kyle Brown. Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Technical Report TR-96-07, North Carolina State University, 1996.

[CAB$^+$94] D. Coleman, P. Arnold, S. Bdoff, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: the Fusion Method.* Prentice Hall, 1994.

[Cas96] E. Casais. State-of-the-art in OO Re-Engineering Methods. FAMOOS internal report SOAMETH-A1.3.1, October 1996.

[CC90] Elliot J. Chikofski and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From Custom Applications to Domain-Specific Frameworks. *Communications of the ACM*, 40(10):71–77, October 1997.

[CO88] James S. Collofello and Mikael Orn. A Practical Software Maintenance Environment. In *Proceedings of the Conference on Software Maintenance*, pages 45–51. IEEE Press, 1988. Reprinted in Software Change Impact Analysis. IEEE Computer Society Press, 1996.

[Cor96] Rational Software Corporation. *Rational Rose Extensibility Reference Manual*, 1996.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, 1991.

[Dav94] Alan M. Davis. Fifteen Principles of Software Engineering. *IEEE Software*, 11(6):94–101, November 1994. Excerpt of 201 Principles of Software Engineering. McGraw-Hill, 1994.

[FPG94] Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 11(4):86–95, July 1994.

[GAL97] Adele Goldberg, Steven T. Abell, and David Liebs. The LearningWorks Development and Delivery Frameworks. *Communications of the ACM*, 40(10):78–81, October 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[Gla94] Robert L. Glass. The Software Research Crisis. *IEEE Software*, 11(6):42–47, November 1994.

[GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding with Objects. Decision Frameworks for Project Management*. Addison-Wesley, 1995.

[Gra89] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioural Composition. In *Object-Oriented Systems, Proceedings of the OOPSLA-ECOOP'90 Conference. ACM Sigplan Notices 25(10)*, pages 169–180. ACM Press, 1990.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, Boston, 1995.

[HLS97] Koen De Hondt, Carine Lucas, and Patrick Steyaert. Reuse Contracts as Component Interface Descriptions. In *Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, number 5 in TUCS General Publication, pages 43–49, September 1997.

[Hon93] Koen De Hondt. A Customizable, Ergonomic, Hybrid Structure-Oriented Editor. Master's thesis, Programming Technology Lab, Vrije Universiteit Brussel, Brussels, Belgium, 1993.

[Hon98] Koen De Hondt. ApplFLab: Application Framework Laboratory. Description, publications and other documents, examples, and software artifacts on http://progwww.vub.ac.be/pools/applflab/, 1998.

[How95] Tim Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, 1995.

[HYR96] David R. Harris, Alexander S. Yeh, and Howard B. Reubenstein. Extracting Architectural Features From Source Code. *Automated Software Engineering*, 3:109–138, 1996.

[IBM98] IBM. *VisualAge for Java, Version 2.0*, 1998.

[IEE91] IEEE. IEEE Standard Glossary of Software Engineering Terminology, 1991. Std. 610.12-1990.

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

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success.* Addison Wesley (ACM Press), 1997.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *Proceedings of OOPSLA'88, Conference on Object-Oriented Programming, Systems, Languages, and Applications. (Sep. 25–30, San Diego, California)*, pages 18–25, November 1988.

[Joh86] Ralph E. Johnson. Type-Checking Smalltalk. In *Proceedings of OOPSLA'86, Conference on Object-Oriented Programming, Systems, Languages, and Applications. (Sep. 29 - Oct. 2, Portland, Oregon)*, pages 315–321, November 1986. Printed as SIGPLAN Notices, 21(11).

[KP88] G.E. Krasner and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 21(3), March 1988.

[Lar98] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1998.

[Leh84] M. M. Lehman. Program Evolution. *Information Processing & Management*, 20(1-2):19–36, 1984.

[Leh97] M. M. Lehman. Laws of Software Evolution Revisited. Technical report, Department of Computing, Imperial College, London, United Kingdom, 1997.

[Lie96] Karl. J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[LSM96] Carine Lucas, Patrick Steyaert, and Kim Mens. Research Topics in Composability. In *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming*, pages 81–86. dpunkt.verlag, 1996.

[LSM97] Carine Lucas, Patrick Steyaert, and Kim Mens. Managing Software Evolution through Reuse Contracts. In *Proceedings of the First Euromicro Conference on Software Maintenance and Reengineering*, pages 165–168. IEEE Press, 1997.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Programming Technology Lab, Vrije Universiteit Brussel, Brussels, Belgium, 1997.

[MLS98] Tom Mens, Carine Lucas, and Patrick Steyaert. Supporting Reuse and Evolution of UML Models. International Workshop UML'98, Mulhouse, France, June 1998, pages 341-350, 1998.

