Dissertation submitted for the degree of
Doctor of Philosophy
IT University of Copenhagen

Upgradable Software Product Customization by Code Query

Code Query by Example and the Upgrade Problem

Sebastien Vaucouleur

July 2009

Supervised by: Prof. Peter Sestoft

Can we build systems with simple and elegant designs that are **easy to understand, modify, and evolve** yet still provide the functionality we might take for granted today and dream of for tomorrow?

"Five Deep Questions in Computing", by Jeannette M. Wing. 50th anniversary issue of the Journal of the ACM [209]

Abstract

The professional press on enterprise systems warns its readers: *"Costly."*, *"A return on investment killer."*, *"Be prepared!"*. What can create some much affliction and turmoil in an otherwise very successful industry? Come and meet the dreadful *upgrade problem*.

Enterprise systems are prime examples of a subset of software systems that we call *software products*: software that needs special support for customization. Through customization, external companies can modify part of the original product to better fit the needs of a niche market. Upon the release of a new version of the original software product, external companies must port their customizations to the latest version of the base software product, a process called an *upgrade*. Companies typically consider upgrades as mandatory, and hence must bear their high cost on a regular basis. The objectives of customizability and upgradability are conflicting – this constitutes the upgrade problem.

We study the upgrade problem in the field of enterprise systems from a technical point of view, and consider the large spectrum of existing software engineering techniques for customization. We ground our work in an empirical study, that shows that customizations cannot be anticipated accurately. This result puts an important constraint on the solution and calls for an approach that complements the traditional customization techniques. We present the novel concept of *code query by example*, an approach that (a) requires little anticipation, (b) is simple and (c) may be adopted incrementally. Last but not least, this solution makes a subset of customizations amenable to upgrades. The implementation of our prototype is based on bytecode matching and bytecode instrumentation, in a managed .Net environment. We study the advantages, the disadvantages, and the limitations of our approach, both of the concept of code query by example, and of the implementation strategy. Finally, we show how our proposal can be used in other contexts where code query is needed, for example rule-based lightweight static analysis.

Acknowledgments

Thanks to my parents Marie-Dominique and Michel Vaucouleur for their support. Thanks to my main supervisor, Peter Sestoft, for his expert guidance. I would like to thanks all my close friends and colleagues – so many names to cite, but we shall proceed, following an alphabetical order. I would like to acknowledge the important friendship of my buddies in Nice: Estelle Barbot, Caroline Bjorkman, Elin Ekeberg, Eva Giraudo, Julien Guillot, Boris Moretti, Olivier Polia, Paul and Marie Ramoin, Patrick Smacchia, and René Valade. Thanks to my good friends from København: Emmanuelle Assenza, Claus Brabrand, Charlotte Denize, Nicolas Guilbert, Rita Larsen, Christine Fur Poulsen, and Asta Spulyte. Thanks to my friends and old colleagues at ETH Zürich: Jean-Raymond Abrial, Karine Arnout, Volkan Arslan, Arnaud Bailly, Stephanie Balzer, Ilinca Ciupa, Adam Darvas, Werner Dietl, Vijay D'silva, Patrick Eugster, Andreas Leitner, Farhad Mehta, Peter Müller, Michela Pedroni, Joseph Ruskiewicz, and Bernd Schoeller. We would like to thanks our co-author, Antonio Cisternino, with whom we developed the idea of code query by example during a stay at University of Pisa. This stay was made possible thanks to the kind invitation of Egon Börger. Thanks to colleagues at the Software Development Group at the IT University of Copenhagen: Jakob E. Bardram, Johan Bolmsten, Jonathan Bunde-Pedersen, Yvonne Dittrich (who co-supervised the empirical part of this work), Alberto Delgado-Ortegon, Afsaneh Doryab, Vibeke Ervø, Anders Hessellund, Rune Møller Jensen, Juan Ramos Hincapie, Søren Lauesen, Søren Lippert, Dario Pacino, Lene Pries-Heje, Rasmus Rasmussen, Morten Rhiger, Hataichanok Unphon, Vibeke Söderhamn Bülow, Neela Narayanan Venkataraman, Andrzej Wąsowski, Shangjin Xu, and Kasper Østerbye. Kind regards to all our colleagues from the FIRST research school, including the administrative staff who have always been very helpful. Thanks to our industrial partner, Microsoft, for their collaboration, especially: Stefen Giff, Lars Hammer, Thomas Jensen, Michael Nielsen, and Torben Wind. Thanks to my old colleagues at Steria Norway and Fujitsu Network Communications USA, with whom I learned a lot. Thanks to the open source community for conceiving and evolving

such a great set of tools and libraries, in particular the Mono project, the LyX project and the LaTeX community. This work takes place under the umbrella of the Evolvable Software Project, and is sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen. To the forgotten ones: if your name should have been in the list, please blame my memory, not my heart.

Contents

List of Figures xiii

List of Tables xiv

Listings xv

I. Thesis 1

1. Customization and upgrade of enterprise systems 2

1.1. Introduction . . . 2
1.1.1. The upgrade problem, a teaser . . . 2
1.1.2. Problem statement . . . 6
1.1.3. Significance . . . 6
1.1.4. Hypothesis (thesis statement) . . . 7
1.1.5. Contributions . . . 7
1.1.6. Research method . . . 7
1.2. Overview of enterprise systems . . . 10
1.2.1. What are ERP systems? . . . 10
1.2.2. Customization of ERP systems . . . 10
1.2.3. Microsoft Dynamics . . . 11
1.2.4. Dynamics NAV versus Dynamics AX . . . 12
1.2.5. Summary of Dynamics . . . 14
1.2.6. Empirical grounds . . . 14
1.2.6.1. Empirical research method . . . 14
1.2.6.2. Data sources . . . 14
1.2.6.3. Process . . . 14
1.2.6.4. Challenges . . . 15

1.2.6.5. Results of the empirical study . . . 15
1.2.6.6. Summary of the empirical study . . . 17
1.3. Concepts . . . 18
1.3.1. Concepts related to Object-Oriented Programming . . . 18
1.3.2. Concepts related to modern managed execution environments 21
1.3.3. Concepts related to software customization . . . 23
1.4. Modular decomposition . . . 26
1.4.1. Local reasoning and modular programming . . . 26
1.4.2. Decomposition à la Parnas . . . 26
1.4.3. The crystal ball assumption . . . 27
1.4.4. Towards code query by example . . . 28
1.5. Code Query by Example . . . 29
1.5.1. An embedded domain-specific language . . . 29
1.5.2. Specification of CQE . . . 29
1.5.2.1. Customization points . . . 30
1.5.2.2. Customization interfaces . . . 30
1.5.2.3. Customization classes . . . 30
1.5.2.4. Customization methods . . . 30
1.5.2.5. Customization objects . . . 30
1.5.2.6. Customization calls . . . 31
1.5.2.7. Query methods . . . 31
1.5.2.8. Queries . . . 31
1.5.2.9. Convention . . . 31
1.5.2.10. Matching of entities . . . 31
1.5.2.11. Disjunction of query methods . . . 32
1.5.2.12. Constraints to the definition of query methods . . . 32
1.5.2.13. Query variables . . . 32
1.5.2.14. Non-linear patterns . . . 33
1.5.2.15. Controlling customization points locations . . . 33
1.5.2.16. Generation of customization interfaces . . . 33
1.5.2.17. Implementation of customizations . . . 34
1.5.2.18. Instantiation of customization objects . . . 34
1.5.2.19. Partial ordering of customization calls . . . 34
1.5.3. Matching examples . . . 35
1.5.3.1. Empty query . . . 35
1.5.3.2. Simple query method . . . 36
1.5.3.3. Another simple query method . . . 36
1.5.3.4. Query variables . . . 37
1.5.3.5. Nonlinear patterns . . . 37
1.5.3.6. Action query variables . . . 38
1.5.3.7. Func query variables . . . 38
1.5.3.8. Try/catch blocks . . . 39
1.5.4. CQE extensions to the original concept . . . 39
1.5.4.1. Query method expansion . . . 40

1.5.4.2. Anchoring 40
1.6. Upgrade with CQE 42
1.6.1. True positives 43
1.6.1.1. Examples of true positive 43
1.6.1.2. Corrective actions 45
1.6.2. False positives 46
1.6.2.1. Example of false positive 46
1.6.2.2. Corrective actions 46
1.6.3. True negatives 48
1.6.3.1. Example of true negative 48
1.6.3.2. Corrective actions 48
1.6.4. False negatives 49
1.6.4.1. Examples of false negative 49
1.6.4.2. Corrective actions 50
1.6.5. Summary 50
1.7. Implementation of CQE 52
1.7.1. Design overview 52
1.7.2. Querying 52
1.7.2.1. Abstract syntax tree matching 53
1.7.2.2. Expression tree matching 55
1.7.2.3. Bytecode matching 56
1.7.3. Visualization 61
1.7.4. Instrumentation 62
1.7.5. Run-time loading of customizations 64
1.8. Limitations 66
1.8.1. Limitations of the approach 66
1.8.1.1. Slow edit-compile-run cycles 66
1.8.1.2. Fine-grained changes 67
1.8.2. Limitations of the prototype 68
1.8.2.1. Limitations due to non-local transformations 68
1.8.2.2. Limitations due to compiler optimizations 72
1.8.2.3. Limitations due to the use of regular expressions . . . 75
1.9. Further software customization techniques 78
1.9.1. In-place modifications and procedural abstraction 78
1.9.2. Covariance 80
1.9.3. Version Control Systems 82
1.9.4. Assembly versioning 85
1.9.5. Design patterns 86
1.9.6. Summary 92
1.10. Lessons learned and discussion 93
1.10.1. The upgrade problem, gathering our wits 93
1.10.2. Reflections on ERP systems 95
1.10.3. Reflections on enterprise systems research 97
1.11. Related work 99

1.12. Conclusions 103

II. Collection of Papers 106

2. Technologies for evolvable software products: The conflict between customizations and evolution 107
2.1. Introduction and definitions 107
2.2. The upgrade problem 109
2.2.1. Customizable software 109
2.2.2. Software evolution 110
2.2.3. The evolution of specifications 111
2.2.4. Upgrade problems in operating systems 111
2.2.5. Conclusion on the upgrade problem 112
2.3. Case study: Dynamics AX and NAV 112
2.3.1. Add-ons and configurations 112
2.3.2. Dynamics NAV versus Dynamics AX 113
2.3.3. The Dynamics developer ecosystem 114
2.3.4. What constitutes an upgrade 114
2.3.5. Upgrade problems in Dynamics NAV and Dynamic AX 114
2.3.6. Constraints on a solution to the Dynamics upgrade problem . . 115
2.3.7. Handling upgrade in Dynamics NAV 115
2.3.8. The layered structure of a Dynamics AX application 116
2.3.9. Customization using AX layers 116
2.3.10. Mitigating code upgrade problems in Dynamics AX 118
2.3.11. Another case study 118
2.4. Evaluation criteria 119
2.4.1. Need to anticipate customizations 119
2.4.2. Control over customizations 119
2.4.3. Resilience to kernel evolution 120
2.4.4. Support for multiple customizations 120
2.4.5. Runtime performance penalty 121
2.4.6. Illustration of the criteria 121
2.5. Survey of software customization methods 123
2.5.1. Inheritance 123
2.5.2. Information hiding using interfaces 126
2.5.3. Parametric polymorphism 128
2.5.4. Synchronous events 129
2.5.5. Partial methods as statically bound events 131
2.5.6. Mixins and traits 132
2.5.7. Aspect-oriented programming 134
2.5.8. Software product lines using AHEAD 138
2.5.9. Software product lines using multi-dimensional separation of concerns 140

2.5.10. The Dynamics AX layer model . . . 143
2.5.11. Summary evaluation . . . 143
2.6. Conclusion . . . 144

3. Customizations and upgrades of ERP systems: An empirical perspective 145
3.1. Introduction . . . 146
3.2. The ERP systems considered . . . 147
3.2.1. Dynamics AX . . . 147
3.2.2. Dynamics NAV . . . 147
3.3. The research method . . . 148
3.3.1. Video recordings . . . 148
3.3.2. Survey . . . 149
3.3.3. Semi-structured interviews . . . 149
3.3.4. How valid are our findings? . . . 150
3.4. Business and work practices for customization and upgrade of ERP systems . . . 151
3.4.1. An example: logging all actions related to customers . . . 151
3.4.2. The companies and the people . . . 152
3.4.3. Project organization and documents . . . 153
3.4.4. Customizing ERP systems . . . 155
3.4.5. Upgrading customizations . . . 157
3.4.5.1. Difficulties when upgrading customizations . . . 157
3.4.5.2. Upgrade practices . . . 158
3.4.6. Quality control . . . 159
3.4.7. Peer learning and knowledge sharing . . . 161
3.5. Topics for discussion . . . 162
3.5.1. A different kind of development . . . 162
3.5.2. Implications on Testing . . . 163
3.5.3. Development groups as communities of practice . . . 164
3.5.4. Making customizations first order inhabitants in the development environment . . . 165
3.5.5. Supporting practices of artful integration . . . 165
3.6. Conclusions . . . 166

4. Customizable and upgradable enterprise systems without the crystal ball assumption 167
4.1. Introduction . . . 168
4.1.1. Enterprise resource planning systems . . . 168
4.1.2. Definitions . . . 168
4.1.3. The Crystal ball assumption . . . 169
4.1.4. Anticipation is not a panacea . . . 170
4.1.5. The upgrade problem . . . 171
4.2. Empirical grounds . . . 172

4.3. Eggther framework . . . 173
4.3.1. Overview of the approach . . . 173
4.3.2. Code queries . . . 174
4.3.3. Customization code . . . 177
4.3.4. Unit testing . . . 180
4.3.5. Stateful customizations . . . 180
4.3.6. Aspect-oriented programming characterization . . . 181
4.4. Exactness and completeness of code queries . . . 181
4.4.1. Precision and recall . . . 181
4.4.2. Precision, recall and code queries . . . 182
4.4.3. Exactness and completeness problems upon upgrade . . . 182
4.4.4. Giving control back to the partners . . . 183
4.5. Implementation aspects . . . 183
4.5.1. General design choices . . . 183
4.5.2. Advantages . . . 184
4.5.3. User interface . . . 185
4.6. Further work . . . 185
4.6.1. Non-Boolean matching . . . 185
4.6.2. Partial ordering of customizations . . . 185
4.6.3. Toward behavioral customizations . . . 186
4.7. Discussion and Related work . . . 187
4.8. Conclusions . . . 189

5. Aspect-oriented programming made easy: An embedded pointcut language 190
5.1. Introduction . . . 190
5.1.1. Modular decomposition . . . 190
5.1.2. Limitations of modular decomposition . . . 191
5.1.3. Aspect oriented programming . . . 191
5.1.4. A teaser . . . 191
5.1.5. Applications . . . 192
5.1.6. Contributions . . . 192
5.1.7. Road-map . . . 192
5.2. Aspect-Oriented Programming: Terminology . . . 193
5.3. An embedded point cut language . . . 193
5.3.1. An embedded domain-specific language . . . 193
5.3.2. Notation . . . 193
5.3.3. Defining pointcuts . . . 194
5.3.4. Query variables . . . 194
5.3.5. Running example . . . 195
5.3.6. Interface generation . . . 195
5.3.7. Two advices . . . 196
5.3.8. Instrumentation and triggering of advices at join points . . . 197
5.3.9. Advice with a side effect at join point . . . 197

5.3.10. Code pattern matching . . . 198
5.4. Implementation . . . 198
5.4.1. Bytecode matching . . . 199
5.4.2. Abstract stack interpretation . . . 199
5.4.3. Regex matching . . . 200
5.4.4. Run-time generation of delegates . . . 200
5.5. Multiple customizations . . . 200
5.5.1. Adding an advice . . . 201
5.5.2. Further instrumentation . . . 202
5.5.3. Advice Composition . . . 203
5.6. Limitations . . . 203
5.7. Comparison with Other AOP Frameworks . . . 203
5.7.1. Discovery of join points . . . 203
5.7.2. Advices loaded statically and/or dynamically: . . . 204
5.7.3. Join point locations: . . . 204
5.7.4. Control flow . . . 204
5.7.5. Introductions, modifications: . . . 204
5.7.6. Runtime overhead . . . 204
5.7.7. Visualization of Join points: . . . 205
5.8. Related work . . . 205

6. Describing default rules, prescribing custom rules 207
6.1. Introduction . . . 207
6.2. Rule checking software . . . 208
6.3. Tools examined . . . 210
6.4. Describing default rules . . . 211
6.4.1. Flaw rules . . . 211
6.4.2. Style rules . . . 212
6.4.3. Documentation rules . . . 213
6.4.4. Design rules . . . 213
6.4.5. Implementation rules . . . 214
6.4.6. Code changes . . . 214
6.4.7. Reflections on the categorization . . . 216
6.5. Prescribing custom rules . . . 217
6.5.1. Code query by example . . . 217
6.5.2. First example . . . 218
6.5.3. Second example . . . 220
6.5.4. Third example . . . 221
6.5.5. Discussion regarding custom rules . . . 222
6.6. Related work . . . 223
6.7. Conclusions . . . 223

Bibliography 225

Bibliography 225

List of Figures

1.1. The upgrade problem . . . 3
1.2. The upgrade problem: further customizations . . . 5
1.3. Dynamics NAV: End-user interface . . . 12
1.4. Dynamics AX: End-user interface . . . 13
1.5. Upgrade . . . 42
1.6. Main internal dependencies . . . 52
1.7. Matching at the object level [204] . . . 54
1.8. Matching . . . 57
1.9. Eggther Add-in . . . 62

2.1. Further customization . . . 122
2.2. Resilience to kernel evolution . . . 122
2.3. Support for multiple customizations . . . 122

4.1. The upgrade problem . . . 172
4.2. Customization scheme . . . 184

5.1. Matching . . . 200

6.1. Flaws . . . 212
6.2. Categorization: style and documentation rules . . . 213
6.3. Code Changes . . . 215

List of Tables

1.1. Research outputs and research activities 9
1.2. Generated customization interfaces 34
1.3. Upgrade 43
1.4. Limitations summary 66
1.5. Claims and argumentation summary 103
1.6. Summary of customization technologies evaluated 104

2.1. The layers of a Dynamics AX application 116
2.2. Summary evaluation of customization technologies 143

3.1. Tasks of the survey respondents in their respective companies 150
3.2. Experience with the ERP systems considered 151
3.3. Who initiates an upgrade? 156
3.4. Reasons to upgrade 156
3.5. Most important factors that complicates an upgrade 159
3.6. Testing (multiple answers allowed) 161

6.1. Rule checking tools summary 210
6.2. Design rules 214
6.3. Implementation rules 215

Listings

1.1. CIL code example . . . 23
1.2. Query method . . . 31
1.3. Action query variables . . . 38
1.4. Target software product . . . 39
1.5. Target program, Func query variable . . . 39
1.6. Target program (try/catch) . . . 40
1.7. Query method expansion . . . 40
1.8. Anchoring . . . 41
1.9. Target program: Anchoring . . . 41
1.10. Target code before upgrade . . . 44
1.11. Query Transaction . . . 44
1.12. A customization: LogTransaction . . . 44
1.13. P^{n+1}, False positive: (7,8) . . . 46
1.14. A more concrete query . . . 48
1.15. P^{n+1}, False negatives: (2,3) and (11,12) . . . 49
1.16. Partial ordering of customizations . . . 65
1.17. Anonymous method (CIL) . . . 69
1.18. Anonymous method with captured variable (CIL) . . . 71
1.19. Try/Catch/Finally (CIL) . . . 77
1.20. Variance using delegates . . . 81
1.21. Diff and patch operations . . . 84
1.22. Template method pattern . . . 87
1.23. Strategy pattern . . . 88
1.24. Template method: Higher-order variant . . . 88
1.25. Factory pattern [125] . . . 90
1.26. Extract from the Visitor pattern [83] . . . 91

4.1. Disjunction of query methods 177
4.2. Example of customization 178

5.1. Target program 196

6.1. QuestionableBoolAssignment (FindBugs) [76] 219
6.2. QuestionableBooleanAssignment (CQE) 220
6.3. EmptyBlocks rule (Semmle) [180] 220
6.4. Empty conditionals, empty loops (CQE) 221
6.5. Performance rule (NDepend) [147] 221
6.6. Performance rule (CQE) 222

Preface

This dissertation contains two parts: the first part is our *thesis* chapter, the second part is a *collection of papers*.

The thesis chapter is self-contained, but makes many forward references to the next chapters. Although consistent with the next chapters, our approach has naturally slightly evolved along the years, the thesis chapter gives the most recent version. Two further applications of *code query by example*, namely *aspect-oriented programming* and *rule-based lightweight static analysis* are not discussed in the thesis chapter, but are addressed respectively in chapters 5 and 6. The thesis chapter has the following goals and structure:

- First, the introduction section presents the problem statement, the hypothesis, the research method, and gives an *abstract* presentation of the upgrade problem. Then, section 1.2 gives a quick introduction to Enterprise Resource Planning systems, and *instantiates* the abstract presentation of the upgrade problem in this industrial context. This section also summarizes what we consider the most important results from our *empirical study* (chapter 3).

- Section 1.3 presents the main *concepts* and gives *definitions* that will be useful for the rest of the dissertation. As discussed in our research method section (section 1.1.6), concepts and terminology are important components of *design science*.

- Section 1.4 introduces *modular decomposition*, and some of its limitations; it motivates our approach.

- Sections 1.5 and 1.6 give respectively a presentation of *code query by example* and its application to *customization* and *upgrade* of software products. This presentation is given in a more detailed and systematic way than what page limitations allowed for in the publications. The prototype implementation of *code query by example* is described in section 1.7.

- Section 1.8 explores some of the known limitations of *code query by example*, as well as limitations of the implementation.
- Section 1.9 discusses a few existing customization techniques, which were omitted in our survey (chapter 2), again due to lack of space.
- Section 1.10 reflects on the upgrade problem, on the results, and on lessons learned. Section 1.11 discusses related work. Finally, section 1.12 concludes.

The second part of this dissertation is a collection of papers that explores in further details the work presented in the thesis chapter. The papers are best read in sequence, but can also be read individually as they are self-contained. We give a very short summary of each paper, and of their respective contributions.

