Vrije Universiteit Brussel
Faculteit Wetenschappen - Departement Informatica

Documenting Reuse and Evolution with Reuse Contracts

Carine Lucas
September 1997

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

Contents

Acknowledgements **1**

Introduction **3**

1 Issues in Reuse and Composition **7**

1.1 Conflicts with Evolving Components . . . 7
1.1.1 An Example . . . 7
1.1.2 An Overview of Possible Conflicts . . . 11
1.2 The Evolution to Component Software . . . 16
1.2.1 Benefits and Inhibitors . . . 16
1.2.2 Systematic Reuse: a Paradigm Shift . . . 17
1.2.3 Kinds of Component Systems . . . 17
1.2.4 The Development of Reusable Components . . . 19
1.2.5 Relationship with Reuse Contracts . . . 20
1.3 Object-Oriented Reuse . . . 21
1.3.1 Polymorphism, Protocols and Inheritance . . . 21
1.3.2 Abstract Classes and Template Methods . . . 21
1.3.3 Frameworks . . . 22
1.3.4 Object-Oriented Methodologies . . . 24
1.3.5 Language Extensions . . . 25
1.3.6 Relationship with Reuse Contracts . . . 26
1.4 Documenting Reusable Components . . . 27
1.4.1 Specialisation Interfaces . . . 27
1.4.2 Contractual Interfaces . . . 28
1.4.3 Documenting Frameworks . . . 28
1.4.4 Cookbooks . . . 29
1.4.5 Design Patterns . . . 29
1.4.6 Interaction Contracts . . . 30
1.4.7 Interface Definition Languages . . . 31
1.4.8 Architecture Description Languages . . . 31
1.4.9 Relationship with Reuse Contracts . . . 32
1.5 Evolution of Reusable Components . . . 32

1.5.1 Binary Compatibility 33
1.5.2 Refactoring and Restructuring 33
1.5.3 Programming by Contract and Formal Methods 34
1.5.4 Consistency of Class Libraries 34
1.5.5 Relationship with Reuse Contracts 35
1.6 Our Approach: Reuse Contracts 35
1.6.1 Summary: the Problems 35
1.6.2 Another Example 36
1.6.3 Reuse Contracts 37
1.6.4 Structure of the Dissertation 40

2 Basic Reuse Contracts **41**
2.1 Definition of Reuse Contracts 41
2.1.1 Participants 41
2.1.2 Acquaintance Clauses 42
2.1.3 Client Interface 43
2.1.4 The Specialisation Interface 44
2.1.5 The ATM Example 45
2.1.6 Well-Formedness 47
2.2 Operators on Reuse Contracts 48
2.2.1 Participant Extension 50
2.2.2 Context Extension 55
2.2.3 Participant Cancellation 58
2.2.4 Context Cancellation 61
2.2.5 Participant Refinement 64
2.2.6 Context Refinement 68
2.2.7 Participant Coarsening 71
2.2.8 Context Coarsening 74
2.2.9 Summary 77

3 Managing Evolution and Composition **79**
3.1 Evolution and Composition of Basic Modifiers 79
3.2 Interface Conflicts 81
3.2.1 Operation Name Conflicts 81
3.2.2 Participant Name Conflicts 82
3.2.3 Operation Invocation Conflicts 82
3.2.4 Acquaintance Relationship Conflicts 84
3.2.5 Summary of Interface Conflicts 84
3.3 Dangling Reference Conflicts 84
3.3.1 Dangling Operation Reference 85
3.3.2 Dangling Participant Reference 85
3.3.3 Dangling Acquaintance Reference 86
3.3.4 Summary of Dangling Reference Conflicts 86

3.4 Conflicts Concerning the Calling Structure 87
3.4.1 Operation Capture 87
3.4.2 Inconsistent Operations 89
3.4.3 Unanticipated Recursion 91
3.4.4 Summary of Conflicts about the Calling Structure 93
3.5 Evaluation 93
3.5.1 Alternative Rules 93
3.5.2 Other Possible Conflicts 95
3.6 Evolution of Chains of Adaptations 96
3.6.1 Chain vs. Single Modifier 97
3.6.2 Annihilation of Conflicts 98
3.6.3 Dependence of Modifiers 101
3.6.4 Transitive Closure Conflicts 102
3.6.5 Summary: Single Modifier versus Chain of Modifiers 104
3.6.6 Conflicts between Two Chains of Modifiers 104
3.6.7 Conclusion 105