[MSL96] Kim Mens, Patrick Steyaert, and Carine Lucas. Reuse Contracts: Managing Evolution in Adaptable Systems. In *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming*, pages 37–42. dpunkt.verlag, 1996.

[Mur96] Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[OJ93] W. F. Opdyke and R.E. Johnson. Creating Abstract Superclasses by Refactoring. In *Proceedings of the 21st Annual Computer Science Conference (ACM CSC'93) (Indianapolis, Feb. 16–19)*, pages 66–73. ACM Press, 1993.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Oss87] Harold L. Ossher. A Mechanism for Specifying the Structure of Large, Layered Systems. In *Research Directions in Object-Oriented Programming, B. Shriver and P. Wegner Eds*, pages 219–252. The MIT Press, 1987.

[OTI95] OTI. *Envy/Developer R3.01 User Manual*, 1995.

[OTI97] OTI. *UML Document Set, Version 1.1*, September 1997. OMG Documents ad/97-08-03 to ad/97-08-05.

[Par94] David L. Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (Sorento, Italy, May 16–21)*, pages 279–287. IEEE Press, 1994.

[Par95] David L. Parnas. On ICSE's Most Influencial Papers. *ACM Software Engineering Notes*, 20(3):29–32, July 1995.

[PD91] Rubén Prieto-Díaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[PD95] ParcPlace-Digitalk. *VisualWorks© User's Guide*, 1995.

[PDF87] Rubén Prieto-Díaz and Peter Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of OOPSLA'93 (Sep. 26 Oct. 1, Washington, DC)*, pages 326–337. ACM/SIGPLAN, 1993.

[Pot93] Colin Potts. Software-Engineering Research Revisited. *IEEE Software*, 10(5):19–28, September 1993.

[PS91] Jens Palsberg and Michael I. Schartzbach. Object-Oriented Type Inference. In *Proceedings of OOPSLA'91 (Oct. 6–11, Phoenix, Arizona)*, pages 146–161. ACM/SIGPLAN, 1991.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Modeling and Design*. Prentice Hall, 1991.

[Sch97] Hans Albrecht Schmid. Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10):48–51, October 1997.

[SF97] Douglas C. Schmidt and Mohamed E. Fayad. Lessons Learned Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM*, 40(10):85–87, October 1997.

[SFJ96] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. Software Patterns. *Communications of the ACM*, 39(10):37–39, October 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SHDB96] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Niels Boyen. Reflective Application Builders. In *Chris Zimmermann, editor, Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press Inc., Boca Raton, Florida, 1996.

[SHDM95] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Marleen De Molder. A Layered Approach to Dedicated Application Builders Based on Application Frameworks. In *D. Patel, Y. Sun, and S. Patel, editors, Proceedings of the 1994 International Conference on Object-Oriented Information Systems*, pages 252–265. Springer-Verlag, 1995.

[SLM97] Patrick Steyaert, Carine Lucas, and Kim Mens. Reuse Contracts: Making Systematic Reuse a Standard Practice. In *Proceedings of WISR 8, Eight Annual Workshop on Software Reuse (Mar. 23–26, Columbus, Ohio)*, March 1997.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA'96 (Oct. 6–10, San Jose, California)*, pages 268–285. ACM/SIGPLAN, 1996.

[Sof98] Take Five Software. Sniff, 1998. http://www.takefive.com/.

[Ude94] J. Udell. ComponentWare. *Byte*, 19(5):46–56, May 1994.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-Oriented Design: A Responsibility Driven Approach. In *Proceedings of OOPSLA'89, ACM SIGPLAN Notices*, pages 71–75, October 1989.

[Wuy98] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proceedings of TOOLS USA'98*, 1998.

Vita

Koen De Hondt was born on March 3, 1967, in Wilrijk (Antwerp), Belgium. After graduating from high school in 1985, he started a university education in mathematics, but soon realised that computer science was his big dream. Therefore, he started computer science studies at the Vrije Universiteit Brussel in 1986. He graduated in July 1990, with a Licentiate in Computer Science in the Faculty of Sciences. He started as a research/teaching assistant at the Vrije Universiteit Brussel in September 1990. During three years, he was part of the Agora group, a small team investigating reflective prototype-based object-oriented languages. In September 1993, he received the degree of Master of Computer Science from the same institution The subject of his Master thesis was structure-oriented editing [Hon93]. From the fall of 1993 until the spring of 1995 Koen focussed on reflective application builders. ApplFLab, or Application Framework Laboratory [SHDM95], [SHDB96], [Hon98], was the artefact that resulted from that research. For two years from March 1995, he worked as a fulltime researcher on an industry-sponsored research project, of which the major goal was to build framework development tools. He received the degree of Doctor of Philosophy on December 11, 1998, from the Vrije Universiteit Brussel.