- "Technologies for Evolvable Software Products: The Conflict between Customizations and Evolution", P. Sestoft, S. Vaucouleur [182].
  - This paper introduces Microsoft Dynamics and evolvable software products in general. It gives a characterization of the upgrade problem, and reviews a number of customization techniques. Each technique is confronted against a set of well-defined criteria.
- "Customizations and upgrades of ERP systems: An Empirical Perspective", Y. Dittrich, S.Vaucouleur [58, 57, 59].
  - An empirical study of the practices around ERP systems. The empirical study is based on 3 different kinds of data: video recordings, an online survey, and face to face interviews with ERP professionals.
- "Customizable and Upgradable Enterprise Systems without the Crystal Ball Assumption", S. Vaucouleur [200, 201].
  - This paper introduces Code Query By Example and its use for customization and upgrade of ERP systems.
- "Aspect Oriented Programming made easy: An embedded pointcut language". A. Cisternino, S.Vaucouleur [40].
  - This paper describes Code Query By Example in the frame of a pointcut language for aspect oriented programming. Among other things, this paper describes bytecode instrumentation.
- "Describing Default Rules, Prescribing Custom Rules", A. Cisternino, R. Rasmussen, S. Vaucouleur [38].
  - This paper provides a taxonomy of the recent tools within the field of rule checking software, and describes how code query by example can be used as a language to prescribe custom rules.

Part I.

Thesis

1

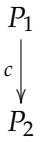
Customization and upgrade of enterprise systems

1.1. Introduction

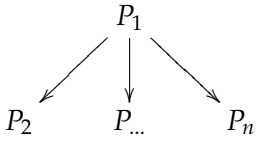
1.1.1. The upgrade problem, a teaser

In this section, we introduce the upgrade problem in a nutshell. We come back to the upgrade problem in details in section 1.2, and in chapter 2, in the more specific context of enterprise systems. The goal here is to give a succinct and abstract description of the upgrade problem.

Let P represents the set of all software products. Let P_1, with $P_1 \in P$, represents the initial version of a software product, conceived by programmer R_1. A directed edge from P_1 to P_2, denotes a change made on P_1 by an external programmer, called R_2. When it is useful for the discussion, we give a label to an edge; it allows us to give a *name* to the corresponding change. A change made to a software product by an external programmer is called a *customization*. This customization is represented by a *vertical* directed edge in the diagram below; in this case, the customization is called c:



Note, and this is important, that R_1 is not aware of the customization made by R_2, he only knows that customizations by external programmers are *likely* to take place. R_2 is not the only external programmer that customizes P_1; let $(n-1)$ be the number of external programmers, here we assume $n > 2$ (in practice, as we will see in the next chapters, n can be very large). In the diagram below, $P_{...}$ denotes a sequence of customized products.



Eventually, R_1 will change P_1 to produce a new version. We call the original version P_1^1, and the new version P_1^2. The horizontal directed edge in the diagram below represent this *evolution*

$$P_1^1 \longrightarrow P_1^2$$

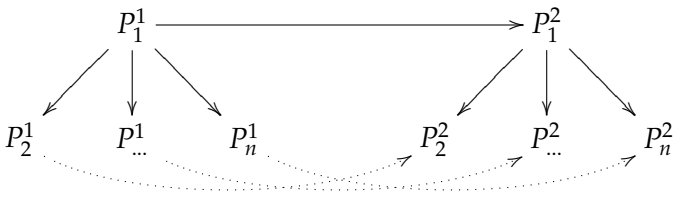
Note that evolution and customization are two forms of *software change*. The differentiating factor is that, in the former case, the change is made by the same programmer that last changed the software product; whereas in the latter case, the change is made by an external programmer. More precisely, given two products P_a^x and P_b^y, and a change c, such that:

$$P_a^x \xrightarrow{c} P_b^y$$

We will say that c is

$$\begin{cases} \text{an evolution} & \text{iff } a = b \text{ and } x \neq y \\ \text{a customization} & \text{iff } a \neq b \text{ and } x = y \end{cases}$$

Later, when it is clear from the context, we will assume that edges directed downward denote customizations and edges directed toward the right denote evolutions. Given the definition that we have stated above, the upgrade problem is represented in figure 1.1.

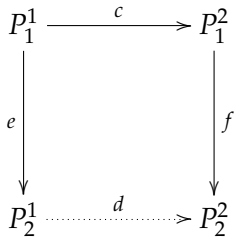


P_1^1 is the base software product, and P_1^2 is its new version after evolution. P_2^1, $P_{...}^1$, P_n^1 are customizations of P_1^1.

FIGURE 1.1.: THE UPGRADE PROBLEM

The set of *dashed* directed edges represents the core of the update problem. We assume that all the external programmers R_n (with $n \neq 1$), want to benefit from the latest version of software P_1^2. The challenge is that each of these programmers will have to adapt the respective changes that they made on P_1^1, to make them fit for P_1^2 (a process symbolized by a dashed edge). As we will see, the problem is actually more complex than this: for example, it might not be possible at all to port the change to the newer version; it might also be the case that the needed functionality that triggered the initial change is now available by default in P_1^2, in which case – arguably – the customization should not be ported at all.

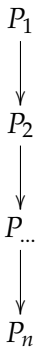
Let us focus, for a while, only on the programmers R_1 and R_2, and their respective set of software products $\{P_1^1, P_1^2\}$ and $\{P_2^1, P_2^2\}$. Consider the changes c, d, e and f according to the following diagram



The resulting diagram resembles commutative diagrams from category theory [161]. Informally, according to this diagram, we note that to go from P_1^1 to P_2^2, one can either first apply the evolution c followed by the customization f, or first apply the customization e followed by the evolution d. More formally, if we consider the changes c, d, e and f as functions that map a software product to an other software product, i.e., $c : P \rightarrow P$, we have $d \circ e = f \circ c$.

Making a connection with commutative diagrams can help to give a clear picture of the upgrade problem. It can also be misleading, in the sense that in the world of enterprise systems, c, d, e and f are not carefully designed together to obtain the required properties of this commutative diagram; the upgrade problem is an important, but it does not drive the way changes are made – the main drivers are more practical considerations, such as time to market, and scalability to a large number of external programmers.

Let us focus again on customizations, and generalize further the problem. Software products that are customized can be further customized by external programmers. For example, consider a software product P_1 made by a programmer R_1, customized by a programmer R_2 into P_2, further customized by an other programmer, etc. (The capacity to *further customize* is a *requirement* of the software products we consider, see section 1.2.2.)



We now consider the full set of programmers, consisting of R_1, the software maker of the original software product, and R_2, ..., R_n, the programmers that customized P_1^1. We can now give a more general representation of the upgrade problem, see figure 1.2.

Strictly speaking, figure 1.2 – as complicated as it already is – does not reflect the reality of customization and evolution of software products. We note in particular two points. First, in this diagram, only one software product is derived from P_2^1. In reality, an unbounded number of variants can always be derived from the same software product. (For example, still in figure 1.2, three software products are directly derived from P_1^1). Second, customized software products can also be derived from more than one software product, for instance:

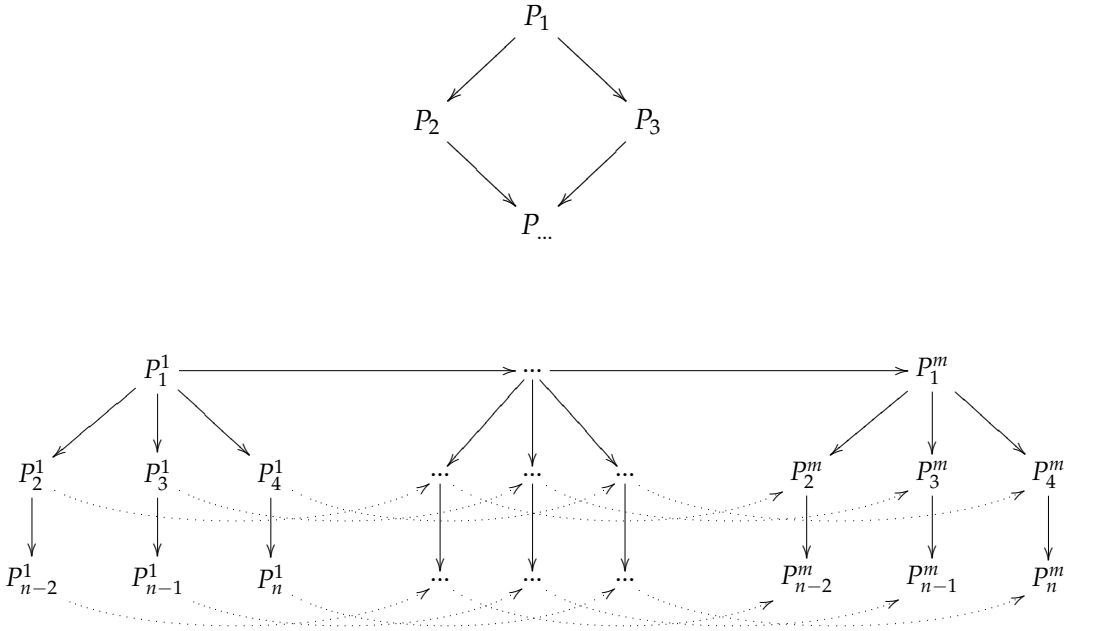


FIGURE 1.2.: THE UPGRADE PROBLEM: FURTHER CUSTOMIZATIONS

1.1.2. Problem statement

> *Software products*, such as enterprise systems, require *customization* to a particular context. Since software systems must evolve, customizations have to be ported on regular basis to a newer version of the base software product – a very costly task. This constitutes *the upgrade problem*. This thesis addresses the following questions. First, what are the characteristics of enterprise systems with respect to the upgrade problem? Second, given those characteristics, how can we make customizations amenable to upgrades?

1.1.3. Significance

There are many problems to tackle within the field of computing science, stating a problem is not enough to justify research on a particular subject – one must outline its *significance*. Furthermore, the reader might be foreign to the field of enterprise systems, such as Enterprise Resource Planning (ERP) systems, one of the object of study of our work. Hence, we want to cite a few figures early, to show that we are not tilting at windmills:

- Beatty and Williams state, in a recent Communication of the ACM dedicated to ERP systems, that *"Customizations that need to be carried over from one version of enterprise software to the next are the biggest technology headache and ROI*[1] *killer that CIOs*[2] *face in upgrades*" [24].
- AMR Research [9], a large consulting company, reports that the average cost of an ERP upgrade for a Fortune 500 company is $1.5 million [113].
- Meta Group [84], another large consulting company acquired by Gartner in 2005, surveyed 63 companies in 2002 about their average *total cost of ownership*[3] (TCO) for ERP software. Of the companies surveyed, the average amount spent was $15 million, while the lowest was $40,000 and the highest amount spent by one company was $300 million [114].

In addition to these existing results, we conducted an empirical study that confirmed the importance of the upgrade problem (see chapter 3). The upgrade problem is *a fact*, not just an hypothesis, and the problem is *significant*.

[1] Return on investment
[2] Chief information officers
[3] TCO includes not only upgrading, but also initial installation, staff training, and hardware maintenance up to 2 years after deployment.

1.1.4. Hypothesis (thesis statement)

> Using the concept of *code query by example*, software products can be customized, and a subset of the customizations can be ported to a new version of the base software product. Code query by example addresses three specific requirements of enterprise systems: (1) it is easy to use (2) it requires little anticipation and (3) it allows for incremental adoption. Code query by example is complementary to existing customization techniques.

1.1.5. Contributions

This thesis makes the following contributions:

- We provide a detailed and up-to-date description of the upgrade problem within the field of ERP systems (see for example sections 1.1.1, 1.2 and chapter 2).
- We provide an empirical study on the upgrade problem that pinpoints some of the characteristics of this problem within the field of ERP systems (see section 1.2.6 and chapter 3).
- We explore the concept of *code query by example* (see section 1.5), and we show how it can be used to deal with the upgrade problem (see section 1.6). We implement a subset of this language, using bytecode matching and bytecode instrumentation (see section 1.7), leveraging existing .Net technologies.
- We explore the limits of code query by example, and we relate it to existing work (see sections 1.8 and 1.11).
- We show how the concept of code query by example can be applied to two other domains: *aspect-oriented programming* (see chapter 5) and *rule-based lightweight code analysis* (see chapter 6).

1.1.6. Research method

Our research goes from the concrete towards the abstract. It is grounded, in the sense that it is motivated by a factual problem which was studied empirically. Finally, our research can be characterized in a design science framework. The object of this section is to describe our research method.

Concrete –> Abstract | Abstract –> Concrete We notice that there are two schools of thoughts in information technology research [96]. Some researchers *tend to* start from *the abstract* to go towards *the concrete* [4], while others favor the dual. More concretely the *abstract*, typically means a form of mathematical object, and the concrete means tools or concepts, that can be used by users and programmers in their daily tasks. We do not claim any clear dichotomy between the two schools of thoughts, the

interplay between abstract models and concrete tools is common within computing science – nonetheless, the distinction is useful. The work we present here *starts from the concrete*, meaning that we ground our research in an industrial context. We then look for *abstractions* that can be used to solve the problems in hands. Sometimes, like in the previous section, we try to give an abstract description of the problem in hand. In which case, typically, the goal is not to use the abstract model to reach a solution but simply to convey a particular point more rigorously.

Grounded research Our work is grounded [171] in the study of the use of two modern industrial ERP systems: *Dynamics AX* and *Dynamics NAV*, called collectively *Dynamics*. Understanding of the domain was made possible thanks to a collaboration with the Microsoft Development Center Copenhagen. Concretely, collaboration with our industrial partner brought us four important assets: (1) we were kindly invited to join a development team on the Microsoft campus for a short period, that allowed us to understand quickly some of the technical details of ERP system from the inside (the build process of Dynamics AX was an interesting experience in itself). (2) we were given video recordings (screen-casts) of experienced ERP programmers in their daily tasks. (3) we joined a Dynamics conference, which brought us some important contacts for our empirical work (4) we had meetings with the Dynamics management team.

Design science Information technology is usually approached from two angles: *natural science* and *design science* [123]. Natural science is concerned with "explaining how and why things are" [123] – it is mostly *descriptive*. Design science seeks to "extend the boundaries of human and organizational capabilities by creating new and innovative artifacts" [94] – it is mostly *prescriptive*. Furthermore, whereas natural science focus on *truth*, design science focus on *utility*. In this sense, our work belongs to design science.

March et al. propose a framework to characterize information technology research [123]. This framework has two dimensions: research outputs and research activities. Among research activities, March et al. distinguish *build*, *evaluate*, *theorize*, and *justify*. Among research outputs, they distinguish *constructs*, *model*, *method* and *instantiation*.

- *Constructs* form a vocabulary and a conceptualization of the problem domain. This is the goal of sections 1.1, 1.2, and 1.3. Chapters 2 and 3 also address concepts.
- A *model* is a set of propositions expressing relationships among constructs (March al. distinguish the term *model* from the term *theory*). The primary concern of a model is its usefulness for designing an information system [123]. Our model is described in section 1.5, as well in subsections of our collection of papers, for example sections 5.3 and 6.5.1.
- A *method*, according to Match et al., is a set of steps used to perform a task.

Table 1.1.: RESEARCH OUTPUTS AND RESEARCH ACTIVITIES

	Build	***Evaluate***
Constructs	Section 1.1, 1.2, 1.3	Chapter 3
Model	Sections 1.5, 5.3	Chapters 5, 6
Method	Sections 1.5, 1.6	Section 1.8.1
Instantiation	Section 1.7	Section 1.8.2

There are two tasks in our case: customization and upgrade. Customization is described in section 1.5; upgrade is described in section 1.6.

- An *instantiation* is the realization of an artifact, this is the goal of our prototype, described in section 1.7.

The focus of our work was the *build* activity: we build concepts, a model for customization, a method to apply customization and upgrade, and a instantiation through our prototype. Our work on relating code query by example to aspect-oriented programming, chapter 5, and to do rule-based lightweight static analysis, see chapter 6, can be seen as *evaluation* of the method. Limitations of the method is discussed in section 1.8.1. *Evaluation* of the instantiation is discussed in length in section 1.8.2, where we discuss the limitation of the prototype. What is missing is an evaluation of code query by example in an industrial context, where customizations are performed on systems in production [94]. According to March et al. [123], this would allow to build *theories* to explain the interaction of the artifact with the industrial environment, and as well as *justifications* of such theories. Table 1.1 summarizes the research outputs and the research activities.

1.2. Overview of enterprise systems

In this section, we recall some of the key concepts behind ERP systems, focusing more particularly on Dynamics AX [137] and Dynamics NAV [138, 144], two ERP systems from Microsoft. We refer the reader to sections 2.3, 3.2, and 4.1.1 for a more complete treatment. For short, we will refer to the Dynamics products (Dynamics AX and/or Dynamics NAV) as *Dynamics*, or *Microsoft Dynamics*. Similarly, the term *Dynamics developers* will refer to the core Dynamics development teams at Microsoft. Note also that while we tend to use the terms *ERP systems* and *enterprise systems* as synonymous, some might consider that the latter as a more general term.

1.2.1. What are ERP systems?

ERP systems are usually defined as being business support systems that try to integrate the various functions found in modern companies, such as *manufacturing*, *finance*, *human resources* and *customer relationship management* [183]. While there is a trend to give better support for processes, ERP systems are still heavily data-oriented. Hence, the back-end database is typically seen as the central element of the infrastructure. Business data is stored in the ERP system, and is kept preciously for many years, for business and legal reasons. Concretely, users typically enter raw data in the system through *forms*, and have access to aggregated data through *reporting*. Obviously, modern ERP systems also put an emphasis on integration with the outside world, for example using web services and various other modern communication protocols.

Example of functionalities Access right management is an example of functionality that should be implemented by ERP systems: for instance, while it might be a bad idea to give to all users access to salary data, a subset of the employees (accountants, managers, etc.) should be given that right. Another example of functionality, is aggregation of data: decision makers are continuously provided with a clear summary of the company's activities and results, and can therefore make informed judgments with respect to critical business decisions.

General model Concerning the general model of ERP system, so far, the basic trend has been to transpose the old paper based systems, such as the traditional *double-entry bookkeeping* in accounting (an important part of an ERP system functionality) to a computerized version. Note that some experts in the area of ERP systems challenge this approach and propose new models, for example the REA community [98] (see the related work section, section 1.11).

1.2.2. Customization of ERP systems

Once an ERP system is deployed, it is fully functional and can be used as is. Nonetheless, many companies need to customize their newly acquired ERP system to the

local context in which it will be deployed [113, 183, 24]. The local context can be for example related to the company's unique business model, or to some local regulations. An alternative is to adapt the company to the ERP system – which does happen in practice – but of course, this is not always feasible nor economically viable [183].

Partners' network Customizations can be made directly by customers, but usually they are done by small software houses that specialize in this activity, which we will call *partners*. The competencies of partners consist in their knowledge of the ERP system, but most importantly in their knowledge of the *vertical domains* (accountancy, transport industry, etc.). Their mastery of both the ERP system and of the vertical domains allow them to quickly develop customizations that fit the needs of their customers, see section 2.3.3. They typically charge high fees for their services, hence time-to-market is important to the customers.

Back to the upgrade problem We invite the reader to refer back to the figure 1.2. The goal of this paragraph is to *instantiate* the abstract description of the upgrade problem defined in section 1.1.1.

The base software maker R_1, in charge of the software products P_1^1, $P_1^{\cdots}$, P_1^m, is in this case Microsoft. Microsoft gives to partners a large subset of the software product in source code form. Furthermore, to make the connection with figure 1.2 more explicit, we note that:

- Down arrows in figure 1.2 denote customization on Dynamics products.
- What we called *external programmers* in section 1.1.1 are, simply, partners.
- Dotted arrows denote the porting of customizations on Dynamics products to a newer version of the base software product.

Major versions of Dynamics are now released every 2 years approximately. For instance, here is the recent history of AX (previously named *Axapta*):

- October 2002: Axapta 3.0
- March 2006: Dynamics AX 4.0
- June 2008: Dynamics AX 2009
- Summer 2010: Dynamics AX 2011 (planned release)

1.2.3. Microsoft Dynamics

Dynamics consists of *a runtime environment*, *a database system*, *a development environment* and a number of *core packages*, e.g., for sales tax reporting in a particular country [190, 88]. Both AX and NAV are partially *model-driven* and partially *programming language based*. Namely, database tables, runtime data structures, and the user interface (forms)

are described by meta-data, not built using programming language declarations. On the other hand, behavior is described using traditional programming language constructs, called code units, which correspond to functions or methods.

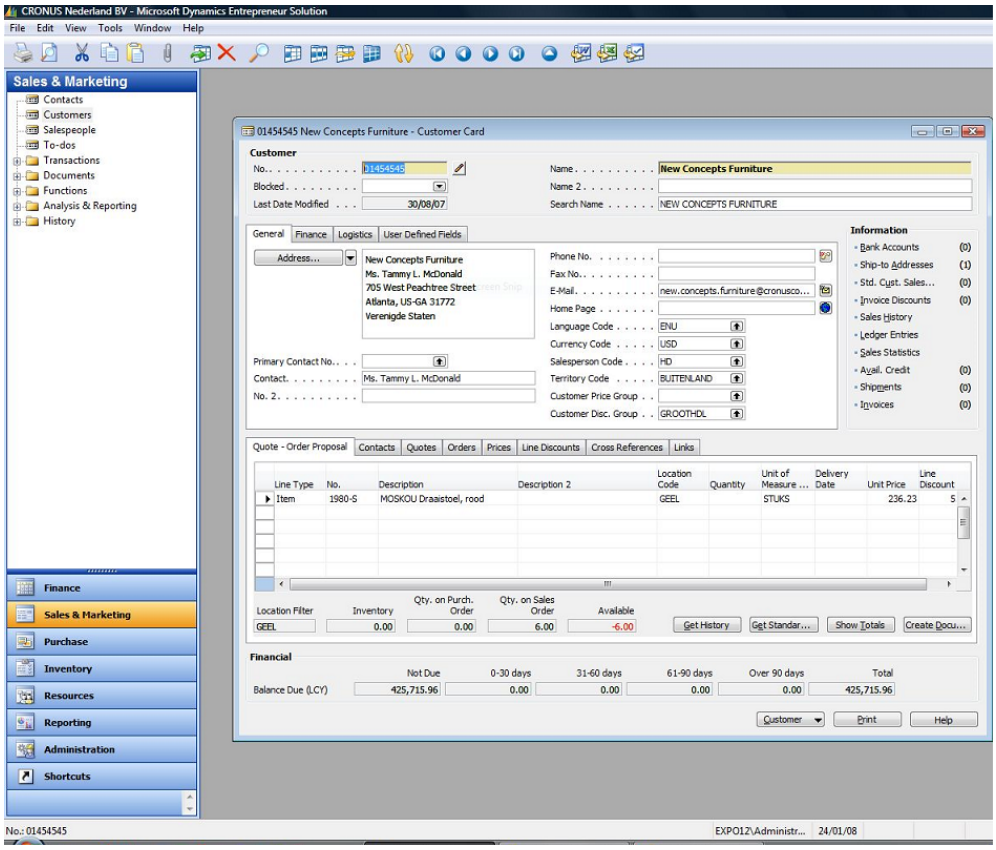


FIGURE 1.3.: DYNAMICS NAV: END-USER INTERFACE

1.2.4. Dynamics NAV versus Dynamics AX

The two systems have distinct organizational and technical characteristics, hence it is interesting to study both of them.