4 Combined Operators **107**
4.1 Composition of Modifiers 107
4.1.1 Applicability 107
4.1.2 Definition and Properties 108
4.1.3 Discussion 109
4.2 Extension and Refinement 109
4.3 Connected Extension 110
4.4 Extending Refinement 114
4.5 Factorisation 116
4.6 Renaming 124
4.7 Summary 125

5 Reuse Contracts for the UML **127**
5.1 Basic Static Structure Diagrams 128
5.2 Integrating the Operators 131
5.3 Impact of Inheritance on the Conflicts 133
5.4 Integrating Late Binding 135
5.4.1 Self Sends 135
5.4.2 Super Sends: Specialisation 136
5.5 Abstract Classes and Methods 139
5.5.1 Extension of the Model 140
5.5.2 A New Operator: Participant Concretisation 141
5.5.3 A Combined Operator: Layered Concretisation 147
5.6 Implementing Reuse Contracts 148
5.7 Collaboration Diagrams 150
5.8 Acquaintance Relationships 152

5.8.1 Extension of the Model 153
5.8.2 A New Operator: Context Concretisation 154
5.8.3 Implementing Collaboration Reuse Contracts 155
5.9 Conclusion 155

6 Evolution of a Reusable Design 159
6.1 Background: Broadcast Planning 159
6.2 The Case: Air-Time Sales 159
6.3 A First Design: Block Spot Spaces 161
6.4 A Second Design: Gross Rating Points 163
6.5 Generalisation 164
6.6 Expressing Specialisations through Reuse Operators 165
6.7 Evolution 165
6.7.1 A Combined System 165
6.7.2 Introducing Clash Codes 172
6.7.3 Optimisation 174
6.8 Conclusion 175
6.9 Acknowledgements 176

7 Reuse Contracts at Work 177
7.1 Extraction 178
7.2 Compliance Checking 182
7.3 A Drawing Tool 185
7.4 Documentation 185
7.4.1 Dependencies between System Parts 185
7.4.2 Assistance of the Software Engineer 185
7.4.3 Layering of Design 186
7.4.4 Core Methods versus Peripheral Methods 188
7.4.5 Layering of Class Hierarchies 190
7.4.6 Views 191
7.5 Design Guidelines and Quality Assessment 192
7.5.1 Well-formed Reuse Contracts 192
7.5.2 Assessments Based on the Contracts and Operators 193
7.5.3 Existing Design Guidelines 195
7.5.4 Tools for Quality Assessment 197
7.6 Enforcing Design 197
7.7 Evolution and Incremental Development 200
7.7.1 Consistency Checking and Conflict Detection 200
7.7.2 Traceability 201
7.8 Re-engineering and Reverse Engineering 202
7.8.1 Extraction 202
7.8.2 Refactoring 202
7.9 Conclusion 202

7.10 Acknowledgements 204

8 Conclusion **205**

8.1 Summary 205

8.2 Evaluation and Future Work 206

8.2.1 Possible Extensions 207

8.2.2 Other Uses 210

8.2.3 Ameliorations to the Model 211

8.3 Main Contribution 212

A Conflict Detection Rules **213**

Bibliography **217**

List of Figures

1.1 Packet Handling in a LAN 8
1.2 Introducing Gateways 9
1.3 Introducing Visitor Packets 9
1.4 Combining Gateways and Visitorpackets 10
1.5 Broken Assumptions 11
1.6 Example of an Interface Conflict 12
1.7 Dangling Reference Conflicts 13
1.8 Regular Operation Capture 14
1.9 Accidental Operation Capture 14
1.10 Inconsistent Operations 15
1.11 Unanticipated Recursion 16
1.12 Evolution of Set 37
1.13 Broken Assumptions 38
1.14 Inconsistent Operations on Set 40

2.1 Two Acquainted Participants 42
2.2 A Participant's Client Interface 43
2.3 Part of the Protocol between Two Participants 44
2.4 The ATM Reuse Contract 46
2.5 An Example Participant Extension 51
2.6 Participant Extension 53
2.7 An Example Context Extension 56
2.8 Context Extension 57
2.9 An Example Participant Cancellation 59
2.10 Participant Cancellation 60
2.11 An Example Context Cancellation 61
2.12 Context Cancellation 63
2.13 An Example Participant Refinement 64
2.14 Participant Refinement 67
2.15 An Example Context Refinement 68
2.16 Context Refinement 70
2.17 An Example Participant Coarsening 71
2.18 Participant Coarsening 73