- Dynamics NAV, see figure 1.3, mostly targets smaller organizations, for which pre-developed add-ons mostly suffice, so they only require minor customizations. A large number of organizations run Dynamics NAV. The integrated development environment is called *C/SIDE*, and the programming language, C/AL, is a relatively simple language with a Pascal-like syntax. The developers employed by NAV partners usually focus on the customer's business and many do not have a strong background in software development. Code unit

customizations are made simply by editing the required code units in the C/AL language [190].

- Dynamics AX, see figure 1.4, mostly targets larger and more complex organizations, that often require extensive customizations. Fewer organizations use Dynamics AX than NAV. The integrated development environment is called MorphX, and its proprietary programming language X++ is an object-oriented language with a Java-like syntax [88]. The Dynamics AX model is structured into a number of layers, with layers for the kernel, layers for partners' customizations, layers for further customizations in the end-user organization, and so on; see section 2.3.8. A code unit customization is made by copying the code unit from the layer at which it was originally defined and then adding and editing at a higher layer. The higher layer version will then be used instead, and is said to shadow the lower layer code unit; see section 2.3.9.

We refer the reader to chapter 2 for a treatment of customizations in NAV and AX.

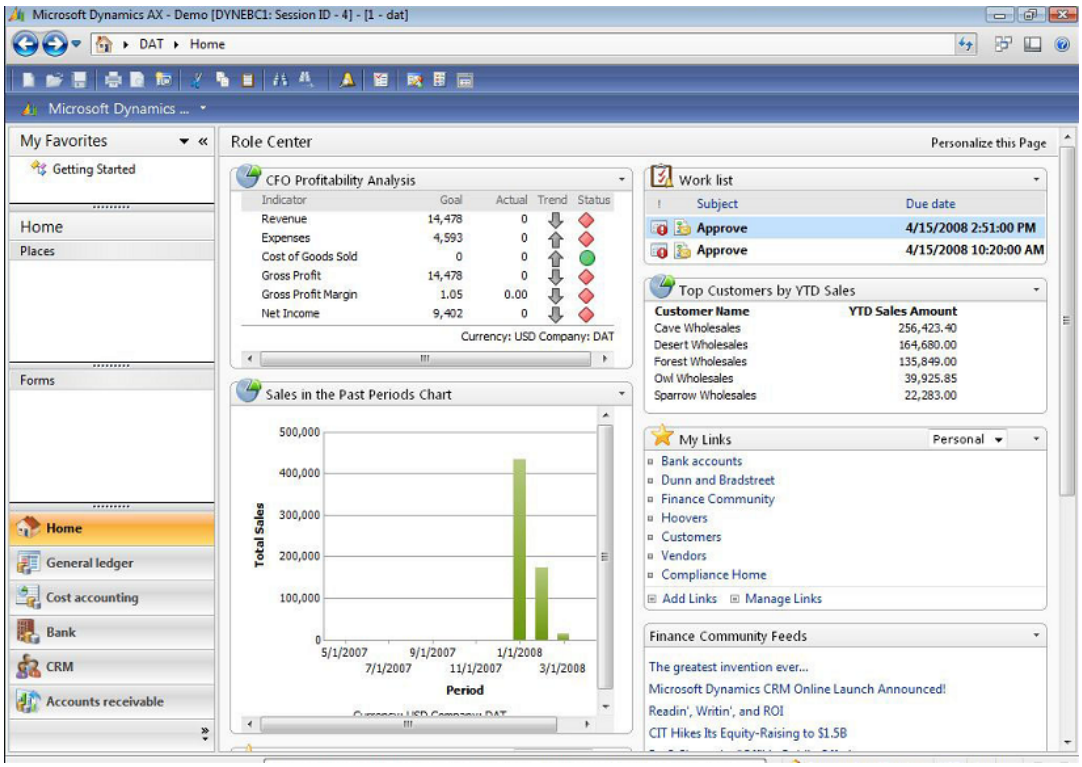


FIGURE 1.4.: DYNAMICS AX: END-USER INTERFACE

1.2.5. Summary of Dynamics

Dynamics AX and NAV are ERP systems developed over a long time and sold in many copies, with a wide range of customizations, to many different customers – they are software products. They also exhibit the upgrade problem outlined in section 1.1.1, in the following way: the add-ons and customizations are developed primarily by partner companies, whereas the core system evolution is controlled primarily by the Dynamics programmers.

1.2.6. Empirical grounds

We performed an empirical study to get an understanding of the challenges and practical problems faced by ERP practitioners, see chapter 3. The scope of this study was confined to ERPs partners, and to Dynamics. The goal of this section is to summarize the approach we took, and to shortlist what we consider as the most interesting empirical results.

1.2.6.1. Empirical research method

We used *grounded theory* for our empirical study. Grounded theory is particularly relevant when researchers want to use a structured way to find a proper set of hypothesis from collected data [171]. The theory is said to be *grounded* in the data obtained during the empirical study. The study that we performed was mostly *qualitative. Quantitative* approaches are more amenable to statistical reasoning, but on the other hand, qualitative approaches are more appropriate to study certain properties related to experiences, such as *satisfaction*, which are by nature difficult to measure [85]. Note that confidence in our results is increased by the fact that we used three different kinds of data, from three different sources – a process sometimes called *triangulation* [171] – and all of them led to the same conclusions.

1.2.6.2. Data sources

The data on which we based our study came from the following sources:

- The user experience group at the Microsoft Development Center Copenhagen provided us with *video recordings* (screen castings that include voice comments) of partners performing customizations.
- We performed *face-to-face interviews* with 3 partners.
- We performed an *online survey* that was filled in by 42 partners.

1.2.6.3. Process

The empirical research was done in three steps:

- First, we reviewed the video recordings and discussed the issues around customization and upgrade with our industrial partner, Microsoft.
- Then, based on our initial knowledge, we devised an online survey targeted toward ERP practitioners.
- Finally, we conducted face-to-face interviews with three IT managers working within ERP focused companies. To support the interviews, an interview guideline was conceived. Its role was to drive the question and answer sessions. This guideline was tested by performing a test interview on a senior Microsoft engineer, well-experienced with ERP systems.

Section 3.3 gives additional details concerning the collection of data, for instance concerning the structure of the interviews. Section 3.3also comments on the validity of the findings.

1.2.6.4. Challenges

Conducting realistic empirical research on ERP systems is difficult for the following reasons:

- Partners are located on very distant geographic areas, which restrict one's capacity to conduct face-to-face interviews (these interviews were preferred to conversations over the phone or via email exchanges, since they are more likely to foster communication).
- ERP practioners have little incentive in participating in an empirical study.
- Non-disclosure agreements, and other legal obstacles linked with intellectual property rights, hinder empirical research.

1.2.6.5. Results of the empirical study

Results concerning the empirical study are described in details in chapter 3 on page 145. Hence, we will only give in this section a short summary of those results, sorted according to the main themes of our study.

Programmer's profile Programmers working for partners tend to have little formal computer science education. Partners typically try to make teams of two persons: a domain expert, and a more technically-minded person. When they have to make a choice (when only one person can be hired, etc.), partners tend to give priority to the domain expert.

Knowledge of the base software product Programmers are faced with very large code base. Even experienced practitioners do not have a complete understanding of their favorite ERP systems. ERP systems are poorly documented; knowledge is mainly acquired by trial and error, peer learning, and by looking at code examples of existing customizations.

Range of customizations and tool support Current development environments are geared towards rapid prototyping. Partners perform a wide range of customization: from simple changes, such as hiding a field in a form, all the way to very complex customizations that change various parts of the system. Customizations are sometimes made directly to the running system.

Testing No regression tests (nor unit tests), are given to the partners with Dynamics; partners have little guarantees with respect to the correctness of their customizations. Testing of customizations is typically confined to performing a few sanity checks: programmers just enter some data in the systems, and quickly check if the customization *seems* to work as expected. Customers are typically not willing to pay for rigorous systematic testing, except for a few exceptions concerning mission critical systems.

Reasons for upgrades The main reason for upgrade is to benefit from the latest bug fixes. Partners also fear that if a release is skipped, it will be harder to upgrade latter. Upgrade is considered mandatory by most of the partners.

Upgrade process Not all existing customizations are upgraded: customizations that have been little used and customizations that are now directly supported by the ERP systems are candidates for removal. Partners try to avoid adding new features during an upgrade: new features are treated as an independent project once the upgrade is done.

Challenges of upgrades The absence of a good documentation is seen as a critical problem: there is typically very little information available when partners try to understand old customizations. In the best case, upgrades are done by the same partner who did the customizations, using existing documentation on customizations. In the worse case, there is no documentation with respect to existing customization, and the upgrade is done by another partner.

Partners' expertise Experienced developers can make the best use of the functionalities provided by the standard system. An experienced developer knows the consequences of his customizations. Expertise depends on the knowledge of the base system – but with new releases this knowledge quickly becomes obsolete. Finally, experienced programmers avoid invasive changes as much as possible.

Factors that complicate an upgrade Missing information about the existing customizations is considered the most important factor that complicates an upgrade (often, knowledge is lost when employees leave the company). Poorly structured customizations hamper upgrades; customizations made to kernel code are more difficult to upgrade; changes to form designs are reportedly time consuming to upgrade. Finally, there is little support for fine-grained upgrades.

1.2.6.6. Summary of the empirical study

Missing knowledge about existing customizations is perceived by the ERP practioners as the most important factor that hamper upgrade. Intrusive customizations are perceived as problematic, experienced partners avoid them as much as possible as they hinder upgrade. Often, correctness is not considered a major issue, partners are more concerned with time-to-market. Knowledge about ERP systems is acquired mainly through trial and error, working with experienced colleagues, and looking at code example of previous customizations.

1.3. Concepts

The goal of this section is to define succinctly three sets of concepts that will be useful for the rest of the discussion: concepts related to object-oriented programming, concept related to virtual machines, and finally concepts related to software customization.

1.3.1. Concepts related to Object-Oriented Programming

To the best of our knowledge, most large enterprise systems, such as ERP systems, are now based on an object-oriented language, and arguably for good reasons. For example, the capacity to easily add new types is an important requirement of enterprise systems [183, 88], and a core asset of *object-oriented* (OO) *programming*. Furthermore ERP systems are by nature stateful [183], and OO languages are typically state-oriented. In this section, for simplicity, we stick to the tradition of using the term *object-oriented programming*, but we prefer the term *class-oriented programming*, since what is being programmed are really classes and not objects [4]. When necessary, to avoid confusion, we will use the term *class-based languages*. Except when mentioned explicitly, we give our definitions in the context of C# [65] (There are a wide range of variants for the concepts that we introduce; whenever relevant for our context, we do our best to mention some of the important alternatives.)

Organizational structures From a high level point of view, we consider three main organizational structures [135]: assemblies, namespaces, and types. Namespaces contain types, and can be defined across several assemblies. An assembly usually takes the form of a file, but not necessarily so. To access types available in assembly, one must *reference* the given assembly. Both how an assembly is referenced, and how an assembly is resolved given an assembly reference, have important impact on evolution, this will be further discussed in section 1.9.4.

Types and variables Types usually have names (the notable exception in recent versions of C# are *anonymous types*). In C#, types are either value types, such as `int` and `bool`, or reference types, such as `object` and `string` [66, page 17]. Variable of value types directly contain their data, whereas variables of reference types store a reference to their data – a reference to an object. An important consequence is that operations on one variable will impact an other variable if they reference the same object. (This is not the case with value types, with the exception of `ref` and `out` parameter variables; we will discuss `ref` variables further down in this section.) All types inherit from the root `object` type. Types can be either primitive types such as `float` and `int`, or programmer-defined types, such as `Octagon` and `Triangle`. There are various ways to declare new types in C#: using *interfaces*, using *classes*,

[4] Note that there are exception to this, such as the language Self [197], where the focus are the objects.

using *delegates*, using *structs*, and using *enums* [65]. We will only focus on the first three.

Interfaces An interface *specifies* a set of *members*, but cannot provide the implementation for those members. Concrete classes (or structs) that *implement* an interface *must* provide an implementation of the members defined by that interface [65]. Classes, or structs, can inherit from multiple interfaces; an interface itself can inherit from other interfaces. For example,

```
interface I { string P {get;}}
```

defines a type named `I`, and specifies that all values of this type must support the getter property `P` of static type `string`.

Classes There are two dual approaches to introduce the notion of class: one can say that a class is the description of a set of objects [1, page 11], or dually, one can say that classes are templates to generate objects [32, page 18]. The former is arguably closer to formal type theory, and the latter is somewhat closer to the actual implementation of modern execution framework based on object-oriented languages. A class has a structure and a name. The structure contains a set of *members*. An instance of a class gives rise to an object. An *abstract* class can defer the definition of some of its members to subclasses, and cannot be used directly to generate objects [65, page 264].

Delegates A *delegate* is a reference to a method, with a specified formal parameters list, and a return type. The .Net framework already defines some commonly used delegate types: among others, `Action` is a delegate type that represents procedures, and `Func<R>` is a delegate type that represents functions that return a value of type `R` [135]. Delegates are convenient, since they allow programmers to assign methods to variables, to pass methods as arguments, etc. Methods that take delegates as parameters, and methods that return delegates, are commonly called *higher-order methods* [3, page 56]. Unlike function pointers that are typically found in lower level programming languages, delegates are type safe [65, page 67].

Parametric polymorphism Classes, interfaces, structs and methods can be *parameterized* with other types [65, page 52]. For example, `class Cell<T> {...}` introduces a type which contains a *formal generic parameter* `T` [134, page 318]. The formal generic parameter can, and sometimes must, be instantiated, as shown in field declaration: `Cell<int> f`. We will say that, in this case, `int` is the *actual* generic parameter [134, page 321]. Similarly, methods can be parameterized, for example, `void Swap<T>( ref T a, ref T b){...}`. In order to be invoked, the generic type parameters of the method `Swap` must be instantiated: this is sometimes done implicitly when the actual type parameter can be inferred by the compiler, or explicitly otherwise, such as `Swap<Matrix>(ref m1, ref m2);` which will swap the value of the variables `m1` and `m2`, where both of them hold a reference to an object of type `Matrix`.

Type systems Type systems and type safety are an important part of the object-oriented language (OOL) that we consider [1, 32]. Whenever we write the statement `x.F();` we want to ensure that `F` will be defined at run-time; as a further example, in the last paragraph, the definition of the `Swap` method ensures that only objects of the same type can be swapped. Types provide a form of documentation, and sometimes allow for compiler or runtime optimizations. More importantly, they allow for a form of abstraction: the abstraction is defined by the name given to the type and its structure. In C# type checking is done (mostly) statically, giving rise to a statically typed language. Some operations such as *casting* [65], cannot always be checked statically, and are hence usually avoided by programmers. Generics, as touched upon in the previous paragraph, provide a form of polymorphism called *parametric polymorphism* [35]. Parametric polymorphism is particularly useful in order to avoid risky casting operation, for example when designing a collection library [65, page 53].

Objects The statement **`new`** `C()` constructs from the class C an object. Upon creation of this object, a special method called a *constructor* is invoked. Semi-formal approaches to OOL claim that the primary role of the constructor is to establish *class invariants* [134]. Invariants are not directly available in C# (as language constructs), but can be expressed using Spec# [18], a variant of C#; a recent version of .Net, version 4.0, also allows to specify contracts and invariants in a language agnostic way through the use of the `System.Diagnostics.Contracts` namespace [135].

Members Classes contain members. Important class members are methods and fields. Members fall into two groups: static members and instance members. Static members belong to classes, and instance members belong to objects [65].

- Methods have a *signature*: a name, a sequence of *formal parameters*, and a number of type parameters. In addition to a signature, a method has a *return type*. When a method returns a result, we will use the term *function*, when it does not, we will use the term *procedure*. Within methods, a special identifier, called **`this`**, refers to the hosts object. In C#, like in most OOLs, it is possible to omit the identifier **`this`** when referring to instance members for the current object.
- A field is a variable that is associated with a class or with an instance of a class.

The visibility of members can be restricted using an access modifier that annotates the definition of members. The .Net framework has for example the *public*, *protected*, *internal* and *private* modifiers. (*Protected* limits access to the current class and to classes derived from the current class. *Internal* limits access within an assembly.) Access modifiers allow for a form of representation hiding. Subtyping allows for another form of representation hiding [1, 32].

Method invocation Given a variable called v, one can access a member of an object using the dot notation: for example, the statement v.F(); will invoke the method F,

where F is defined by the class of the object reference by v. We will say that that the class that invokes F is the *client*, whereas the class that defines F is called the *supplier*, or the *receiver*. During method invocation the proper method definition has to be *looked-up* – this in itself is a complex problem [1, 32].

Reference parameters A *reference parameter* is a parameter declared with a `ref` modifier. "Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation." [65]. We can illustrate reference parameters by giving an implementation of the aforementioned `Swap` method:

```
static void Swap<T>(ref T m1, ref T m2) {
 T temp = m1;
 m1 = m2;
 m2 = temp;
}
```

Inheritance When a class B inherits from a class A, B can make some incremental changes to A: new members can be added [1, 32]. We say that B is a subclass of A, and that A is a superclass of B. B can also redefine methods of A that were marked virtual. We say that the new definition *overrides* the existing one. Class designers specify which method can be overridden, through their usage of the *virtual* modifier [65, page 292]. This is obviously an important decision in term of evolution and customization: marking all methods of a class A as virtual will give a lot of flexibility to the person that implements B, the subclass of A, but it also makes it more difficult for the maker of A to reason about the behavior of A. Dually, not using the virtual keyword makes the task of the designer of A easier, but limits what changes can be made by the maker of B [47].

1.3.2. Concepts related to modern managed execution environments

Virtual machines and *managed execution environments* are now commonly used within enterprise systems; Microsoft Dynamics builds on the .NET framework [136], while others, like SAP [177], build on the Java framework. The implementation of *code query by example*, described in section 1.7, builds on .Net bytecode [66], and on other concepts related to modern virtual machines. We shortly discuss managed code execution and modern virtual machines in the context of the .Net framework. The subject is much too broad to be covered in details, we shall only touch upon the basics of the .Net framework and discuss some particular points that will be useful for the rest of the discussion. One typically distinguish three core components in modern frameworks [87, 135, 66]:

1. An *execution environment*.

2. A large set of *base classes*, containing not only core types such as `int` or `string` but also convenient classes for the most common programmers needs, such as file handling, networking, graphical forms and widgets, etc.

3. A set of tools, such as compilers for high level languages.

In .Net, the execution environment is the role of the *Common Language Infrastructure* (CLI). The *Common Language Runtime* (CLR) is an implementation of the CLI by Microsoft. A major component of CLI is the *Common Type System* (CTS), which allows programs written in different high level languages to interact in a type safe manner [66]. The main role of the CLR is to execute bytecode. Bytecodes are expressed in the *Common Intermediate Language* (CIL). High level compiler, such as *csc,* the Microsoft C# compiler, compiles high level code into CIL (a stage called *compile time*). The CIL bytecode is then executed by the CLR (a stage called *runtime*). At runtime, the CIL code is *compiled* into native code, code which is specific to the operating system. The CLI has been implemented on a variety of operating systems, see for example the Mono project [142] for an implementation that supports the Linux operating system.

Abstract Machine The execution state of a method is maintained with an *evaluation stack*, and an *activation record* [87, page 24]:

- The evaluation stack contains values: data is pushed into the stack and popped from the stack.
- An activation record contains incoming arguments and local variables (when a method is called, an activation record is created.)

CIL Instruction The CIL contains more than 200 instructions; we refer the reader to the Ecma International standard for an exhaustive description [66]. Each instruction has a label, an opcode and optionally an operand. There are 3 main categories of instructions [87]:

- Instructions that *push* values onto the stack.
- Instructions that *perform operations* on values that are on the stack.
- Instruction that *pop* values from the stack.

A minimalist example As a simple example to illustrate the execution of a sequence of bytecodes, we consider the C# expression statement: `x++;` where `x` is of type `int`. Using csc, , the Microsoft C# compiler, this expression statement is translated into bytecode as shown in listing 1.1.

LISTING 1.1: CIL CODE EXAMPLE

```
.locals init ([0] int32 x) // declares a 32-bit signed integer
...              // instructions before the statement x++
L_0003: ldloc.0  // loads the variable x onto the evaluation stack
L_0004: ldc.i4.1 // pushes 1 onto the evaluation stack
L_0005: add      // adds the two values and pushes the result
L_0006: stloc.0  // Pops the top of the stack, stores value into x
```

Stack Following the previous example, the stack:

- Starts empty.
- Is at height 1, after the instruction ldloc; the top of the stack contains the value of `x`.
- Is at height 2, after the instruction ldc.i4; the top of the stack contains `1`.
- Is at height 1, after the instruction add; the top of the stack contains `x+1`.
- Is at height 0, after the instuction stloc.

At this point, the variable `x` will contain the expected value.

Method calls We will not illustrate method calls at the bytecode level, we refer the reader to the Ecma standard [66] or to a textbook on CIL [87]. For our purpose, it is sufficient to say that method calls take place in a very similar manner to what we describe in the previous paragraph: arguments for the method call are pushed onto the stack in the order defined by the formal parameters, and a call instruction is executed (in the case of a static method) [87]. The call instuction will pop all of the arguments from the evaluation stack.

1.3.3. Concepts related to software customization

Software products A software product is software that is typically highly customizable to permit effective use in many different applications and contexts. Successful software products are released in many versions over many years. Software products should be contrasted with software that has been developed in a project for a particular purpose, such as the software used by tax authorities in a particular country, see chapter 2.

Customization versus configuration We differentiate *customization,* which can add new and possibly unforeseen features to software, from *configuration*, which enables or disables features that are already present in the software, see chapter 2.

Customization versus adaptation The terms *customization* and *adaptation* are sometimes used interchangeably in the literature. Nonetheless, the term *adaptation* tends to be used in contexts where software change happens dynamically, at run-time. Since we mostly focus on design-time changes, we use the term *customization*.

Reuse Our work can be seen as a continuation of a long tradition in the software engineering community to promote *reuse*. See for example the survey on software reuse by Krueger [116] or the one by Mili et al. [141]. One can see the software products being *reused* and *customized* for a particular context. To the best of our knowledge, the only differentiating factor with existing work that frame themselves in the context of reuse, is the size of the unit which is being reused. Traditionally, reuse deals with classes, components, or libraries. The software products that are "reused" in our work typically have several million lines of code. Given well-known *modularity techniques*, it is arguable whether this distinction matters, see section 1.4.

Add-ins, plug-ins, etc. Various terms in the literature relate to very similar concepts around the theme of software extension: *add-ins*, *add-ons*, *plug-ins*, *extensions*, etc. *Add-in* and *add-on* are, currently, two of the most popular terms. *Add-in* emphasizes that one is adding functionality *inside* the host program, while *add-on* emphasizes that one is building *on top* of the host program. For example, the Visual Studio community talks about *add-ins* [135] (see, for instance, our Eggther add-in in section 1.7.3), while the Firefox community talks about *add-on* [145]. The Eclipse community talks about *plug-in*, etc [64]. While some try to find some subtle distinctions between those terms, there is in practice – and to the best of our knowledge – very little difference. The fundamental ideas are:

- A host application provides most of the required services that are used by add-in/plug-in, etc.
- The host application can work by itself.
- Add-in/plug-in can have dependencies to other add-in/plug-ins.
- In some cases, add-in/plug-in execution environment is isolated from the host environment. The goal is to try to avoid failure of the host in case of failure of the add-in/plug-in.
- Obviously, there is an API that add-in/plug-in programmers can use to access the host application.

Software product customization versus Add-ons Microsoft Dynamics (which is introduced in section 1.2) uses the term *add-on* to denote a set of customizations targeted to a particular industry (such as textile) or to a particular activity (such as customer relationship management) [136]. One should note that in the context of Microsoft Dynamics, the term add-on has a different meaning: customizations

typically change part of the source code of the host program, whereas, outside of the Dynamics world, the term add-on is usually used in context where the code source of the host program is not modified.

Quantification and obliviousness Quantification and obliviousness are two concepts on which we build part of our work. According to Filman, "Quantification is the idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system; obliviousness, that the places these quantifications applied did not have to be specifically prepared to receive these enhancements." [73, page 1]. In our work, we use static quantification [75, page 27]: quantification over the static structure of the program. To the best of our knowledge, those terms were originally defined in the context of aspect-oriented programming [74]. We frame our work in the context of aspect-oriented programming in chapter 5.

1.4. Modular decomposition

In section 1.5, we discuss in details our approach. In this section, we proceed with a detour: we discuss *modularity*; the concept is important to explain our thesis, and how we depart from usual software engineering approaches.

1.4.1. Local reasoning and modular programming

An intuitive way to introduce modules, is to first approach the concept of *local reasoning* [95]. It is well-known that it is difficult for programmers to reason about the effects of a large program. To deal with this problem, one can apply a *divide-and-conquer* strategy, where the program is no longer constructed as a large uniform block, but is rather *composed* of a number of individual building blocks. The goal, is that the programmer should be able to *reason* about a certain code block independently, meaning without having to look at the other blocks [92]. By reasoning, we mean being able to assert certain property of the software (before execution). *"The return value of function f must be positive"* is a *safety* property (expressed in natural language). *"Execution of function f will eventually terminate"* is a *liveness* property [117, 118, 179, 2].

Decomposition Informally, the designer must decide how many building blocks will be present and what will be the responsibility of each building block – in other words, the programmer must decide upon the *decomposition* of the software [158]. We think that decomposition is at the core of the act of *designing*.

Modules, interfaces, and information hiding Following the standard software engineering methodology, each building blocks will have an explicit *interface*. A building block delimited by an interface is called a *module*. The interface hides some details of the implementation [92].

Composition Once the software is decomposed, it can be constructed by *composing* building blocks together [2]. This creates dependencies between *client* modules and *supplier* modules. We say that a module A is a client of module B if A *uses* the module B [158]. Typically, the client module will invoke a function whose implementation is within the supplier module.

Top-down versus bottom-up Software construction methodologies that focus on decomposition are usually called *top-down*. Software construction methodologies that focus on composition are usually called *bottom-up* [141, page 7]. In practice, software construction methodologies tend to mix top-down and bottom-up approaches.

1.4.2. Decomposition à la Parnas

Criteria for modular decomposition One of the major contribution of Parnas was to give *a set of criteria* for the decomposition into modules [158, 159]. The main

criterion, which also happens to be the most relevant for us, is that the set of program fragments that are *likely* to change together should reside within the same module [158, 159]. Note the emphasis on "likely".

Connection with evolution The obvious connection with software evolution, is that since the clients of a module only deal with its interface, one *hope* that the implementation can be changed without impacting the clients [92]. Obviously the problem does not stop here, since:

- The precise definition of what constitutes an interface will have an impact on both the client and on the implementer of a module. A more complete interface will be a *benefit* for the client of module but an *obligation* for the implementer of the module – and will as well reduce the opportunity for non-breaking changes.
- Although discouraged, the interface itself is subject to change.

Concretely, where are the modules? Interestingly, although textbooks tend to give a very clear theoretical description of modules, many do not make an explicit connection with general purpose languages such as C# or Java. Is a module a class, a namespace or an assembly? Depending on which school of thought you subscribe to, the answer will vary. Object-orientation purists might say that a module is a class (for example the Eiffel community [134]); practitioners tend to rely heavily on assemblies as a unit of modular decomposition; namespaces play a more subtle but yet important scoping mechanism (for example, Bergel goes as far as calling namespace *a unit of modularization* in C# [25]).

1.4.3. The crystal ball assumption

We dubbed the required anticipation of likely changes, *the crystal ball assumption* (see section 4.1.3). The purpose of this pun is to attract attention to the fact that, at the core of many approaches to software evolution, lies an un-reasonable assumption: the capacity of software designers to predict the future needs for change. Ossher and Tarr write:

> "Anticipation causes ulcers : Deeply ingrained within software engineering is the notion of anticipating and designing for the *most likely* kinds of changes, towards the goal of limiting the impact of future evolution. [...] This is true, and we believe in anticipating and planning for changes whenever possible. Anticipation is not, however, a panacea for evolution. It clearly is not possible to anticipate all major evolutionary directions. Further, even if it were possible, building in evolutionary flexibility always comes at a price: it increases development cost, increases software complexity, reduces performance, or often all of the above." [156].

The work of Ossher and Tarr on Hyper/J [156, 157] (see section 2.5.9) was influential on our work; like them, we think that anticipation is *useful*, but is *not a panacea*. As observed in chapter 4, enterprise system is a field where anticipation is difficult: ERP systems are deployed worldwide in very different contexts. Who knows how legal regulations will evolve in the next 2 years in more than 50 countries? Who can anticipate the future need for evolution in vertical domains as diverse as transport, textile and farming? ERP systems can be contrasted with software products that deal with more *stable domains*. Consider for example graph libraries: graphs have been thoroughly studied and the concepts and design alternatives around graphs are well comprehended, have been extensively explored, and are well documented. See for example Schmidt and Ströhlein for graph theory [178], or the Boost [30] and the Quickgraph [168] projects for graph library implementations. Of course, new important variations around graphs do surface once in a while – but it is relatively rare. Typically, that new variation would simply be incorporated in the next version of the library: what is required in this case is not code customization but *code evolution*.

1.4.4. Towards code query by example

Through our empirical study of ERP systems (chapter 3), we learned, among other things, that:

- Customizations *cannot be anticipated.*
- With exceptions, partners are more *domain experts* than IT specialists.
- Any new customization solution would have to build on a *very large existing code base*.

Each of those points put further constraints on a solution to the upgrade problem and calls for a departure from standard methods: modular decomposition *requires* anticipation which is not possible in our case; due to its large size an existing ERP system cannot be refactored just for one partner (and refactoring will have to be re-applied manually upon upgrade). Trying to solve the upgrade problem, while at the same time taking into consideration those constraints, led us to an approach based on the concept of *code query by example*. Section 1.5 presents this approach in detail.

1.5. Code Query by Example

This section presents *code query by example* (CQE), the proposed approach to customization of software products. We want to emphasize that we see CQE as *complementary* to existing customization techniques: for example, when very fine-grained customizations are needed, in-place customizations are probably the only viable approach. CQE can *quantify* over the existing code source of large software products without the need for special markers in the target code base: the queried software products are *oblivious* to their queries [75, page 24]. Hence, CQE requires little anticipation. (Section 1.3.3 gives a definition of *quantification* and *obliviousness*). Quantification is done by matching code patterns against the base software product, and is used to denote *customization points*. One of the main aspect of the query language is its *ease of use*, due to its use of *code example*s, making it convenient for domain-experts (non computing science specialists). Finally, it can be adopted incrementally (since queries are made against an *existing* software product, without the need for special markers in the base code of the queried software).

1.5.1. An embedded domain-specific language

The approach is based on an *embedded domain-specific language* [99, 77]. It is *domain specific* since it deals with a well defined task: quantification over an existing software product. It is *embedded*, in the sense that all valid definitions of customization points, and all valid customizations in our language, are also valid programs in the host language. Of course, the *interpretation* of a particular program differs depending if one looks at it in the context of our embedded language or in the context of the host language [99]. A benefit of using an embedded language, is that we have *full support* for all the functionalities offered by existing development environments, such as design time and interactive typing, refactoring, etc.

1.5.2. Specification of CQE

Programmers express their queries using a high-level programming language for the .Net platform. Queries are written using *code examples*. Those code examples are contained in methods. When a query is applied on a software product, it returns a set of *code fragments*. Each of these code fragments is an occurence of a *sequence of statements* in the queried software product (*the target code*). Code fragments returned by code queries, denote *customization points*. A customization point is a program location where behavior can be added. A tool instruments the software product according to those customization points, and generates interfaces. Those interfaces can be implemented by partners to construct customizations. A customization resides in its own assembly that is loaded, at run-time, by the instrumented software product.

In summary:

1. Partners write code queries to denote *customization points*. The queries are expressed using *code examples.*

2. Using a tool, queries are *matched* against an existing software product.
3. Using the same tool, customization points are *inserted* at the matched locations. The tool also generates *customization interfaces*.
4. Partners implement some of the generated interfaces according to their customization needs.
5. At *run-time*, customizations are loaded, and invoked when the program reaches a customization point.

Note that partners mentioned at step 1 and at step 4 are not necessarily the same individuals.

1.5.2.1. Customization points

A *customization point* is a location *inside a method body* where a sequence of customizations *can* be triggered. Customization points are not necessarily located at the start or the end of a method body, but they are never located inside an existing statement. That is, customization points can be located in-between existing statements; before the first statement of a method body; or after the last statement. The concept of *customization point* is similar to the concept of *join point* in aspect-oriented programming [74], see chapter 5.

1.5.2.2. Customization interfaces

A *customization interface* is a machine generated interface, that contains a single abstract method, namely: `Customization`.

1.5.2.3. Customization classes

A *customization class* is a class that implements a customization interface. Customization classes are written by partners.

1.5.2.4. Customization methods

A *customization method* is a concrete implementation of a `Customization` abstract method defined in a customization interface. Customization methods are implemented by partners. (Section 1.5.2.17 describes how to implement a customization method using one of the generated interfaces.)

1.5.2.5. Customization objects

A *customization object* is an instance of a customization class. Customization objects are automatically instantiated by the framework; customization objects are stored in a container.

LISTING 1.2: QUERY METHOD

```
public class Example {
  [Query("Q")]
  public void Q1(T1 p1, T2 p2, ...) {
    // code pattern
  }
  [Query("Q")]
  public void Q2(T1 p1, T2 p2, ...) {
    // code pattern
  }
}
```

1.5.2.6. Customization calls

A *customization call* is a method call to a customization method at a customization point. (Note that customization calls are *not* inserted in the base software product; instead, a call is made using a customization interface, and at runtime the framework automatically maps this call to a sequence of customization calls).

1.5.2.7. Query methods

A *query method* is a method annotated with a query attribute. We make use of the concept of an *Attribute* in the .Net framework [66, 65] to add metadata to a procedure: our framework defines a special attribute called *Query*.

1.5.2.8. Queries

Queries are *code patterns.* They are used to quantify over existing software products, in order to locate customization points *statically*. All query attributes take as an actual parameter of their constructor a string. This string will be used to *name* the query. A query has the form shown in listing 1.2. Q1 and Q2 are two *query methods* that define the *query* Q.

1.5.2.9. Convention

As noted previously, all queries and all customizations must be valid programs in the host language. For brevity, we sometimes only show the relevant methods, and assume that they are defined in an enclosing class. The name of the enclosing class in non significant, and mainly matters for documentation purposes.

1.5.2.10. Matching of entities

Let `P` be an entity in a query, and let `T` be an entity in the target code. `P` will match `T` only if `P` and `T` share the same static type. The name of `P` and `T` is non-significant for

matching. For example, entity `int x;` in a query will match entity `int y;` in some target code. Matching requires that constant values are the same in the query and in the target code; for example, statement `F("x");` will only match statement `F("x");` where `F` is the same method.

1.5.2.11. Disjunction of query methods

Multiple query methods can share the same query name. The semantics of the query Q, in listing 1.2, is defined as the disjunction of two cases:

- code pattern of the form Q1,
- or, code pattern of the form Q2.

A query can have as many disjuncts as required.

1.5.2.12. Constraints to the definition of query methods

There are two main constraints to the definition of query methods:

1. Query methods must not return any value – they must be *procedures* and not *functions.*
2. Query methods that participate in the definition of the same query must have the same *signature*.

For instance, the two query methods defined in listing 1.2 are both procedures, and have the same signature.

1.5.2.13. Query variables

Formal parameters of query methods define *query variables*. Given a query variable p of static type T, we distinguish two cases :

1. First case, if T is a *delegate type*, instances of a method call to p in the method body of a query method will denote:
   a) a sequence of statements in the target code, if T is a procedure – ie., if T is of type `Action`
   b) an expression of type R in the target code, if T is of type `Func<R>`.
2. Second case, if T is not a delegate type, an occurrence of p in the method body of a query method denotes a variable in the target code.

1.5.2.14. Non-linear patterns

Several instances of the same query variable *p* in the method body of a query method, denotes the same sequence of statements, the same expression, or the same variable respectively if *p* is of type `Action`, `Func<R>`, or an instance of a non-delegate type. This is commonly referred to in the literature as a *non-linear pattern* [204]. For example, the following query Q uses two instances of the query variable `x`.

```
[Query("Q")]
void Q1(int x) {
 x = 0;
 x = 1;
}
```

1.5.2.15. Controlling customization points locations

A sequence of statements can only be matched if its enclosing method is publicly accessible. This feature allows designers of software products to control which method can be customized using the standard *public* access modifier. Constructors are not matched.

1.5.2.16. Generation of customization interfaces

If the query `Q` described in listing 1.2 matches at least one code fragment within a software product, 4 customization interfaces will be generated automatically by the framework, and added to the software product:

- **interface** `Q.Before`
- **interface** `Q.After`
- **interface** `Q.BeforeByRef`
- **interface** `Q.AfterByRef`

Each interface contains a single abstract method, respectively:

- **void** `Customization(T1 p1, T2 p2, ...);`
- **void** `Customization(T1 p1, T2 p2, ...);`
- **void** `Customization(ref T1 p1, ref T2 p2, ...);`
- **void** `Customization(ref T1 p1, ref T2 p2, ...);`

The first two interfaces and the last two interfaces share the same single abstract method because interfaces play a dual role: first, through the signature of the abstract method `Customization`, they enforce how advices should be implemented;

Table 1.2.: GENERATED CUSTOMIZATION INTERFACES

Customization point is	*Arguments passed by value*	*Arguments passed by reference*
before a matched code fragment	`Before`	`BeforeByRef`
after a matched code fragment	`After`	`AfterByRef`

second, they indicate whether the customization should be executed before or after the matched code fragments. Except for the `ref` modifiers, the signature of the `Customization` abstract method is the same signature of its corresponding query method.

1.5.2.17. Implementation of customizations

In order to construct a customization, partners implement one of the generated interfaces. Given a matching query Q, if a partner wants to implement a customization at a customization point *before* a matched code fragment, he should implement either `Q.Before` or `Q.BeforeByRef`. Similarly, if a partner wants to implement a customization at a customization point *after* a matched code fragment, he should implement either `Q.After` or `Q.AfterByRef`. Obviously, the namespace enclosing a generated customization interface is the name of the query. Finally, and intuitively, arguments of customization calls are passed by reference with the interfaces `BeforeByRef` and `AfterByRef`, and by value otherwise. Table 1.2 summarizes the intent behind the generated customization interfaces.

1.5.2.18. Instantiation of customization objects

As explained in section 1.5.2.16, each customization point has two corresponding customization interfaces, for example `Q.Before` and `Q.BeforeByRef` . During execution, when a software product reaches this customization point, the framework searches for customization classes that implement `Q.Before` or `Q.BeforeByRef`. For each of those classes, the framework looks if there is a corresponding instance, a singleton, in the customization objects container. If no such object exists, the framework automatically instantiates this class, and places a reference to the singleton object in the container.

1.5.2.19. Partial ordering of customization calls

When a software product reaches a customization point, the customization objects corresponding to that particular customization point are retrieved from the container, and the customization methods are invoked. It is sometimes useful to specify an order in which those customization methods will be invoked. The framework lets partners specify a *partial order* between customization classes. Since customization objects

are singleton, a partial order on customization classes also defines a partial order on customization objects. The framework provides two attributes `After` and `Before` , that can be used to annotate customization classes in this respect. For example, in the listing below, the partner specifies that the `Customization` method of the class `CustomizationClass2` should be invoked after the `Customization` method of class `CustomizationClass1`. At runtime, the framework will look for a *linear ordering* (a *total ordering*) that satisfies the partial ordering defined by the partners. Customization methods will be invoked following this linear ordering. If no linear ordering satisfying the partial ordering can be found, the framework will throw an exception.

```
[After(typeof(CustomizationClass1)]
public class CustomizationClass2 : Q.Before {
 void Customization(...) {
   [...]
 }
}
```

1.5.3. Matching examples

Convention Given a query Q, and a target program T, $Q(T)$ denotes the set of code fragments matched by Q in T. A code fragment is represented as an ordered pair of natural numbers (a, b), where $a \leq b$. This convention is only used to ease the discussion around matching examples; in C#, statements can span multiple lines. (The implementation, discussed in section 1.7, is not dependent on line numbers). Each of those pairs denotes a code fragment as follows:

- The first entry (the left projection) gives the start line number of the matched code fragment, in the context of the listing of the target software product.
- The second entry (the right projection) gives the end line number of the matched code fragment, in the context of the listing of the target software product.

Hence, $Q(T) = \{(1,1),(4,8)\}$ denotes a result set of a code query Q applied to a target program T, with two matched code fragments: the first matched code fragment consists of only the first line of the listing, whereas the second matched code fragment consists of the sequence of statements starting at line 4 (inclusive), and ending at line 8 (inclusive).

1.5.3.1. Empty query

An *empty query* is a query method with no statement in its method body. An empty query matches every statement inside method bodies of a target software product. The following query Q is an empty query:

```
[Query("Q")]
static void Q() { }
```

Let T be the target software product below, $Q(T) = \{(4,4),(5,5)\}$

```
1 int x;
2 int y;
3 public void M() {
4   x = 0;
5   y = 0;
6 }
```

1.5.3.2. Simple query method

A simple case of a query method is a query method without formal parameters. The following query will match all statements that consist of method call to the method `F`.

```
[Query("Q")]
static void Q() {
  F();
}
```

Let T be the target software product below, $Q(T) = \{(2,2),(4,4)\}$

```
1 public void M(ref int y) {
2   F();
3   y++;
4   F();
5 }
```

1.5.3.3. Another simple query method

A small variation on the example above, slightly more complicated, is two consecutive method calls to two different methods.

```
[Query("Q")]
static void Q() {
  A.F();
  A.G();
}
```

Let T be the target software product below, $Q(T) = \{(2,3)\}$. Note that $(4,6)$ is not part of the result set since the last occurrences of method calls to `A.F()` and `A.G()` are not consecutive.

```
1 public static void M() {
2  A.F();
3  A.G();
4  A.F();
5  Console.WriteLine(...);
6  A.G();
7 }
```

1.5.3.4. Query variables

As mentioned earlier, a formal parameter of a query method denotes a *query variable*. Given the following query method, `s` is a query variable of static type `string`.

```
[Query("Q")]
static void Q(string s) {
 Console.WriteLine(s);
}
```

Let T be the target software product below, $Q(T) = \{(2,2),(3,3)\}$. Note that the last statement of the method `M` does not match since the argument of the method call to `Console.WriteLine(char)` does not match the static type of the query variable `s`.

```
1 void M() {
2  Console.WriteLine("x");
3  Console.WriteLine("y");
4  Console.WriteLine('x');
5 }
```

1.5.3.5. Nonlinear patterns

There can be multiple occurrences of a query variable in a query method, giving rise to a *nonlinear pattern* [204]. For example:

```
[Query("Q")]
static void Q(string s) {
 Console.WriteLine(s);
 Console.WriteLine(s);
}
```

Let T be the target software product below, $Q(T) = \{(3,4)\}$.

```
1 void M() {
2  Console.WriteLine("x");
3  Console.WriteLine("y");
4  Console.WriteLine("y");
5 }
```

LISTING 1.3: ACTION QUERY VARIABLES

```
[Query("P1")]
void Q1(Action a) {
  a();
}
[Query("P2")]
void Q2(Action a, int x) {
  x++;
  a();
}
[Query("P3")]
void Q3(Action a, int x) {
  x++;
  a();
  x++;
}
```

1.5.3.6. Action query variables

A *query variables* of type `Action` matches a sequence of statements in the target code base. This quantifier is *greedy*, it will try to match the longest possible sequence of statements that satisfies the query. Consider the queries in listing 1.3. The query `P1` will match the longest possible sequence of statements within the target code base: full method bodies. The query `P2` will match the longest sequence of statements where an integer variable is incremented followed by a sequence of statements. Query `Q3` will match all code fragments that start by a variable being incremented and end with the same variable being incremented once again.