2.19 An Example Context Coarsening 74
2.20 Context Coarsening 76

3.1 Base Reuse Contract Exchange 80
3.2 An Operation Name Conflict 82
3.3 An Operation Invocation Conflict 83
3.4 A Dangling Operation Reference 86
3.5 Regular Operation Capture 88
3.6 Inconsistent Operations 90
3.7 Unanticipated Recursion 91
3.8 Indirect Unanticipated Recursion 92
3.9 An Acquaintance Relationship Conflict 96
3.10 Chain vs. Single Modifier 97
3.11 Transitive Closure Conflict Annihilation 103
3.12 Two Chains of Modifiers 105

4.1 An Example Connected Extension 112
4.2 Connected Extension 113
4.3 An Example Extending Refinement 115
4.4 Extending Refinement 117
4.5 An Example Factorisation 118
4.6 Factorisation 120
4.7 Annihilating Conflicts through Factorisation 122
4.8 An Example of Renaming 124

5.1 Model-View-Controller 129
5.2 MVC for BasicButtonView and BasicButtonController 131
5.3 Contract Refinement 133
5.4 Refinement of Controller 134
5.5 Inconsistent Methods on Set 134
5.6 The Representation of Self Sends 136
5.7 Message Sends between Instances of the Same Class 136
5.8 An Example of Specialisation 138
5.9 An Abstract Class 140
5.10 Participant Concretisation 143
5.11 Layered Concretisation 149
5.12 A Collaboration Diagram 151

6.1 Overview of the Air-Time Sales System 160
6.2 Block Spot Spaces 162
6.3 Gross Rating Points 163
6.4 Generalised Air Time Sales Behaviour 164
6.5 Specialising the Air Time Sales Contract 166

6.6 Specialising Air Time 167
6.7 Specialising Planner 167
6.8 Specialising Block Distribution 168
6.9 A Combined System 169
6.10 Evolution of Air Time 170
6.11 Revision of Air Times 171
6.12 Introducing Clash Codes 172
6.13 Optimising Distributions 175

7.1 Reuse Contract Extractor 179
7.2 Decomposition into Reuse Operators 180
7.3 A Layered Design of Buttons 187
7.4 Clustering of the Class Dictionary 189
7.5 Spotting Possible Design Flaws 194
7.6 Part of the AWT Hierarchy 196
7.7 The Decorator Design Pattern 198
7.8 Reuse Contracts for Decorator 199
7.9 Generic Structure of Decorator 199

List of Tables

2.1 Basic Operators 49

3.1 Interface Conflicts 85
3.2 Dangling Reference Conflicts 87
3.3 Conflicts concerning the Calling Structure 94
3.4 Dependencies between Modifiers 101

4.1 Conflicts with Extension and Refinement 111
4.2 Conflicts with Connected Extension 114
4.3 Conflicts with Factorisation 123

5.1 Conflicts with Specialisation 139
5.2 Conflicts with Participant Concretisation and Abstraction 147

6.1 Adding `validSol:in:` to `AirTime` 170
6.2 Adding `reschedulable` to `AirTime` 171
6.3 Clash Code Behaviour on `Distribution` 173
6.4 Clash Code Behaviour on `AirTime` 174
6.5 Optimisation of ATSContract - change 1 175
6.6 Optimisation of ATSContract - change 2 176

A.1 Interface Conflicts 213
A.2 Dangling Reference Conflicts 213
A.3 Conflicts concerning the Calling Structure 214
A.4 Conflicts with Specialisation 214
A.5 Conflicts with Participant Concretisation and Abstraction 215

Acknowledgements

I thank my advisor, prof. Theo D'Hondt for convincing me a PhD was well within my stride. I highly respect the way he runs the Programming Technology Lab and is concerned for both his staff and his students. He was always generous with advice and support at crucial times and provided me with a strict deadline when I needed one.

I also owe a lot of gratitude to my co-advisor, dr. Patrick Steyaert for starting up and leading the Reuse Contracts Group, thus providing me with an inspiring and original subject. He always made time for discussions on the big principles as well as the technical details and always encouraged me to try and do just a bit better. He also had a substantial influence on the structure of this text.

I thank Kim Mens, my "partner in crime". I hope our co-operation will prove to be as valuable to his research as it was to mine. Kim helped in every stage of this work. Moreover, he had to share an office with me during these trying times and passed the test with distinction.

I thank Koen De Hondt, Tom Mens and Roel Wuyts, who — as part of the Reuse Contracts Group — were instrumental in most of the experiments described in chapters 6 and 7. They were helpful with a lot of things and did a fair amount of proof reading of numerous drafts and versions of this dissertation.