Let T be the target software product in listing 1.4:

- $P1(T) = \{(5,6),(10,12)\}$
- $P2(T) = \{(5,6),(11,12)\}$
- $P3(T) = \{(5,6),(10,12)\}$

1.5.3.7. Func query variables

Query variables of type `Func<T>` denotes an expression of type `T` in the target code. The query `Q` below matches all statements that assign an expression of type **`int`** to a variable of type **`int`**.

```
[Query("Q")]
void Q1(Func<int> f, int x) {
  x = f();
}
```

Let T be the target software product in listing 1.5, $Q(T) = \{(4,4),(5,5)\}$

LISTING 1.4: TARGET SOFTWARE PRODUCT

```
1  int x;
2  int y;
3
4  public static void M1() {
5    x++;
6    x++;
7  }
8
9  public static void M2() {
10   y++;
11   x++;
12   y++;
13 }
```

1.5.3.8. Try/catch blocks

A programmer can specify try/catch/finally blocks in its queries, matching only the target code that has the corresponding try/catch/finally clause.

```
[Query("Q")]
int Q1(Action a, Action b) {
 try {
  a();
 } catch (IOException ex) {
  b();
 }
}
```

Let T be the target software product in listing 1.6, $Q(T) = \{(2,6)\}$

1.5.4. CQE extensions to the original concept

We consider two extensions to improve the expressiveness of CQE: *anchoring*, and *query composition* using query expansion.

LISTING 1.5: TARGET PROGRAM, FUNC QUERY VARIABLE

```
1 int x;
2 int y;
3 void M() {
4  x = 1;
5  y = x + y;
6 }
```

LISTING 1.6: TARGET PROGRAM (TRY/CATCH)

```
1 public void M(string file) {
2  try {
3   Console.WriteLine(File.Read(file));
4  } catch(IOException ex) {
5   Log(ex);
6  }
7 }
```

1.5.4.1. Query method expansion

Query expansion allows partners to compose queries from existing query method. (Note that queries are not composed from existing *queries*, but from existing *query methods*.). It is reminiscent of macro expansion.

After query method expansion, query `Q` in listing 1.7 will be equivalent to query `Q` in the following query:

```
[Query("Q")]
void M(string s) {
  F(s);
  G(s);
}
```

1.5.4.2. Anchoring

Anchoring allows partners to specify if the matched code fragments should be located at the beginning or at the end of a method body. The framework defines a enum type `Anchoring` (an enumeration [65]), with a small set of named constants (an enumeration list):

- Anchoring.Any
- Anchoring.Start
- Anchoring.End

LISTING 1.7: QUERY METHOD EXPANSION

```
[Query("Q")]
void M(string s) { N(s); O(s); }
[Query("Q1")]
void N(string s) { F(s); }
[Query("Q2"]
void O(string s) { G(s); }
```

LISTING 1.8: ANCHORING

```
[Query("Q1", Anchoring = Anchoring.Start)]
void P1(string s) { F(s); }

[Query("Q2", Anchoring = Anchoring.End)]
void P2(string s) { F(s); }

[Query("Q3", Anchoring = Anchoring.Start | Anchoring.End)]
void P3(string s) { F(s); }
```

Intuitively, they specify that a matched code fragment should be anchored respectively: at any position (default value), at the start, and at the end of a method body. This enum can be used as part of the `Query` attribute, as illustrated in listing 1.8. Query `Q3` specifies that the matched code fragment must be the full method body of a target method.

Let T be the target software product in listing 1.9,

- $Q1(T) = \{(2,2),(6,6)\}$
- $Q2(T) = \{(3,3),(6,6)\}$
- $Q3(T) = \{(6,6)\}$

LISTING 1.9: TARGET PROGRAM: ANCHORING

```
1 public void M1() {
2   F("x");
3   F("y");
4 }
5 public void M2(bool b) {
6   F("y");
7 }
```

1.6. Upgrade with CQE

We now come back to *the upgrade problem*. Recall the core of the upgrade problem, see figure 1.5. P_1^1 is the original software product; P_2^1, P_3^1, etc., are *customizations* of the original software product. P_1^2 is an *evolution* of P_1^1. Customizations have to be ported to P_1^2 (a process symbolized by dotted-lines in figure 1.5). Existing queries, that were written for P_1^1, will be matched against P_1^2. As explained previously, matching returns code fragments, which in turn give rise to customization points in the target code.

More generally, we consider a query Q as a function that maps software products to a set of code fragments. Recall that a *code fragment* is an occurrence of a sequence of statements in the code base, and that each code fragment give rise to two customization points (see section 1.5).

- Before upgrade, $Q(P^n) = \{C_1^n, C_2^n, C_3^n, ...\}$, where each C_i^n is a code fragment.
- Upon upgrade, Q is reapplied to P^{n+1}, the new version of the software product: $Q(P^{n+1}) = \{C_1^{n+1}, C_2^{n+1}, C_3^{n+1}, ...\}$.

We distinguish four cases:

- Let C be a code fragment such that $C \in Q(P^{n+1})$, if at least one of the customization points denoted by C is required to implement an existing customization, then we call C a *true positive*.
- Let C be a code fragment such that $C \in Q(P^{n+1})$, if none of the customization points denoted by C are required to implement an existing customization, then we call C a *false positive*.
- Let C be a code fragment such that $C \notin Q(P^{n+1})$, if none of the customization points denoted by C are required to implement an existing customization, then we call C a *true negative*.
- Let C be a code fragment such that $C \notin Q(P^{n+1})$, if at least one of the customization points denoted by C is required to implement an existing customization, then we call C a *false negative*.

Table 1.3 summarizes this terminology. In this section, we will look at different cases that cause true positives, false positives, true negatives, and false negatives.

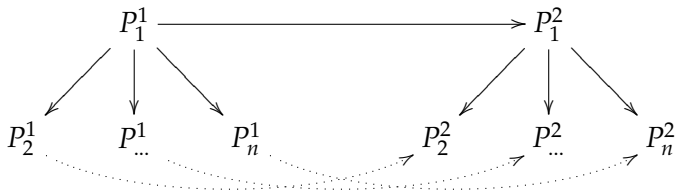


FIGURE 1.5.: UPGRADE

Table 1.3.: UPGRADE

	One of the customization points denoted by C is required	*None of the customization points denoted by C are required*
$C \in Q(P^{n+1})$	true positive	false positive
$C \notin Q(P^{n+1})$	false negative	true negative

1.6.1. True positives

A true positive is a code fragment $C \in Q(P^{n+1})$, such that at least one of the customization points denoted by C is required to implement an existing customization written for P^n.

1.6.1.1. Examples of true positive

Consider the query `Transaction` in listing 1.11, initially written by partner R to customize the software product P^n, mentioned in listing 1.10. The concept of a transaction, according to R, is a `Debit` operation on an account, followed by a sequence of statements, followed by a `Credit` operation on an account. (The `Debit` and the `Credit` operations must be applied to the same amount *variable*).

The *intent* of partner R is to log all instances of what he calls a transaction, at run-time, see listing 1.12.

Consider the following four possible changes from P^n, each giving a different P^{n+1}:

- Method `M1` is renamed `M4`.
- The method body of `M1` is moved to `M3`.
- The method body of `M1` is duplicated to `M3`.
- Method `Debit` is renamed `Withdraw`.

In this example, each of those changes gives rise to a new software product; changes are considered *individually*. For each of the four changes, the query `Transaction`, applied to P^{n+1}, will respectively return the following results:

- Two true positives: A code fragment in `M2`, and a code fragment in `M4`.
- Two true positives: A code fragment in `M2`, and a code fragment in `M3`.
- Three true positives: A code fragment in `M1`, a code fragment in `M2`, and a code fragment in `M3`.
- None (compile-time error).

The last case creates a compile-time error when the query `Transaction` is compiled, and requires a corrective action.

LISTING 1.10: TARGET CODE BEFORE UPGRADE

```
public void M1(Account a, Account b, double amount) {
 a.Debit(amount);
 b.Credit(amount);
}

public void M2(Account a, Account b, double amount) {
  a.Debit(amount);
  Wait(1000);
  b.Credit(amount);
}

public void M3() {
}
```

LISTING 1.11: QUERY TRANSACTION

```
[Query("Transaction")]
void Q1(Account a, Account b, double amount, Action action) {
 a.Debit(amount);
 action();
 b.Credit(amount);
}
```

LISTING 1.12: A CUSTOMIZATION: LOGTRANSACTION

```
public class LogTransaction: Transaction.After {
 public void Customization(Account a,
                           Account b,
                           double amount,
                           Action action){
   EventLog.Log(..., a.Id, b.Id, amount);
 }
}
```

1.6.1.2. Corrective actions

A corrective action is a measure taken by partner to change an existing set of queries to make them fit for a new version of a software product.

- The last case requires a corrective action: the query `Transaction` does not compile any longer since the method `Debit()` does not exist in P^{n+1}. The partner should then simply change the statement `Debit();` in query method `Q1` into `Withdraw();` Note that the corrective action is done in only one location.

- Consider the other cases. Why should there be a corrective action for a true positive? This seems counter-intuitive, since a true positive is *required* to implement a customization written for the previous version. To the best of our knowledge, there is only one reason: when a feature that was implemented for P^n is now provided by default in P^{n+1}. (The matched code fragment is classified as a *true positive* since we make the distinction between *a customization*, and a *customization point* necessary to implement a customization, see section 1.5.) For example, assume that in P^{n+1}, the base software maker introduces a method `Transaction`, that makes use of the Template method design pattern (see section 1.9):

```
public void Transaction(Account a, Account b, double amount) {
 a.Debit(amount);
 b.Credit(amount);
 EventLog.Log(..., a.Id, b.Id, amount);
 PartnerAction();  // call to an abstract method
}
public abstract void PartnerAction();
```

  We assume that the rest of code base was fixed by the base software maker to use this `Transaction` method. The `Transaction` method logs the transactions. Since the feature is now implemented directly by the base software product, the customization is no longer necessary. Even if this method did not log the transactions, the partners would have the opportunity to implement this feature by redefining the abstract method `PartnerAction` (in this case, the base software maker could *anticipate* the partner's needs). When a feature that was implemented through customization in P^n is implemented by default in P^{n+1}, the partner has two options:

  - Either keep using the existing customization that implements the feature. In this case, no corrective action should take place. (In the example, the query `transaction` will only match one code fragment, inside the method `Transaction`).
  - Or, decide to use the implementation of the feature provided by default (in partner's parlance *"go standard"*, see chapter 3). In this case, the partner should remove the corresponding customization classes, and if no other customization depends on one of the generated interface, the query

LISTING 1.13: P^{n+1}, FALSE POSITIVE: (7,8)

```
1  public void M1(Account a1,
2                 Account b1,
3                 Account a2,
4                 Account b2,
5                 double amount) {
6   b1.Credit(amount);
7   a1.Debit(amount);
8   b2.Credit(amount);
9   a2.Debit(amount);
10 }
```

`Transaction` itself can be removed. Note that the partner should also correct the client code that makes use of the customization class.

1.6.2. False positives

A false positive is a code fragment $C \in Q(P^{n+1})$, such that none of the customization points denoted by C is required to implement an existing customization written for P^n.

1.6.2.1. Example of false positive

Consider the query `Transaction` in listing 1.11, initially written by partner R to customize the software product P^n, mentioned in listing 1.10. Consider the consecutive software product P^{n+1}, mentioned in listing 1.13.

The method `M1` was changed and now has 4 accounts as formal parameters, and contains two consecutive transfers – but the `Credit` and `Debit` operations take place in reverse order. Applying the query `Transaction` on P^{n+1} would match the code fragment:

```
a1.Debit(amount);
b2.Credit(amount);
```

This is not the intent of R; we have a false positive, a matched code fragment that is not required. The false positive by itself does not create a defect; it is the combination of the the false positive and of the customization `LogTransaction` that causes the event log to show a wrong entry.

1.6.2.2. Corrective actions

False positive can happen in many cases. Fortunately, there is an asymmetry that we can leverage: the list of matched code fragment is likely to be relatively small compared to the large size of the software product. Hence, a programmer can

inspect the query result set, and validate if a customization should take place at those locations.

In the example, the query `Transaction` has to be fixed. Note that we not only have a false positive in `M1`, we also have two false negatives. One can try to fix the query by introducing a disjunct to the definition of the query `Transaction`:

```
[Query("Transaction")]
void Q2(Account a, Account b, double amount, Action action) {
 b.Credit(amount);
 action();
 a.Debit(amount);
}
```

In which case both disjuncts are matched: when `Debit` is applied first, and when `Credit` is applied first. While this fixes the false negatives, it has two major drawbacks:

- When the customization method will be called, one cannot differentiate any longer which is the debited account, and which is the credited account. Sometime this distinction is not important (when one just need to know that a transaction has happened between two accounts), sometimes the distinction matters. In the latter, instead of adding a disjunct, one can introduce another query, called for example `TransactionCreditFirst`. In which case, it is straightforward for partners to distinguish the accounts involved in the transaction (by implementing either `Transaction.After` or `TransactionCreditFirst.After`). If this solution is chosen, and if a partner is not concerned with the distinction between accounts, then the same customization class can implement both interfaces.

- It does not fix the false positive: $(7,8)$ will still be matched[5]. One could try to add an exception mechanism to CQE to remove certain code fragments from a result set. The challenge then is how to denote $(7,8)$, without using explicit marker on the source code, and similarly without using line an column numbers (that hinder upgrade). Removing the first disjunct, where `Debit` is applied first, is not a solution neither, since this would give rise to false negatives.

While false positives can be easily *identified* by validating a reasonably small list of positives, *fixing* this particular false positive prove to be challenging. The core of the problem is that it seems that there is no *oblivious* ways to denote the false positive. When queries cannot be fixed, partners can simply remove the query and use the traditional customization methods.

Other false positives can be fixed by making the query *less abstract*: for example, by removing the action query variable, and by giving *the extension*, using disjuncts, of all possible cases that the partner wishes to capture: the case when there is no statement between the `Debit` and `Credit` operations, the case where method `F` is invoked between the two operations, the case where method `G` is invoked, etc., see

[5]The convention for this tuple notation is explained in section 1.5.3

LISTING 1.14: A MORE CONCRETE QUERY

```
[Query("Transaction")]
void Q1(Account a, Account b, double amount) {
 a.Debit(amount);
 b.Credit(amount);
}
[Query("Transaction")]
void Q2(Account a, Account b, double amount) {
 a.Debit(amount);
 F();
 b.Credit(amount);
}
[Query("Transaction")]
void Q3(Account a, Account b, double amount) {
 a.Debit(amount);
 G();
 b.Credit(amount);
}
```

listing 1.14. The problem with very *concrete queries*, is that they are fragile to evolution; very concrete queries are likely to give rise to false negatives – which are difficult to detect, as we will see in section 1.6.4.

1.6.3. True negatives

A true negative is a code fragment $C \notin Q(P^{n+1})$, such that none of the customization points denoted by C is required to implement an existing customization written for P^n.

1.6.3.1. Example of true negative

A true negative is the simplest and the most common case. The example in section 1.6.1, shows, among others, the following true negative: when the method body of method `M1` is moved to `M3` in P^{n+1}, no customization point is found in `M1` upon upgrade.

1.6.3.2. Corrective actions

To the best of our knowledge, no corrective actions are required for true negatives.

LISTING 1.15: P^{n+1}, FALSE NEGATIVES: (2,3) AND (11,12)

```
1  void M1(Account a, Account b, double amount) {
2   Withdraw(a,amount);
3   b.Credit(amount);
4  }
5
6  void Withdraw(Account a, double amount) {
7   a.Debit(amount);
8  }
9
10 void M2(Account a, Account b, double amount, MethodInfo debit) {
11   debit.Invoke(a,amount);
12   b.Credit(amount);
13 }
```

1.6.4. False negatives

A false negative is a code fragment $C \notin Q(P^{n+1})$, such that at least one of the customization points denoted by C is required to implement an existing customization written for P^n.

1.6.4.1. Examples of false negative

Section 1.6.2 showed two examples of false negatives. In P^{n+1}, `M1` was changed to contain two transfers. But the `Credit` operation is applied first, hence query Transaction did not match the two transfers. By attempting to fix the false positive, we also fixed the two false negatives.

We are going to look at two other examples of false negatives. Consider once again the listing 1.10 (P^n), and the listing 1.12 (the customization), and listing 1.11 (the query). P^{n+1} is shown in listing 1.15.

Concerning `M1`, the `Debit` operation is no longer in the same lexical scope with the `Credit` operation and hence the query `Transaction` will not match the body of `M1`. This is a false negative, in the sense that the *intent* of partner R is not satisfied upon upgrade. If one stick to the strict definition of the query `Transaction`, one could argue that this is not a false negative – in the sense that the definition of a transaction as given by the query `Transaction` *implied* that the two operations *must* be in the same scope. We will assume the former, this is a false negative.

Concerning `M2`, the `Debit` method is (potentially) called by reflection. The query `Transaction` will not match the method body of `M2`. This is a false negative. Again, one could argue that this is not a false negative – since the strict definition of a transaction as given by the query `Transaction` exclude a reflexive call.

1.6.4.2. Corrective actions

False negatives are difficult to deal with, since contrary to false positives, *we cannot leverage the relatively the small size of query result sets*; the tools will not detect false negatives, the partners must have a complete understanding of the base software product, which is obviously not feasible since the software products that we consider are very large.

Concerning `M1`, how to fix the false negative? In this particular example, a partner could try to add a disjunct case to the query `Transaction`:

```
[Query("Transaction")]
void Q2(Account a, Account b, double amount, Action action) {
 Withdraw(a,amount);
 action();
 b.Credit(amount);
}
```

Although this solution fixes this particular problem, it is not satisfactory in the general case, since the number of disjunct would quickly explode, and as a side-effect the number of false positives would also increase.

Concerning `M2`, the problem caused by the use of reflection, one could attempt the following query:

```
[Query("TransactionDebitByReflection")]
void Q2(Account a, Account b, double amount, MethodInfo debit) {
  debit.Invoke(a,amount);
  b.Credit(amount);
}
```

This will capture the transaction in `M2`, but it will also capture reflective calls to methods which are not `Debit` – giving rise to false positives; this demonstrates that the use of reflection can hinder upgradability.

1.6.5. Summary

As we illustrated, all queries potentially need to be modified upon upgrade. So how does code query by example help with respect to upgrades? We summarize the main points that were raised:

- Using queries, customization points appear, disappear, and move from one location to another, depending on the evolution of the software product.
- There is an asymmetry between size of the software product and the size of the query result sets. Partners can browse through the (relatively) small list of positives, and check if each positive is a true positive of a false positive.
- Some customizations can be adjusted by modifying the existing queries – textually located modifications.

- Queries are type checked. For instance, if a query uses a method `F`, and this method is removed in P^{n+1}, any standard compiler will reject the query.

We illustrated that there is an interesting trade-off between:

- *Abstract queries* (queries that make use of query variables), that increase the number of false positives, but decrease the number of false negatives.
- *Concrete queries* (queries that make little use of query variables), that increase the number of false negatives, but decrease the number of false positives.

Not everything is for the best; *false negatives* are particularly problematic, and even true positives can require corrective actions. More generally, partners still need to know the subtle effect of their customization in the next version of the software product. Correctness of ported customizations is not ensured; nor was the correctness of customization before upgrade; nor was the correctness of the software product before customization. There is no specific mechanism to exclude a particular false positive from a result set. Furthermore, problems due to the use of reflection cannot be detected.

1.7. Implementation of CQE

This section describes a prototype implementation of CQE [67], and discusses some of the various design decisions that were explored.

1.7.1. Design overview

There are 3 main top level components in our implementation: *a query engine*, a so-called *customizer*, and a *Visual Studio add-in*, see figure 1.6.

- The main role of the QueryEngine component is to perform *code matching* and to return a collection of code fragments.
- The main role of the Customizer component is to *instrument* target assemblies, using results from the QueryEngine.
- Finally, the main role of the Addin component is to provide a *visualization* of the results from the QueryEngine, and to trigger customization of the target assembly.

1.7.2. Querying

Three alternative approaches Among the various possible design choices for the query engine, we distinguish concretely three options:

1. Matching at the C# abstract syntax tree level.
2. Matching at the expression tree level.
3. Matching at the bytecode level.

The first option, matching at the abstract syntax tree (AST) level, is, arguably, the more obvious design choice: query and target software texts are parsed into a tree

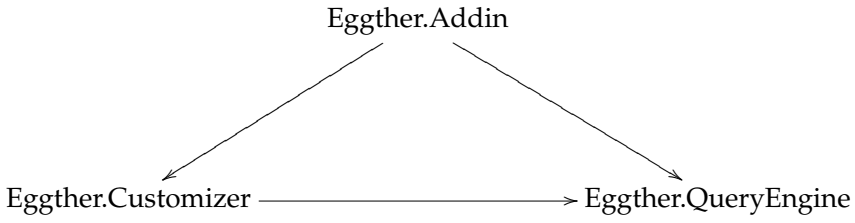


FIGURE 1.6.: MAIN INTERNAL DEPENDENCIES

representation, and code matching comes back to tree matching [204]. All the important information is readily available, and the tree data structure makes matching convenient to implement. The second option, matching at the expression tree level, is interesting in the sense that it relies on a more recent technology [135] that was less explored than the first one. Finally, the third option is the most challenging of the three: not all the information is available at the bytecode level, and the available information is sometime difficult to collect [87]. On the other hand, this an exciting approach, since it is the least explored, and has, as we will see, some interesting properties.

Dead-ends We quickly comment on two implementation ideas that lead to dead-ends:

- **CodeDom:** `System.CodeDom` is a namespace within the standard .Net base class library [135]; CodeDom contains types that can be used to represent the elements and structure of a source code document. This namespace is mostly dedicated to creating and compiling code at run-time. CodeDom could be useful to implement code matching, since most of the C# constructs are abstractly represented: for example, the type `CodeVariableReferenceExpression` for a reference to a local variable. Although one can find a `ICodeParser` interface and a `CreateParser` method in the CodeDom namespace [135], Microsoft does not provide an implementation of the parser. (A call to the parser factory method just returns null.) One can hope that a full implementation of CodeDom will be released in the near future, but CodeDom is not currently an option.
- **CodeModel:** Similarly to CodeDom, using Visual Studio, one has access to an abstract representation of the elements and structure of a source code document [135]. This namespace is simply called CodeModel. Unfortunately, there is no support for code elements within method bodies, such as statements. Hence, this solution is not directly useful to us.

1.7.2.1. Abstract syntax tree matching

This approach is based on matching at the abstract syntax tree level. We only touched upon a small prototype; our contribution is to report what seems to be the pros and cons of such an approach.

Overview The query and the target code are used in their high-level software text representation, for example C# files. The source code documents are scanned, then parsed. The result is an abstract syntax tree that gives a structured representation of the program [5]. During matching, the two abstract syntax trees are recursed simultaneously at the method body level.

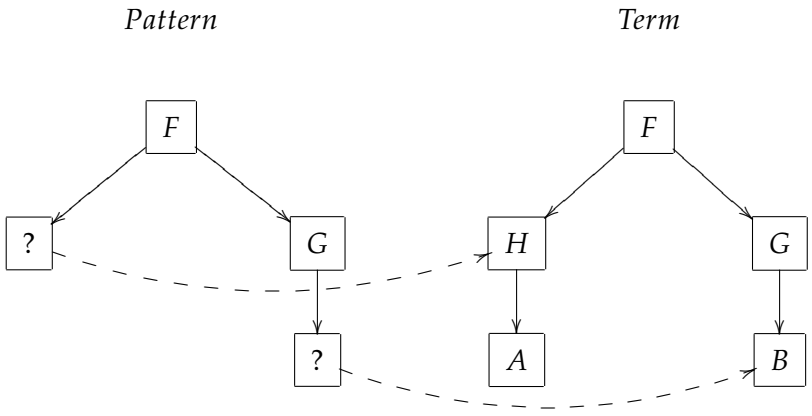


FIGURE 1.7.: MATCHING AT THE OBJECT LEVEL [204]

Main external dependency The main external dependency that we used to implement this approach is the NREFactory library [150], that contains a parser for C#.

Matching Given a query (a pattern), and a target method body (a term), matching consists in finding substitutions for the variable in the query such that the query and the pattern become equal. We used the matching strategy described by Visser [204], that allows to match objects without language extensions. Figure 1.7 summarizes matching at the object level. The pattern is a graph that contains variable objects, indicated by question marks. After matching, these variables are bound to the corresponding object in the target term (dotted arrows).

As mentioned previously, the two abstract syntax trees are recursed simultaneously at the method body level. A dictionary keeps tracks of the bindings between query variable and terms (due to non-linear patterns [204]). When the recursive descent in the tree encounters a query variable, the implementation checks if the variable is already bounded:

- If it is already bounded, it checks the current term within the target code for equality with the term bound to query variable.
- If it is not already bounded, it adds the current term in the target code to the dictionary for the given query variable.

The matching procedure yields a result if the query method and the target method match. Then, the procedure moves to the next method body in target program, and tries to perform the match once again.

Advantages of this approach Obviously, the main advantage of using a parser to perform code matching is that all the required information is available at the C# level. The parser takes care of most of the work to get the structured representation of the program. Furthermore, matching can be done on a convenient data structure that eases the implementation.

Disadvantages of this approach The main disadvantage of this approach, is that matching programs written in a complex language like C# is difficult to implement. We only experimented with a prototype that deals with a small subset of C#. (As we will explain in section 1.7.2.3, matching at the bytecode level allows us to lift some of this difficulty). Another practical problem with this approach is that high-level languages, such as C#, tend to evolve faster than lower level languages, such CIL; existing parsers for C# typically lag behind the most recent language releases. Furthermore, publicly available parsers typically do not support some advanced functionalities, like pre-processors directives. Unfortunately, to the best of our knowledge, Microsoft currently does not allow programmers to pragmatically access their internal parser. Finally, given the implementation that we described, matching has a binary result: either two method bodies match or they do not. This is not exactly what we need for CQE: we need all the code fragments within the target method body that match the query body. Of course, one could try to adapt the implementation strategy in this respect.

1.7.2.2. Expression tree matching

This approach is similar to the approach described previously and was not implemented. We only give a few pointers for future work.

Overview Expression tree is relatively recent technology, with explicit compiler support, conceived by Microsoft. Expression trees represent language-level code in the form of data [135], this data is stored in a tree structure. For example, the code below assigns a lambda expression to the `exprTree` entity:

```
Expression<Func<int, bool>> exprTree = num => num < 5;
```

The code above give rise to an expression tree [135]. This expression tree can be directly decomposed as:

```
var param = (ParameterExpression)exprTree.Parameters[0];
var operation = (BinaryExpression)exprTree.Body;
var left = (ParameterExpression)operation.Left;
var right = (ConstantExpression)operation.Right;
```

To have access to the expression tree, one would have to express what we called previously query methods, as an assignment to an entity of type `Expression<T>`. For example:

```
[Query("Q")]
Expression<Action> PrintX = () => Console.WriteLine("x");
```

Main external dependency Since expression trees are directly supported by the standard .Net framework [135], no special external dependency is required, except of course, for the standard base class libraries.

Matching Matching can be implemented similarly to what we described in section 1.7.2.1

Advantages of this approach The main advantage of this approach is that expression trees are directly supported by Microsoft.

Disadvantages of this approach A serious limitation of this approach is that only entities of type `Expression<T>` can be matched: regular methods such as `void f(){ ... }` cannot be matched, since the compiler only build an expression tree when there is an assignment to entity of type `Expression<T>`. Furthermore, in .Net 3.5, expression trees are limited: for example, lambda expressions with method bodies cannot be converted to expression trees [135].

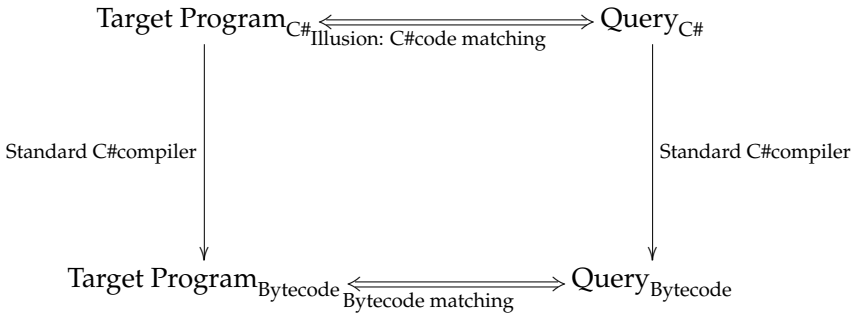
1.7.2.3. Bytecode matching

Bytecode matching is the main approach that we explored. It is challenging, for the obvious reason that some information is lost during the compilation process, among which, a part of the original structure of the code [66, 87].

Overview The first step in this approach is to compile both the query code and the target code, *using the same compiler*. We emphasize this last point. The result from this first step is two assemblies: the query assembly and the target assembly. The query engine is then used to match code fragments in the target assembly using the query assembly, see figure 1.8. With a few exceptions (see section 1.8.2), matching code statements that do not contain query variables result in the same sequence of bytecode instructions in the query assembly and in the target assembly, making matching straightforward.

Main external dependency The main external dependency is with Cecil, a library from the Mono project [143], dedicated to reading and writing assemblies.

Regex versus Regular expressions Bytecode matching leverages the Regex namespace from the .Net base class library [81, 135]. One should differentiate regular expression as studied in the field of theory of programming language from modern regular expression libraries, called *regex*. Regex libraries were extended with features that make them much more powerful that their formal counterparts. For example, our implementation makes use of *named captures* [81, 135]. In few words, *named capture* means that matched substring can be referred to with a name. It is particularly important since, for example, variable names in the pointcut and variable names in the



Single-shafted arrows denote compilation. Double-shafted arrows denote matching.

FIGURE 1.8.: MATCHING

target program can differ. For instance, the code fragment: `int x = 0; x++;` and the code fragment: `int y = 0; y++;` use different variable names, but yet should match. Another example where name capture is needed, is for branch instructions: the instruction labels in the pointcuts are obviously different from the instruction labels in the target program. An advantage of using the .Net Regexp library is that patterns can be runtime compiled, yielding better performance [81, 135].

Matching The first step is to gather in formations about query methods: the query engine looks for all methods annotated with the `Query` attribute. This does not cause any particular difficulty, but needs to be handled properly: for instance, recall that there is a *many-to-many* relationship between query names and query methods (see section 1.5) :

- A query method can have multiple query attributes.
- A query name can be used by multiple instances of query attributes.

For example, in the listing below, the query `A` is defined as the disjunction of the query methods, `Q1` and `Q2`; the query method `Q2` participates in the definition of two queries: `A` and `B`. In other words, we have a *binary relation* between query names and query methods.

```
[Query("A")]
void Q1() { ... }

[Query("A")]
[Query("B"))
void Q2() { ... }
```

The second step is to convert query methods into a regex expression. Instructions can be converted directly into a text representation, with a few exceptions. Among the main exceptions:

- Reference to variable names and parameter names must be replaced by named captures. For example, the following bytecode instructions load the field `x`, and call the method `F`, passing `x` as an argument; then again, load `x` onto the stack and call `F`.

```
ldsfld int32 A::x
call void A::F(int32)
ldsfld int32 A::x
call void A::F(int32)
```

  The operand of the first `ldsfld` instruction is turned into a new named group using the `(?<>)` construct [135], and the operand of the second `ldsfld` instruction is turned into a so-call *backreference* to the previously defined named group, using the `\k<>` construct:

```
ldsfld int32 (?<A::x>)
call void A::F(int32)
ldsfld int32 \k<A::x>
call void A::F(int32)
```

- Instruction labels will differ in the method body of the query method, and in the code fragment of the target assembly. This has an impact on matching, since branch instruction use instruction labels as their operand. This is handled similarly to what we described in the last paragraph, using named groups.
- Obviously, special characters such as parenthesis, must be escaped.
- Recall that method call to a Action delegate in the method body of a query method denotes a sequence of statements. Method calls using Action delegates are detected and turned into *star quantifiers*.

Converting method bodies of target methods to a text representation is straightforward, with one notable exception. We are looking for *code fragments*, and since a code fragment is defined as a *sequence of statements*, we need to be able to identify the exact locations where statements start and end at the bytecode level. Using C#, for example, this information is relatively easy to gather since statements are well delimited, using for example the semi-column separator [65]. This is not so with method bodies at the bytecode level: one faces a simple sequence of instructions, the information must be recovered. We looked for a systematic way to find statement at the bytecode level, and used an approach based on abstract stack interpretation.

Abstract stack interpretation The interpretation that we perform is abstract in the sense that we are only concerned with one property, *the stack height*. Interpretation is done as follows:

- For each method body, we go sequentially through the list of instructions.

- For a given instruction, we know exactly what will be the impact on the stack height [87]. For example the instruction `ldsfld int32 A::x` loads a value onto the stack, and hence grow the stack exactly by one.
- When the interpretation routine encounters a jump operation, the current stack height must be saved in a dictionary using the target label of the jump as a key. This stack height will be retrieved when the target instruction of the jump is reached.
- Obviously, the start location of the first statement and the end location of the last statement are respectively the first instruction of the method body and the last instruction of the method body. Less obviously, the locations where the stack height leaves zero delimit statements inside method bodies.

Unfortunately, one must extent the basic algorithm to take into some exceptions, statements that do not impact the stack height. For example, a method invocation to a parameter-less static procedure does not impact the stack height at call site [87].

Query method expansion CQE extensions mentioned in section 1.5.4, are currently not supported. Nonetheless, we made an attempt to implement one of the propose extension, *query expansion*, at the bytecode level. There are two main difficulties with respect to the implementation:

- First, instructions that load a parameter in sub-queries need to be replaced with the proper instruction to load the corresponding variable or parameter in the enclosing query. For example, given the composed query:

```
[Query("Q")]
void Q1() {
 string s1 = "x";
 Q2(s1);
 Q2(s1);
}

[Query("R")]
void Q2(string s2){
  Console.WriteLine(s2);
}
```

  After expansion, the query method Q1 should be equivalent (at bytecode level) to:

```
[Query("Q")]
void Q1() {
 string s1 = "x";
 Console.WriteLine(s1);
 Console.WriteLine(s1);
}
```

The method body of Q2 uses an instruction that load the parameter s2. Upon expansion this instruction should be translated into an instruction that loads the variable s1. Therefore, one has to track at call site the mapping between the arguments of the method call and the formal parameters of the corresponding method (for example, using a dictionary).

- Second, branch instructions have to be handled properly. Branch instructions use *a label* to designate the target instruction of the jump. Obviously, upon expansion, those labels will not be valid any longer. This can be handled by simply inserting all the instructions in a first pass and performing a second pass that re-assigns the operand of the jump instructions.

Note that if two variables, the first one in an enclosing query, the second one in a sub-query, share the same name, no name clash will take place upon expansion, since variables are referred to by their index, not by their variable names.

Advantages of this approach The main advantages of this approach are that:

- By working at the bytecode level, we worked directly on a normalized representation of the program: a lot of information that is not needed during matching such as code indentation and code comments is removed by the compiler.
- More importantly, type safety is enforced by the compiler. For example, given the query A:

```
[Query("A")]
void Q1(T t) {
  t.F();
}
```

  The compiler enforces that the method `F()` is defined on type `T`. This is obviously important for matching, but it will also be important during code upgrade, see section 1.6.

- Most complex language features such as pre-processor directives or complex compound statements that do not involve query statement are relatively easy to match. As explained previously, a few things need to be handled explicitly like variable names and instruction labels that differ between the query and the target code.
- The Regexp library has been well optimized, and allows for run-time query compilation [81, 135].
- It is a fact that the CIL language tends to evolve slower than high-level language such as C#. Hence the query engine requires less frequent maintenance than it counter parts that would work at the abstract syntax tree level. (And obviously, compilers for high-level language such as C#, are de facto supported by vendors).

- Finally, since matching is done at the bytecode level, and since queries and target programs are compiled using the same compiler, the query engine can potentially support multiple high-level language for .Net, such as VB or Eiffel. This is only an hypothesis, it has not been validated.

Disadvantages of this approach The disadvantages of bytecode matching are discussed in section 1.8.2, where we discuss the limitations of the prototype.

1.7.3. Visualization

Overview Since code fragments can be matched at almost arbitrary locations within method bodies, it is important to provide programmers with a way to visualize those locations. We provide a simple graphical interface in the form of an add-in for Visual Studio.

Main external dependencies The main external dependencies of this component are the visual studio assemblies required to host an add-in.

Add-in Figure 1.9 shows a snapshot of the Visual Studio add-in. Programmers open a Visual Studio solution, then select one Visual Studio project as the *target code*, and another project as the *query code*. Projects can be added and removed using the standard Visual Studio facilities. Programmers can visualize the list of queries in the query code, and can select the queries that will be used for matching. After matching, the list of code fragments is shown in the same panel. Programmers can jump to the location of a code fragment by simply clicking an entry in this list. This functionality is fully integrated with Visual Studio [135], in the sense that the code window that shows a matched code fragment is the same window that can be used to edit code. See figure 1.9.

Advantages of this approach The advantages of this approach, based on a add-in, is very similar to the advantages gained by using an ERP system: many functionalities are provided by the host program. In our case, programmers can use all the functionalities that are already provided by the host development environment to program their code queries: syntax highlighting, refactoring, and more importantly interactive type checking.

Disadvantages of this approach The main disadvantage with this approach is that users are forced to use Visual Studio as their development environment; some programmers prefer more lightweight development environments.

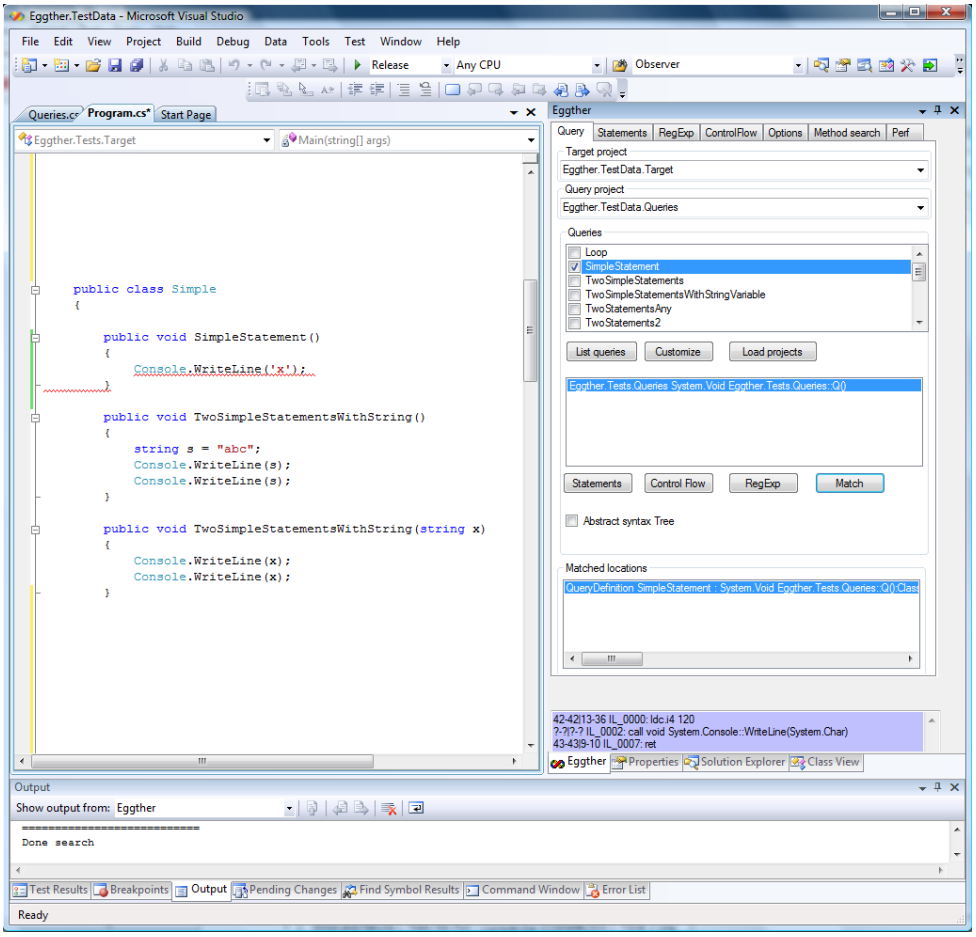


FIGURE 1.9.: EGGTHER ADD-IN

1.7.4. Instrumentation

Overview The query engine returns a set of code fragments. Each code fragment denotes two customization points: the location just before the code fragment and the location just after the code fragment. Target assemblies are instrumented at those locations. Furthermore, for each query that has at least one matched code fragment, an interface is generated at the bytecode level, and added to the target assembly.

Main external dependency The main external dependency for this component is the Cecil library [143], introduced in section 1.7.2.

Instrumentation The two goals of instrumentation are:

- To generate interfaces that will be used to implement customizations

- To enable triggering of customizations at join points

Let the code query Q be:

```
[Query("Q")]
public void Q(string name) { ... }
```

If the code query Q matches a code fragment in a target assembly, four interfaces will be generated generated at bytecode level:

- **interface** `Q.Before`
- **interface** `Q.After`
- **interface** `Q.BeforeByRef`
- **interface** `Q.AfterByRef`

Each interface contains a single abstract method, respectively:

- **void** `Customization(string name);`
- **void** `Customization(string name);`
- **void** `Customization(ref string name);`
- **void** `Customization(ref string name);`

Why four interfaces when the first two and the last two share the same single abstract method? Because interfaces play a dual role: first, through the signature of the abstract method `Customization`, they enforce how advices should be implemented (see below); second, they indicate if the customization – the advice – should be executed before or after the matched code fragments.

Now that interfaces are generated and added to the target assembly, method bodies can be instrumented at customization points:

- Before matched code fragment, all customizations that implement `Q.Before` and `Q.BeforeByRef` will be invoked; we show the C# equivalent of the bytecode instructions that are added at join points:

```
var refCustomizations = Container.Get<Q.BeforeByRef>();
var refCount = reCustomizations.Count();
for(int i=0; i < refCount; i++) {
 refCustomizations[i].Customization(ref n);
}
var customizations = Container.Get<Q.Before>();
var count = customizations.Count();
for(int i=0; i < count; i++) {
 customizations[i].Customization(n);
}
// Matched code fragment below this point
```

- After the matched code fragment, all customizations that implement `Q.After` and `Q.AfterByRef` will be created and invoked; this is very similar to what we described above.

Note that customization objects are created lazily, that is, created the first time the program reaches a join point. They remain in the container, and subsequent execution of the program at a join point merely retrieves the existing objects. Note also that multiple classes can implement the same customization interface.

Advantages of this approach Compilers enforce that programs are well-formed; programs can be further-instrumented, see section 5.5.

Disadvantages of this approach Bytecode instrumentation can be challenging, for example concerning binding of customization method parameters, concerning bytecode handling of generic types, etc. Testing bytecode instrumentation is challenging as well; tools that validate assemblies (for instance, *peverify* [135]) typically return limited information, which hinder debugging.

1.7.5. Run-time loading of customizations

Overview At this point, assemblies where instrumented at customizations points, and interfaces were generated. Partners implement one of the generated interface and simply drop the assembly that contains the customization in the same folder that contains the base program. The customization assembly is discovered *dynamically*, and customizations are triggered when the base program reaches a customization point. More assemblies can be added at *run-time* to further customize the software product.

Main external dependencies The main external dependency is with the Managed Extensibility Framework (MEF), a library recently made available by Microsoft [127].

Run-time loading of the assemblies The *base directory* is the directory that contains the base software product. Our framework uses a `FileSystemWatcher` [135], a standard class of the .Net framework to listen to the file system and receive notifications when a file is added to the base directory. Upon notification, we use MEF [127] to discover dynamically customizations. When the program reaches a customization point, all classes that implement the corresponding customization interface are instantiated, they are placed in a container and their customization method is invoked. Upon subsequent execution of the customization point, customization objects are simply retrieved from the container and their customization method is invoked.

LISTING 1.16: PARTIAL ORDERING OF CUSTOMIZATIONS

```
[After(typeof(Discount)]
public class Tax : Total.AfterByRef {
void Customization(ref Invoice i, ref double total) {
  ...
}
}
```

Linear ordering of customization calls Recall that the order in which customization methods are invoked can be specified using a partial order (see section 1.5). As illustrated in section 5.5, there are cases in which the order of customization calls matters. Programmers can use the `After` and `Before` attributes on customizations classes (classes that implement at least one of the generated customization interface). As mentioned in section 1.5, customization objects are singleton, hence an ordering on the customization classes also defines an ordering on the customization objects. For example, listing 1.16 specifies that the `Tax` customization should be performed after the `Discount` customization (see section 5.5 for the complete example).

Linear ordering is trivially implemented by the framework as topological sorting on a graph of customization classes, which is further implemented as a depth first search on the same graph. If it is not possible to find a linear order (for example, if a partner adds the attribute `[Before(typeof(Discount)]` to the class mentioned in listing 1.16), the framework will throw an exception.