I thank the members of my jury: Mehmet Aksit, Viviane Jonckers, Oscar Nierstrasz and Dirk Vermeir for their feedback on how to improve the text and on interesting research topics to pursue next. A jury this astute is a source of inspiration.

Wim Codenie and Wilfried Verachtert from OOPartners provided the crucial practitioner's feedback and were also very helpful in working out the ATS case from chapter 6.

Wolfgang De Meuter and Serge Demeyer were my two most feared proof readers, which only goes to say their comments were very valuable. Stephane Ducasse —

though we never even met — volunteered to proof read, which was greatly appreciated. Kris De Volder also read parts of this thesis.

Thomas Unger was very helpful and valuable as LaTeX specialist. Thomas and Wolfgang also helped to relieve some of my educational tasks when time was running short.

As part of his graduation thesis Gerrit Cornelis extended Java interfaces to incorporate reuse contracts and worked out the AWT example, which I use at a number of places.

I'd like to thank everybody at the Programming Technology Lab and the Department of Computer Science for supporting me in all kinds of ways over the last few years. Besides those mentioned above, I thank Brigitte Beyens, Niels Boyen, Jan De Laet, Tom Lenaerts, Wim Lybaert, Lydie Seghers, Marc Van Limberghen, Mark Willems, and especially Linda Dasseville for the many inspiring conversations. Thanks for making the Programming Technology Lab not only an inspiring, but also a fun place to work.

I thank my friends for providing me with the necessary distraction once in a while and for putting things in perspective. I especially thank Marleen Easton. Though working in completely different fields, we seemed to share all the same experiences.

I thank my parents not only for giving me the opportunity to study, but for letting me grow up to believe I can achieve anything I set my mind to. Likewise, I thank my brother and his family and my grandparents for supporting me in everything I did over the years.

Finally, I thank Kristof. Without his love, patience and support I never would have finished this.

Carine Lucas
September 1997

Introduction

In recent years software development has been subject to numerous innovations, with a focus on reuse and increasing productivity. A shift is noticeable from software engineering as a discipline concerned with the construction of hand-crafted, single systems, to an industry centred around the production of software components, aimed at building systems much like product lines. Software engineering techniques have not been able to keep up with this rapid evolution. Amongst others object-orientation has failed to deliver much of its promises, while formal techniques do not succeed in getting widely adopted. The classical waterfall model does not serve the new paradigm well, while new iterative process models have not yet reached an adequate level of maturity. The central tenet of this dissertation is that *evolution* is at the heart of reuse. Evolution is crucial because reusable components have a long life span, because good reuse can only be achieved after a component has been reused and adapted several times and finally, because it is simply not conceivable to predict all possible uses of a component upon its conception. Current software engineering techniques focus too much on passive support as separation of concerns, separation of interface and implementation and formal specifications. Tools actively supporting software engineers in issues as traceability and change management are completely lacking. This dissertation introduces reuse contracts as structured documentation to support the evolution of reusable components.

The study of different approaches to reuse reveals that there is a general understanding that reusable systems — be it libraries of reusable components, object-oriented frameworks or componentware approaches — should mainly be used in a pre-defined way: the basic structures should not be violated. Black-box frameworks where different variations of the components must be plugged into a general design are but the one example. Similarly, formal approaches to reuse often focus on behavioural subtyping, meaning that specialisations of components in a framework should always be substitutable for their basic component. Such approaches suffer from a lack of flexibility. First, allowing customisations that respect the original design only is based on the assumption that all possible reuses can be anticipated. Practice has proven it unfeasible, however, to develop reusable applications that comply with all the requirements of a large user community and that keep on doing so as time - and requirements - evolve. Second, such approaches do not take the intrinsic evolutionary nature of reusable systems into account. Reusable components tend to evolve after they have been developed and reused. Changes to components might be necessary to fix flaws in the requirements or when the requirements themselves evolve. More importantly, iterations over reusable components are inherent to their development. Therefore, managing the impact of changes on existing applications is crucial when components change.

So in order to get more *flexible* reusable systems, reusers should be allowed to make changes that were not foreseen. On the other hand, reuse should be *disciplined* enough to allow support on updating applications when the reusable components they are built on evolve.