Advantages of this approach The main advantage of this approach is that we allow for addition of customizations at run-time. Also it is relatively easy for partners to specify a partial order in which customizations should take place.

Disadvantages of this approach The main disadvantages of this approach are linked with loading of external assemblies in the .NET framework that present a number of technical difficulties [212, 135]. For example, the CLR cannot unload individual assemblies (*Application domains* offer a work-around [135], but bring problems of their own, among which performance penalty due to the use of the remoting namespace).

Table 1.4.: LIMITATIONS SUMMARY

Limitation	*Limitations due to*	*Subsection*
Slow edit-compile-run cycle	CQE	1.8.1.1
No fine-grained changes	CQE	1.8.1.2
Expressiveness of the language	CQE	6.5.5
Limitations due to non-local transformations	Implementation	1.8.2.1
Limitations due to compiler optimizations	Implementation	1.8.2.2
Limitations due to the use of Regex	Implementation	1.8.2.3

1.8. Limitations

We discuss the limitations of our approach. We make a distinction between two kinds of limitations:

- Limitations due to *code query by example*.
- Limitations due to our *implementation* of code query by example.

Table 1.4 summarizes the limitations and points to related subsections.

1.8.1. Limitations of the approach

1.8.1.1. Slow edit-compile-run cycles

As explained in chapter 3, partners appreciate the rapid edit-compile-run cycles offered by the current Dynamics development environments. Recall that the Dynamics graphical interface has a dual role:

- It lets users *interact* with the ERP system (user can enter data through forms; users can aggregate data using reports, etc.)
- It lets programmers *modify* the ERP system (programmers can change existing forms; programmers can create new procedures, etc.)

Dynamics Using Dynamics, given the proper license, one has access to the development tools. The key point is that changes can be performed on the running system [88]. This is very different from traditional development environments, such as Visual Studio or Eclipse, where programmers typically: modify the program; compile; run the program; stop the execution of the program; modify the program again, etc.

Code exploration We observed in our empirical study, see chapter 3, that partners like to explore and experiment with the system, in order to acquire an knowledge of the base software product. Using the Dynamics development environment, partners can perform modifications on the running base system and immediately evaluate the effects of changes. Obviously, this approach would not work if every little change

would require a long recompilation phase, followed by a time consuming restart of the system.

CQE The approach that we propose, CQE, is closer to the traditional approach, with a slower cycle: partners change the queries; compile the queries; instrument the base software; implement the customizations, etc. One can consider this relatively slow cycle as a limitation of our approach, and more precisely a limitation that would force partners to change the way they work.

Run-time loading of customizations On the other hand, using CQE, note that once customization points are inserted, customizations can be added at run-time (during execution of the base software). This particular point brings, arguably, more flexibility.

1.8.1.2. Fine-grained changes

Example of a fine-grained change A fine-grained change is a change that must be applied to an explicitly mentioned location in the source code. For example, given the source code:

```
...
Console.WriteLine(customer.Name);
...
```

Assume that, in this precise location, and in this location *only*, one would like to change this statement to:

```
...
Console.WriteLine(customer.FullName);
...
```

If anticipation is possible Recall that one of our base hypothesis is that, with respect to ERP systems, *anticipation is difficult* (see section 1.4.3). If one does not take this hypothesis into consideration, there are , of course, a large range of customization technique that allows for fine-grained changes. Most of them use markers in the base code, in one form or another: for example, a call to an abstract method in the template method design pattern (see section 1.9.5). For example:

```
...
PrintCustomerName(customer); // call to an abstract method
...
```

Using CQE The problem with using explicit markers in the base code, is that customizations become fragile with respect to upgrades. One of the main objective with CQE was to strictly maintain *obliviousness* (see definition in section 5.2). It is obliviousness that allows us to improve resilience to upgrades (see section 1.6).

On the other hand, due to this design choice, CQE cannot deal with fine-grained changes. In this sense, CQE is only *complementary* to the other techniques that allow for fine-grained changes.

When fine-gained changes are required We already mentioned the template method pattern, that can be used when fine-grained changes are required and anticipation *is possible*. When fine-grained change are required and anticipation is *not possible*, probably the most convenient way to proceed with change is to simply perform customizations in-place, with the help of a version control system that can detect *some* conflicts upon upgrade [130, 46, 45], see section 1.9.3. On the other hand, it is important note that the notion of *conflict* is typically limited with traditional version control systems, for example they do not take into account type information.

1.8.2. Limitations of the prototype

It is important to make the distinction between the limitations of CQE, and the limitations of the prototype since, as we observed in section 1.7, matching can be implemented by other means, such as abstract syntax tree matching. We distinguish 3 categories of limitations in the prototype: limitations due to non-local transformations, limitations dues to compiler optimizations, and limitations due regexp matching.

1.8.2.1. Limitations due to non-local transformations

An local transformation is a transformation performed by the compiler in-place, meaning for example that a statement is not moved upon compilation outside of its enclosing method. Our prototype only addresses *local-transformations*. (Non-local transformations due to compiler optimizations will be discussed in the next section.) We look at a few C# constructs that cause non-local transformations. Once again, we refer the reader to section 1.3.1 for concepts related to C#, and to section 1.3.2 for concepts related to intermediate language (IL).

Anonymous methods Before C# 2.0, the only way to declare a delegate was to use a *named* method [135]:

```
delegate void Del(int x);
void Print(int i) { Console.WriteLine(i); }
...
Del d = (Del) Print;
```

Recent versions of C# allow programmers to write an *anonymous method* and to treat it as a value [65], for example:

```
static public void M(List<int> intList) {
 Action<int> Print = x => Console.WriteLine(x);
 intList.ForEach(Print);
}
```

LISTING 1.17: ANONYMOUS METHOD (CIL)

```
.method private hidebysig static void <M>b__0(int32 x) {
[...]
L_0000: ldarg.0
L_0001: call void [mscorlib]System.Console::WriteLine(int32)
L_0006: ret
}
.method public void M(class List`1<int32> intList) {
.locals init (
[0] class Action`1<int32> Print)
L_0000: ldsfld Action`1<int32> Test::CS$<>9__CachedAnonymousMethodDelegate1
L_0005: brtrue.s L_0018
L_0007: ldnull
L_0008: ldftn void Test::<M>b__0(int32)
L_000e: newobj instance void Action`1<int32>::.ctor(object, native int)
L_0013: stsfld class Action`1<int32> Test::CS$<>9__CachedAnonymousMethodDelegate1
L_0018: ldsfld class Action`1<int32> Test::CS$<>9__CachedAnonymousMethodDelegate1
L_001d: stloc.0
L_001e: ldarg.0
L_001f: ldloc.0
L_0020: callvirt instance void List`1<int32>::ForEach(class Action`1<!0>)
L_0025: ret
}

.field private static class Action`1<int32> CS$<>9__CachedAnonymousMethodDelegate1
[...]
```

Let's observe how csc, the Microsoft C# compiler [135], transforms the above statements, see listing 1.17. (For readability, namespaces and other relatively less important bytecode instructions were removed.) A method `b__0(int32 x)` was generated by the compiler, this method contains the body of the lambda expression in the first statement above. The compiler also generated a field of type `Action`1<int32>` to cache the delegate. The method body of M tests if this field is null, and if not, creates and stores an instance of an `Action` delegate using a pointer to the `b__0(int32 x)` method as constructor parameter (instruction label L_000e). The field is then placed in a local variable, the actual parameter of the method M is loaded onto the stack (the list of int), the local variable `Print` is loaded onto the stack (the delegate), and the instruction at label L_0020 calls the late-bound method `ForEach` on the list object.

Anonymous methods with captured variables The local variables and parameters whose scope contains an anonymous method declaration are called *captured variables* of the anonymous method [135, 65]. Let's look at a simple variant of the example given in the previous paragraph:

```
static public void M(List<int> intList) {
 int coefficient = 2;
 Action<int> Print = x => Console.WriteLine(coefficient * x);
 intList.ForEach(Print);
}
```

This code is transformed by the csc compiler into the code mentioned in listing 1.18 (only the bytecodes of interest are mentioned). The basic scheme, mentioned in the previous paragraph, remains the same, with one notable exception: an inner class is generated, which contains a field `coefficient`, corresponding to the captured variable. This field is assigned before the delegate is created (see the stfld instruction at label L_0008). Since the captures variable is turned into a field, does it impact further matching? it depends how the query is written. If the query is written as:

```
[Query("Q")]
public void M() {
 int coefficient = 2;
 [...]
}
```

.. the query engine will look for a statement that assigns the number 2 to a *variable* of type **`int`**. Hence, matching will impacted. On the other hand, if the query is written as:

```
[Query("Q")]
public void M(int coefficient) {
 coefficient = 2;
 [...]
}
```

... the query engine will look for a statement that assigns the number 2 to a *variable,* to a *field*, or to a *parameter* of type **`int`**. Hence matching will not be further impacted. Of course, since non-local transformations are not supported, this subtle point does not make a major difference.

Iterator blocks An iterator block is a block that yields an ordered sequence of values [65]. For example, the following iterator blocks yields the `n` first non-negative integers:

```
static public IEnumerable<int> First(int n) {
 for (int i = 0; i < n; i++)
  yield return i;
}
```

A client can, for example, use this iterator block to print the first 10 non-negative integers:

```
First(10).ToList().ForEach(Console.WriteLine);
```

Most of the logic to construct an iterator block, at the bytecode level, is done using a generated inner class that implements the `IEnumerable`, `IEnumerator`, and the `IDisposable` interfaces. This generated class also contains 6 generated fields for the given example. The goal of the class generated for an iterator block is to implement a state machine (to control the execution of the iterator block) [65, page 420]. We won't go into the details of the bytecode translation, it is sufficient to say that the transformation is clearly non-local and would require extensive support for matching.

LISTING 1.18: ANONYMOUS METHOD WITH CAPTURED VARIABLE (CIL)

```
.method public void M(class List`1<int32> intList) {
.locals init (
 [0] class [mscorlib]System.Action`1<int32> Print,
 [1] class Class3/<>c__DisplayClass1 CS$<>8__locals2)
[...]
L_0007: ldc.i4.2
L_0008: stfld int32 Class3/<>c__DisplayClass1::coefficient
L_000d: ldloc.1
L_000e: ldftn instance void Class3/<>c__DisplayClass1::<M>b__0(int32)
L_0014: newobj instance void Action`1<int32>::.ctor(object, native int)
[...]
}

// Inner class
 .class private beforefieldinit <>c__DisplayClass1  {
  [...]
   .method public   void <M>b__0(int32 x) {
     L_0000: ldarg.0
     L_0001: ldfld int32 Class3/<>c__DisplayClass1::coefficient
     L_0006: ldarg.1
     L_0007: mul
     L_0008: call void [mscorlib]System.Console::WriteLine(int32)
     L_000d: ret
    }
    .field public int32 coefficient
 }
```

1.8.2.2. Limitations due to compiler optimizations

In this section, we consider the impact of common compile-time optimizations on code matching – the risk that compiler optimizations will cause counter-intuitive results on code queries by transforming the program text (the target code and/or the query code) into an optimized but less appropriate form for code matching.

Most compilers, such the Microsoft C# compiler, csc, allow programmers to disable at least some compile-time optimizations. For example, using of command line switch [135]:

```
csc File.cs /optimize-
```

In some cases as we will demonstrate, the optimizations will not cause any problem since (following our approach) both the query code and the target code are compiled with *the same compiler*, hence are subject to *the same optimizations*; in some other cases, some optimizations being context sensitive will impact code matching if the code query does not reproduce the context in which the target code will be optimized, or dually if the target code does not reproduce the context in which the code query will be optimized.

Constant folding Constant folding is a common compile-time optimization that simplifies constant expressions [11, 5]. The terms in constant expressions can be literals, such as the integers 1 and 2 in the following example, or variables that are never modified.

```
void ConstantFolding(out int x, out string y) {
 x = 1 + 2;
 y = 1 + 2;
}
```

As we can observe in the IL generated by csc, the expression `1 + 2` was replaced by the load constant 3 instruction, at label L_0002. Similarly the expression `1 + 2` was replaced by the load string instruction `12`, at label L_0005.

```
static void ConstantFolding([out] int32& x, [out] string& y)
{
 L_0001: ldarg.0
 L_0002: ldc.i4.3
 L_0003: stind.i4
 L_0004: ldarg.1
 L_0005: ldstr "12"
 L_000a: stind.ref
 L_000b: ret
}
```

On one hand, since both the query code and the target code will be subject to the same constant folding, this particular optimization will not cause any problem during matching: the matching engine will match the constant `3` instead of the expression `1+2`, and similarly for the string `12`. On the other hand, if a `Func<int>` query variable is used in the query method, matching will be impacted.

Constant propagation Constant propagation is another common compile-time optimization that substitutes the value of known constants in expressions at compile time [11, 5].

```
const int two = 2;
int ConstantPropagation() {
 int x = two;
 return two;
}
```

Matching is impacted, since, as we can observe in the generated IL code, the compiler propagated the constant `2` to the local variable `x` (the field `two` was not loaded). The constant was not further propagated by the compiler: the integer constant `2` is explicitly stored in local variable `x`, which is then loaded onto the stack.

```
static int32 ConstantPropagation()
{
  .locals init ([0] int32 x)
  L_0000: ldc.i4.2
  L_0001: stloc.0
  L_0002: ldloc.0
  L_0003: ret
}
```

Note that when the field two is not declared as **`const`** but as **`static`** `readonly` [65], the compiler does not perform any constant propagation, and explicitly issues a load field (ldsfld) instruction to load the field two in the local variable x.

Common sub-expression elimination Common sub-expression elimination looks for expressions that evaluate to the same value and replace them with a single variable that stores that value [11, 5]. For example given listing below, a compiler may consider it worthwhile to store the value of the sub-expression `(x*y)` in a local variable, and to substitute the occurrences of that sub-expression by that local variable.

```
static int CommonSubexpressionElimination(int x, int y, int m, int n, int o)
{
  int a = (x * y) + m;
  int b = (x * y) * n;
  int c = (x * y) / o;
  return a + b + c;
}
```

Based on our experiments, the csc compiler does not perform common sub-expression elimination; we suspect that this is handled by the runtime. It is documented that the mono runtime implements PRE (Partial Redundancy Elimination) in runtime [142], on the SSA (Single Static Assignment) code representation [108]. (PRE is a form of sub-expression elimination that eliminates expressions that are redundant on some but not necessarily all paths through a program.)

Inlining Inlining replaces a function call site with the body of the callee [11, 5]. This optimization tries to improve runtime performance at the cost of increasing the

program size. Performance improvement comes from avoiding both the function call and return instructions and by making possible other optimizations on the inlined method body (constant propagation for example).

```
static bool IsEven(int x) {
  return (x & 1) == 0;
}
static bool Inlining(int m) {
  return IsEven(m); // (m & 1) == 0;
}
```

Judging from our experiments, the csc compiler does not perform those operations, even when given the optimization flag. It is possible that inlining is performed by the just-in-time compiler [135] (JIT) (Section 1.3.2 summarizes the compilation and execution phases.)

Dead code elimination Dead code elimination removes the statements that do not affect the program, hence reducing the code size and avoiding to execute unnecessary operations [11, 5]. Some part of the code is said to be dead if it is not reachable, or if it affects only dead variables. Note that the csc compiler will compile the code below, but will emit three warnings: the first one states that the variable y is assigned but its value is never used, and the second and third ones state that unreachable code is detected.

```
static int DeadCodeElimination(int x)
{
  int y = 2; // unused
  if (false)
    return 0; // unreachable
  return x;
  y++; // unreachable
}
```

In release mode, the csc compiler nicely optimize the code above by removing the unnecessary statements:

```
static int32 DeadCodeElimination(int32 x)
{
  L_0000: ldarg.0
  L_0001: ret
}
```

In debug mode the Microsoft C# compiler keeps the dead code in the IL. If deadcode elimination is performed only in the target code, or only in the query code, matching will obviously be impacted.

Loop unrolling Loop unrolling tries to improve the program execution speed [11, 5], but typically makes the size of the program text larger. The speed increase is gained by avoiding the loop test at each iteration, and the increment on the counter variable. For example the following listing:

```
int x = 1;

for(int i= 0; i < 4; i++)
  x *= x;
```

.. can be optimized into this one:

```
int x = 1;
x *= x;
x *= x;
x *= x;
x *= x;
```

Based on our experiments, the csc compiler does not perform loop unrolling.

Summary of compiler optimizations We experimented with the csc compiler [135] to evaluate whether 6 common compiler optimizations impact bytecode matching when given the optimization flag. Judging from our experiments, dead code elimination, constant propagation, and constant folding impact bytecode matching using the csc compiler.

1.8.2.3. Limitations due to the use of regular expressions

Expression query variables In CQE, a query variable of type `Func<R>` denotes an expression of type `R` in the target code, see section 1.5.2.13. Type information of variables is directly available at the bytecode level; type information of sub-expressions is more difficult to recover. For example, using CQE, programmer could express the following query:

```
[Query("Q")]
void Q1(int x, Func<int> f) {
  x = f() + f();
}
```

This query should match the first line of method M:

```
void M(int m){
 int n = (m+m) + (m+m);
 ...
}
```

This statement is translated into the following bytecode sequence:

```
...
ldarg.1
ldarg.1
add
ldarg.1
ldarg.1
add
add
```

```
stloc.0
...
```

This is not directly amenable to regex matching: bytecode matching would require a more complex pre-processing phase that recover the locations and the type of the sub-expressions, perhaps doing an interpretation similar to the abstract interpretation of stack height that we perform to detect statement locations, see section 1.7.2.3.

Try/Catch/Finally statements Consider the following example :

```
public void M() {
 try {
  Console.WriteLine("try");
 } catch(Exception ex) {
  Console.WriteLine("catch");
 } finally {
  Console.WriteLine("finally");
 }
}
```

The csc compiler will translate the above example into the bytecode mentioned in listing 1.19. On the one hand, simple queries, like the following, will correctly match the corresponding code fragments in a try/block/finally statement:

```
[Query("Q")]
void Q1(string s) {
  Console.WriteLine(s);
}
```

On the other hand, since structure of the try/catch/finally block is not completely inlined in the method body (a part of it is at the end of the method, using a special clause, `.try`), the current implementation does not fully support try/catch/finally matching. It is possible that a more complex pre-processing phase could deal with this problem, since all the required information is present at the IL level.

LISTING 1.19: TRY/CATCH/FINALLY (CIL)

```
.method public hidebysig instance void M() cil managed {
.maxstack 1
L_0000: ldstr "try"
L_0005: call void Console::WriteLine(string)
L_000a: leave.s L_0019
L_000c: pop
L_000d: ldstr "catch"
L_0012: call void Console::WriteLine(string)
L_0017: leave.s L_0019
L_0019: leave.s L_0026
L_001b: ldstr "finally"
L_0020: call void Console::WriteLine(string)
L_0025: endfinally
L_0026: ret
.try L_0000 to L_000c catch Exception handler L_000c to L_0019
.try L_0000 to L_001b finally handler L_001b to L_0026
}
```

1.9. Further software customization techniques

The full spectrum of software customization techniques is large. It ranges from unstructured and un-controlled *in-place modifications*, all the way up to highly-structured and controlled *refinements steps* in formal methodologies [4]. In chapter 2, we discuss 7 different techniques to proceed with customization, and we compare them according to 4 different criteria. In this section, we study further existing customization techniques with respect to the upgrade problem: procedural abstraction, version control systems, assembly versioning and design patterns.

Challenge We emphasize that existing customization techniques are often *not mutually exclusive*, these techniques *tend* to be used together. Each community has a particular focus: large software framework makers (such as the Eclipse community [64] or the .Net framework team [47]) typically emphasize *design patterns*, the operating system community relies extensively on *version control systems* [46] and *patches* [121], the embedded systems and the proof community is often concerned with *refinement* [4, 211], etc. Ideally we would like to study not only each of those techniques individually, but also *how they interact*.

1.9.1. In-place modifications and procedural abstraction

Perhaps the simplest way to perform customizations is to make changes *in-place*, directly in the source code, wherever change is required. This is how many software products deal, in practice, with customization. More precisely, a subset of the source code is given to partners, who are free to make the changes they need [88], see for example our description of customizations in Dynamics NAV, section 2.3. As noted in chapter 2 and 3, experienced Dynamics programmers try to minimize their (invading) in-place customizations by invoking their own procedures: in the best case the change is just a line for the method invocation, in practice it is typically a conditional statement that calls the procedure when required. Below `F1` and `F2` are customization methods; the method calls are simply inserted in the base code.

```
...
Partner1.F1(...);
basecode;
Partner1.F2(...);
...
public static class Partner1 {
 public static void F1(...) {...}
 public static void F2(...) {...}
}
```

The advantages of this approach are:

- Simplicity.
- Little reliance on anticipation.

- Enabling fine-grained modifications.

Why is there still *some* reliance on anticipation? It has to do with *procedural abstraction* [3, page 26], arguably the simplest form of abstraction: when a base software maker notices a repetitive pattern in his code, or a pattern that is *likely* to be customized, he forms a *named* procedure (a form of *abstraction*) that contains that pattern and calls this procedure from the places where the functionality is needed. Since the code is *factored-out* into the named procedure, changes, in the best case, can be performed in only one location, hence upgrade will be facilitated. This scheme is simple and effective – but of course, changes *must* be anticipated. More precisely, the base software maker must anticipate (a) where to call this customization method (b) what will be the formal parameters of this customization method (or more generally how much side-effect can a customization method have).

The disadvantages of in-place customizations are that:

- The programmers implementing the change have to know the system very well – this approach does not scale (see our remarks on information hiding and local reasoning in section 1.4, and the conclusions of our empirical study in section 1.2.6).
- The changes are very sensitive to upgrade, since the code that was modified can be potentially modified in the new version of the base product.

One (large) central customization method It is useful to consider an extreme hypothetical approach: consider a designated method, that would be called by the software product on special occasions. Partners would be given the right and the means to modify that designated method, and hence be given the opportunity to customize the behavior of the software product. Since this *central method* would be invoked from a large number of client methods, perhaps tens of thousands, it would be difficult to find a list of formal arguments that would be convenient for all of them. As a workaround, the designer of the software product could think to give to this central method a signature of type:

```
  public static void Customize(string customizationName, object[] args
)
```

Methods that use this central method are then free to pass the arguments they wish for, using an object array. After some time, we can imagine that this method would contain a large number of conditional statements and downcasts:

```
void Customize(string customizationName, object[] args) {
 if(customizationName == "OnNewInvoice") {
   var invoice = (Invoice) args[0];
   var product = (Product) args[1];
   ...
  }
 else if(customizationName == "CancelInvoice") { ... }
 ...
}
```

Let us evaluate this approach. What would be the advantage of this solution? Arguably:

- Its simplicity: it is a low-tech approach; one could claim that this approach is easy to understand.
- All the customizations are textually centralized: upon upgrade, only one location – possibly only one file – must be changed.