Current approaches do not adequately address these needs. On the one hand, reuse approaches stress the need to address reuse in a systematic fashion, but they are often too coercive in allowing only reuse in predefined ways. On the other hand, object-oriented approaches to reuse as inheritance are much more flexible, but they lack discipline and formal underpinnings. Another important observation is that systematic reuse is concerned primarily with reuse of higher-level life cycle artifacts [Fra94]. As a consequence, the operational part, which is often the most extensive and the most complex component, is generally the result of a one-shot development effort. Object-oriented approaches as frameworks or design patterns often focus on lower-level artifacts. However, object-oriented methodologies as UML do not have an adequate notation for reuse and fall short in addressing notions of evolution and iteration. They still mainly focus on static descriptions of single systems, without adequate notations for families of systems and traceability between different variations. Therefore, we argue that a new approach for disciplined reuse, establishing a *vocabulary*, *notation* and *methodology* is required. Establishing such a new discipline, that guides developers in writing at least partway reusable software is an immensely complex task. This work represents the first stage in such an undertaking.

Let us start by setting up some criteria we deem crucial in such a methodology. A first concern is that we want to develop *practical models*, that are close to the actual code and are applicable in different stages of the life cycle. With a practical approach we imply models that are automatically processable and are a good basis for tools.

In order to reuse a system in ways different from what was foreseen, a general understanding of its structure and behaviour is essential. With structure we imply a description of how different parts in a system are arranged. By behaviour we mean a description of the way in which a system functions or operates. While most approaches to evolution focus on declarative behaviour, i.e., what a system does, we focus on *operational* behaviour, i.e., *how* it is achieved. We thus follow the spirit of recent research efforts such as Lamping's specialisation interfaces [Lam93], Holland's interaction contracts [Hol92] and Lieberherr's adaptive programming [Lie95]. We do however concentrate on documenting dependencies to which the compliance of code can be checked automatically.

A second important issue, when accepting the premises that evolution is at the heart of reuse, is *impact analysis*. The ability to perform impact analysis is key to numerous unsolved problems in software engineering. In the iterative development of frameworks and component systems, the ability to upgrade applications with new versions of the frameworks or components is paramount to gain a return on investment. In order to be able to upgrade applications it is crucial to be able to assess the impact of changes to components on the applications. When building applications from existing components, another unsolved problem is how to decide which components work together correctly. It is equally important there to be able to assess the impact of replacing one component with a slightly different version. Similarly, problems occur when applications are partially automatically generated. When adapting the input and re-generating code one needs to assess where the manually added code might cease to work with the newly generated code. Currently, no active support exists for any of these tasks. The work presented in this dissertation is a first effort towards a general approach for impact analysis that can assist in all of these problems.

The key to the solution is the observation that when reusing or adapting a system, developers make assumptions about how different parts of the system cooperate. When changes are made to part of a system, (some of) these assumptions might be broken. As currently these assumptions are always *implicit*, it is not possible to check whether they are respected upon change. Therefore, we suggest to make these assumptions *explicit*. This forms the basis for a structured approach to change propagation and impact analysis. The explicit documentation of assumptions implies that not only the component provider should provide adequate information about the components he delivers, but the reuser should also document the assumptions he relies on. This is the basis of a contract between provider and reuser.

This dissertation introduces *reuse contracts* for this purpose. Reuse contracts augment conventional interfaces with documentation of structural dependencies in a system. For example, information is added about which system components are acquainted and which operations rely on which other operations. This provides reusers with crucial information about the operational behaviour of a system. Moreover, this information can be retrieved statically, which makes automated support and the development of tools much easier.

Reuse contract interfaces can only be composed or adapted by means of certain predefined *reuse operators*. Reuse operators enable reusers to explicitly document the assumptions they make about the components they reuse and thus what parts of the interface they rely on. This makes their applications more robust to change, since explicitly documenting these assumptions allows verifying whether these assumptions are broken when changes are made. Similarly, explicitly documenting where the general design is not respected helps in assessing where co-operation with this part might cause problems.

Note that, as opposed to other methods, documenting assumptions is the basis of conflict *detection* rather than conflict *avoidance*. On the one hand, reusers can reuse components in any way they want. This accounts for the flexibility in our approach. On the other hand, they have to document the way they reuse components in a disciplined way. This accounts for the support for change propagation and impact analysis.

The work in this dissertation is part of a larger research effort aiming to establish a full-fledged methodology for disciplined reuse. Several researchers are involved covering various topics such as formalisation, tool support etc. This work establishes a blue-print for these different approaches by establishing an initial methodology, a partway formalisation and proof of concept on the basis of a number of non-trivial experiments.