Now we shall look at the disadvantages of such an approach; the list is long, and we shall only give part of it here:

- Weak type safety: (a) the base code can change the string value the first actual argument upon evolution, in which case, the customization would no longer be triggered (there is also the risk that the *wrong* customization is now triggered). In the best case, this would be discovered during testing; in the worse case, this would be discovered when the system is in production (b) since the other arguments are encoded in a one-fits-all object array, the type system becomes of little use, and the downcast operations are likely to create type mismatch exceptions upon upgrade.
- It does not scale in term of complexity: there is little abstraction, this customization method will become quickly difficult to understand and maintain.
- One needs to anticipate where to call this customization method.

1.9.2. Covariance

Covariance and contravariance have been discussed thoroughly in the literature, see for example Castagna [36] for a good introduction. This topic is important with respect to object oriented software extension, and is closely related to extension of the base software product by partners using inheritance, see section 2.5.1. We summarize the main issues, building on the concepts defined in section 1.3.1.

In C#, methods cannot change the type of the formal parameters of the method they override, neither their return type. However, using delegates, C# 3 allows for *covariance* in the return type and *contra-variance* in formal parameters types [65, page 364], see listing 1.20.

These variances are type safe, but unfortunately give little freedom with respect to extensibility. Some languages, like Eiffel, trade type safety for extensibility and allow for covariance in the return type *and* in the parameter type [134]. Type safety is then sacrificed [36, page 3], but the language allows for convenient subclassing with more precise redefinition of formal parameter types [32, 36]. Take for example the following two classes, that could belong to the base software product:

```
public class Point { ... }
public class Screen {
  public virtual void Print(Point p) { ... }
}
```

LISTING 1.20: VARIANCE USING DELEGATES

```
[...]
class A { }
class B : A { }
static void TakesA(A a) { }
static B ReturnsB() { return ... ; }
static void F(){
 Func<A> a = ReturnsB;
 Action<B> b = TakesA;
}
```

When new hardware technology brings us *color screens*, partners would like to introduce a customization to print on the screen *color points*. At the core of this problem, we have two parallel hierarchy of classes that need to co-evolve, the screen hierarchy and the point hierarchy. What partners could wish for, in this case, is the following:

```
public class ColorPoint : Point {
 public RGB Color {get; set;}
}
public class ColorScreen : Screen {
  public override void Print(ColorPoint p) { ... }
}
```

... where the classes `ColorPoint` and `ColorScreen` are introduced by partners. This is not directly possible in C#, as it would sacrifice type safety. Consider for instance the following listing:

```
static public void Display(Screen s, Point[] points) {
  foreach(var p in points)
    s.Print(p);
}
```

The array `points` can contain instances of both `Point` and `ColorPoint`. When the method call `s.Print(p)` is dynamically bound to the method `ColorScreen.Print(ColorPoint)`, the access to the property `Color` would create a run-time error, if the variable `p` is not of type `ColorPoint` [36, page 8]. A recent version of C#, version 4.0, allows for covariance in output positions through the use of the `out` keyword on formal generic parameters of interfaces [194]; since covariance is only allowed in output positions, this language evolution is type safe [32, page 26], but does not solve the above problem.

Summary of covariance Covariance would allow partners to redefine existing methods of the base software product with more concrete formal parameters, possibly using types that they have themselves introduced. Such a language features would be an important facilitator of extensibility, but would sacrifice type safety.

1.9.3. Version Control Systems

Version control systems (also called *revision control systems*) are tools dedicated to manage change over time of (software) artifacts among a set of possibly loosely connected developers [45]. Versioning control systems, can be seen as an extension of methods based on in-place modifications, where one tries to reap the benefits (fine-grained modifications, simplicity of the approach, etc.), without suffering from its deficiencies: in our case potential conflicts between customizations and evolution. This field has received a lot of attention, especially from the industry, where those tools are used extensively. Subversion, for example, is a popular and modern version control system [45] . (Note that we touch here upon *code versioning* techniques, *assembly versioning* will be discussed in section 1.9.4.)

An analogy A version control system can be seen as a form of *concurrency control system*, where the data is the source code, and the thread of controls are the programmers. In a concurrency control system, one can differentiate between two basic approaches [93, 130]: *optimistic concurrency control*, and *pessimistic concurrency control*.

- Following the pessimistic approach, all participants work on the same set of software artifacts, and the system tries to *prevent* conflicts from happening, typically using *locks* to control access to resources [130]. In our case, the unit of granularity of locked data could be a C# file.
- Using an optimistic form of concurrency control, all users can make changes *on their own copy* of the data [93, 130], and conflicts must be resolved when changes are reconciled, in our case when changes are merged. Using this scheme, the system does not *prevent* conflicts from happening, but rather *resolves* conflicts whenever they are detected. The notion of conflict can vary [130], a conflict could be for example when the same line of code, or the same character, was changed by more than one programmer. One can come up with a different definition of conflict, for example using the method or the class as the unit of granularity, or even a more elaborate one that would use the notion of a slice [206].

Advantages and disadvantages The advantages of version control systems are:

- All the benefits that we listed in the context of in-place modifications.
- Reverting changes to earlier version is easy, since the system keeps an history of changes, or an history of versions [45, 46].
- Integrated diff and merge tools typically give the programmer some help in case of a conflict [121]. Again, the definition of the notion of *conflict* depends on the tool being used.

- The disciplined usage of branches can isolate and group a set of related changes [45, page 48].
- One can easily come up with a *configuration* that decides upon which version of which file to use [46]. Conradi differentiates between extensional versus intensional versioning [46]. Extensional versioning gives an enumeration of versioned items, whereas intensional versioning uses a predicate to define membership [46]. This closely resembles the distinction that we made between extensional customization point definition and extensional customization point definition, see section 4.3.2 – with the notable exception that, in our case, we decouple a customization *point* from the *implementation* of the customization, see section 1.5.

Among the disadvantages, one can point out that:

- Successful (textual) merging can provide a false sense of confidence among junior programmers that automatically merged customizations will compile correctly, or even function correctly. Since traditional version control systems have little knowledge about the particular language of the versioned items, they cannot ensure, for example, type safety upon merging.
- If pessimistic concurrency control is used, one can potentially have to wait for an other programmer to release their locks to be able to proceed with one's own modifications. This strict consistency model is inadequate for large number of collaborating developers [130, page 1].
- If optimistic concurrency control is used, conflicts upon merging will have to be dealt with, typically manually [45, page 37].

In the defense of version control systems, those last two problems can be seen as an intrinsic difficulty of shared state in a concurrent context – version control systems provide tool support to deal with those problems. One should also note that modern version control tools, such as Subversion [45], also provide an extensive set of features, that, in practice, turn out to be very useful: for example, access right managements, history logs, branch and patch management systems, etc.

Three kinds of merges The concepts of *merge*, *diff* and *patches* are strongly associated with version control systems, but can also be used independently [121]. Conradi differentiates between three kinds of merges [46]: raw merging, two-way merge, and three-way merge. Raw merging simply applies a change in a different context [46]. Two-way merge compares two versions of an artifact and merges them into a single version. In addition to the two versions of a artifact, three-way merge uses their common ancestor. Mens further differentiates between textual, syntactic, semantic and structural merging [130]. Most of the popular merge tools are textual.

LISTING 1.21: DIFF AND PATCH OPERATIONS

```
$ echo 'A' > file1
$ echo 'B' > file2
$ diff -u file1 file2 > seb.patch
$ cat seb.patch
--- file1       2009-05-25 00:13:42.000000000 +0200
+++ file2       2009-05-25 00:13:48.000000000 +0200
@@ -1 +1 @@ -A +B
$ patch < seb.patch
patching file file1
$ cat file1
B
```

Diff and patches The diff command compares two files, and reports differences between them (if this report is turned into a file, the resulting file is called a *patch*). The patch command understands these differences as modifications to make to a file, see for instance listing 1.21.

More advanced diff and patch tools have language specific or data type specific functionalities, see for example Beyond Compare [27], that has specialized viewers for a variety of data types (HTML, images, etc.), and WinMerge [210], that can compare Excel files using a dedicated plug-in.

Diff3 and 3 way merges Diff3 can be seen as more powerful variant of the diff tool illustrated previously [109]: when two programmers have made changes to the same file, Diff3 can incorporate changes from two modified versions into their common preceding version. Naturally, the command has to be provided with 3 files: the two modified versions, and their common preceding version. The upgrade problem could be framed in the context of a 3 way merge: for example, if one consider figure 1.5, P_1^1 is the common preceding version of P_2^1 and P_1^2.

CQE Code query by example (CQE), see section 1.5 does not try to merge two different versions of an artifact, but rather re-apply code quantification upon upgrade, see section 1.6. (Code query, using CQE, provides an intensional definition of customization points using matching: the customization points are defined by the matched code fragments.)

Advances in the recent years The open source community has been very prolific in the recent years concerning version control systems, making, for instance, advances regarding *decentralized* repositories. Some interesting projects:

- The Mercurial project [133]; its main particularity is that it can be completely decentralized, and hence imposes little policy on how people ought to work

with each other. This, interestingly, is very close to the network of ERP partners described in section 2: a web of loosely connected businesses sharing source artifacts. Mercurial allows for deep control over which changes are incorporate into one's build tree.

- The Git project [86]; Git is a performance oriented version control system, which has strong support for non-linear development (branching). Like Mercurial, it allows for distributed development: each developer has a local copy of the entire development history, and changes are copied from one such repository to another.
- The Darcs project [50]; this is another interesting, and recent, version control system, based on a so-called *theory of patches,* which focuses on patch composition [51]. This is a relatively new project, but it would be interesting to further study how patch composition, as defined by Darcs, could apply to the specific context of the upgrade problem.

1.9.4. Assembly versioning

A central building block When the .Net framework was conceived, assembly versioning received a lot of attention, in particular in order to face the *DLL hell problem*, see section 2.2.4. According to Microsoft official documentation, assemblies are nothing less than *"the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions."* [135]. Two of those subjects are of a direct interest to us: *version control* and *reuse*. The other themes mentioned are also related to the upgrade problem. Hence, it is worth examining what assemblies can provide.

Roles of an assembly Obviously, the main role of an assembly is to contain the CIL code that was mentioned in section 1.3.2. Perhaps less obviously, an assembly:

- Forms a security boundary: "an assembly is the unit at which permissions are requested and granted" [135].
- Forms a type boundary: "every type's identity includes the name of the assembly in which it resides" [135].
- Forms a reference scope boundary: "the assembly's manifest contains assembly metadata that is used for resolving types" [135].
- Forms a version boundary: "the assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit." [135].
- Finally, forms a deployment unit.

Assembly versioning and upgrade We found in our empirical study that one of the main reason behind the upgrade of Dynamics products was to benefit from the latest error fixes, see chapter 3. Hence, one could attempt to ease the upgrade problem by promoting a more systematic use of assembly versioning and assembly deployment for error fixes within Dynamics. For example, by giving a concrete semantics to assembly strong names (each strongly named assembly carries versioning information structured into four fields: Major, Minor, Revision and Build – alas with no clearly defined semantics). By changing the granularity of updates, one can hope to reduce the efforts to port existing customizations. This is only an hypothesis, here more research is needed – perhaps using a formal framework such as the one conceived by Eisenbach et al. [68], based on the Alloy modeling language [8]. Stuckenholz [189] gives a good overview of the state of the art in assembly versioning.

Anticipation To be able to benefit from efficient upgrades at the assembly-level, the base software maker has to anticipate what would be the proper decomposition of types into assemblies – concretely, which types should belong to which assembly. Ideally, only a few small assemblies would change upon an error fix release. This is another instance of the modular decomposition problem introduced in section 1.4. In this case, the unit of modular decomposition is the assembly.

1.9.5. Design patterns

Some of the most frequent solutions to recurrent problems have been abstracted in the form of design patterns. Many of them deal, in one form or another, with customization: good design is an important facilitator of customizability. We refer the reader to Gamma and Martin for a good introduction to design patterns [83, 125]; we review here a small number of design patterns and discuss their applicability to the upgrade problem. Once again, we assume that the reader is familiar with the concepts defined in section 1.3.1.

Template method and strategy The *template method pattern* and the *strategy pattern* are closely related. Both allow to separate a generic algorithm from a specific context [125]. The former uses inheritance, and the latter delegation. The main idea is that part of a given algorithm, a method body, can be fixed while some particular steps in this algorithm can be abstracted. Template method simply uses abstract methods: methods in subclasses provide a specific implementation of a step in the generic algorithm, see listing 1.22.

The strategy pattern uses a more elaborate approach: contrary to the template method, the generic algorithm is not placed in an *abstract* base class, but rather is placed in a *concrete* class [125]. The abstract method, that the generic algorithm calls, is defined in an interface, see listing 1.23. Here, `steps` is a formal argument of the method `Algorithm`, but could also be a formal argument of a constructor of the concrete class `Program` [83, page 320]. Using .Net delegates, one can very easily come

LISTING 1.22: TEMPLATE METHOD PATTERN

```
public abstract class Program {
 protected abstract int StepA(int x);
 protected abstract int StepB(int x);

 public int Algorithm(int input) {
  [...]
  var x = StepA(some_int);
  [...]
  var y = StepB(some_other_int);
  [...]
  }
}
public class ConcreteProgram : Program { [...] }
```

up with a variant of this pattern, such that each step can be specified independently, see listing 1.24.

Using the concise notation of *lambda expressions* to define delegates [65], the method above can be invoked as shown in the following method F:

```
int F() {
 Func<int,int> AddOne = (x) => x + 1;
 Func<int,int> SubstractTwo = (x) => x - 2;
 return Program.Algorithm(AddOne, SubstractTwo, 10);
}
```

The method F can be itself abstracted so that the delegates are not defined directly in the method body of F, but are rather defined by another higher order function passed as a parameter to F, for example:

```
void F<T>(Func<T,Func<int,int>> g, T t1, T t2) {
 var result = Program.Algorithm(g(t1), g(t2), 10);
}
```

This trivial example shows how delegates can improve the standard design patterns, as initially defined by Gamma et al. [83].

How does these design patterns relate to the upgrade problem? In the case of the template method, the method `Algorithm` takes the role of the base software product, and the implementation of the abstract methods take the role of the customizations. In the case of the strategy pattern, the implementation of the interface (or the delegates if the programmer prefers the higher-order variant) takes the role of the customization. Let us evaluate the patterns that we just described. On the positive side, we note that:

- It is easy for the base software product maker to control which method can be customized (controlled with its use of calls to abstracts methods, with its use of abstract formal arguments, etc.) [83, page 326].

LISTING 1.23: STRATEGY PATTERN

```
interface ISteps {
 int StepA(int x);
 int StepB(int x);
}

public class Program {
 public static int Algorithm(ISteps steps, int input) {
   [...]
   var x = steps.StepA(some_int);
   [...]
   var y = steps.StepB(some_other_int);
   [...]
  }
}
```

- Using the strategy pattern, it is easy to change the algorithm at run-time [125], since customizations are specified using actual arguments upon method invocation.
- Upon upgrade, if the signature of one of the abstract method changes, the compiler will detect, at compile time, any type mismatch with customization definitions in subclasses (or in subtypes in the case of the strategy pattern). Similarly, added or removed abstract steps in the generic algorithm will cause a compile time error if the customization was not adapted to the newer version of the base software product.

Now let's look at some of the disadvantages of those patterns:

- Partners cannot change the order of the steps of the algorithm [125, 83], since

LISTING 1.24: TEMPLATE METHOD: HIGHER-ORDER VARIANT

```
public class Program {
 public static int Algorithm(Func<int,int> a,
                              Func<int,int> b,
                              int input) {
   [...]
   var x = a(some_int);
   [...]
   var y = b(some_other_int);
   [...]
  }
}
```

the locations of the abstract steps are hard-coded in the method body. In the example above, `StepA` must always precede `StepB`.

- As we illustrated, the customization steps and their locations must be anticipated.
- A non-negligible amount of wiring machinery is needed: for example, in the case of the template method pattern, an instance of the appropriate subclass must be created whenever necessary (using, for instance, a variant of the factory pattern described below).
- The customizations are difficult to reuse, since the customizations have a dependency with an abstract class, or with an interface, which is part of the base software product. The template pattern is particularly rigid since the concrete class that implement the customization can only be used for one template method; the strategy pattern is more flexible in the sense that the same customization can be reused in various locations of the base software product, since the customization has no dependency with the customization point where it is being used. This argumentation closely relates to the recommendation of Gamma et al. to "favor object composition over class inheritance" [83, page 20].

Factory and service locator Factory is a so-called *creational pattern*, a pattern focused on abstracting instantiations . Creation statements are a key aspect of evolvability: whenever a programmer writes `var t = new T();` he creates a dependency with a type `T`. If a programmer wants to replace all instantiations of `T` by `T2`, a subclass of `T` , he has to modify many different statements all over his software text. Like the other patterns, Factory pattern has many variants, such as *Factory method* and *Abstract factory [83].* A simple example of a factory method is the static method `File.Open` in the .Net framework [47, page 333], that creates instances of file streams:

```
public class File {
 public static FileStream Open(String path, FileMode mode) { ... }
}
```

A more complex factory pattern allows programmers to create instances of object while depending only on a *abstract* factory interface [125, page 439], see listing 1.25.

We note the following problems with this pattern:

- The problem of creating an instance of a `ICustomization` (and with it, dependencies to types `CustomizationA`, `CustomizationB`, etc.), has only been factored out into the classes that implement the interface `ICustomizationFactory` .
- Every time one wants to add a new kind of customization, both the interface `ICustomizationFactory` and all its concrete implementations have to be modified [125, page 440].

LISTING 1.25: FACTORY PATTERN [125]

```
public interface ICustomization { ... }
public class CustomizationA : ICustomization { ... }
public class CustomizationB : ICustomization { ... }

public interface ICustomizationFactory {
 ICustomization MakeCustomizationA();
 ICustomization MakeCustomizationB();
}
```

- Factory methods are often limited to creating instances of specific type determined at design time [47].
- Factory patterns can hinder usability: based on their experience designing the base classes for the .Net framework, Cwalina et al. claim that constructors are more "usable, consistent, and convenient than specialized construction mechanisms". Furthermore, they note that Intellisense has typically little support for factory methods. Hence they recommend .Net programmers to prefer constructors to factories [47, page 333].
- Finally, once again, the designer has to anticipate the need for a factory pattern, the structure of the interface `ICustomizationFactory` must be anticipated, and the location where an instance of an `ICustomizationFactory` is used must be anticipated.

Service locator can be seen as a variant of a factory pattern [165]. The basic idea of a service locator is to have an object that knows how to get hold of all of the services that an application might need. The service locator has to be configured, for example using a setter [79]:

```
ServiceLocator locator = new ServiceLocator();
locator.loadService("ICustomization", new CustomizationA());
ServiceLocator.load(locator);
```

The service locator can use, for example, a map to link the name of the service (the key) to a concrete instance. Clients use the key to get hold of a service:

```
var sf = ServiceLocator.GetService("ICustomization");
```

The use of a key is not mandatory, one could use explicit methods instead [79]. The service locator is arguably easier to use (for the restricted case of service location) than the abstract factory pattern, but it suffers from the same deficiencies: the dependency with the concrete type of the services has just been moved, the required flexibility must be anticipated, etc.

In summary, the factory and the service locator patterns can add flexibility to the enterprise systems considered by abstracting instantiations of customizations, but

LISTING 1.26: EXTRACT FROM THE VISITOR PATTERN [83]

```
public interface Element { void Accept(Visitor v); }
public class ConcreteElementA : Element {
 public void Accept(Visitor v) {
   v.VisitConcreteElementA(this);
 }
  ...
}
public interface Visitor {
 void VisitConcreteElementA(ConcreteElementA e);
}
public class ConcreteVisitor1 : Visitor { ... }
```

add a non-negligible amount of code artifacts. Furthermore, they require anticipation from the base software maker on how and where to be used.

Visitors The intent of the visitor pattern is to represent an operation to be performed on the element of an object structure, without modifying the existing classes [83, 125]. In our case, the existing hierarchy of classes is the base software product. Once again, one can find a wide range of variants around this pattern, we will only comment on one of them. Listing 1.26 shows an extract from the visitor pattern [83].

The visitor pattern distinguishes two class hierarchies [83, page 333]:

- The hierarchy for the elements being operated upon (the Element hierarchy).
- The hierarchy for the visitors that define operations on the elements (the Visitor hierarchy).

A new operation is created by adding a subclass to the Visitor hierarchy. There is typically one abstract method in the Visitor interface for each concrete class in the Element hierarchy. (Listing 1.26 only shows one concrete element, and one concrete visitor.) At the heart of the visitor pattern, one finds a double dispatch (also called dual dispatch [125, page 548]): the first dispatch is done upon the call to the `Accept` method which resolves the concrete type of the element; the second dispatch is done by the method call to the `Visit` method, which resolves the method to be executed on the visitor. In the context of ERP systems, visitors could be customizations written by partners, and the Element hierarchy could be part of the base software product.

Among the advantages of the visitor pattern, one notes that:

- It is easy to add new operations [83, page 335].
- One can localize related behavior in a visitor [83, page 335].

The main disadvantages of using a visitor pattern are:

- Adding new concrete element class is hard, since potentially a large number of visitors classes have to be modified accordingly [83, page 336]. This point can be problematic for ERP systems since the addition of new types is one of their requirement, see chapter 2.
- The concrete elements must be designed in such a way that their members allow the visitors to do their job [83, page 337].
- Of course, the designer must anticipate the need for a visitor pattern and must design the element classes accordingly (addition an interface with an `Accept` method, etc.).

1.9.6. Summary

All the software customization techniques that we reviewed rely on *anticipation*, with two notable exceptions: simple in-place modifications, and aspect-oriented programming (see chapter 2). The former is very sensitive to upgrades, and the latter is very much related to our proposal (the relation between *code query by example* and *aspect-oriented programming* is studied in chapter 5). The use of version control systems also requires little anticipation, and can be considered as tool support for the other language-based techniques.

1.10. Lessons learned and discussion

In this section, we give a more personal perspective on the upgrade problem, on ERP systems, on research with an industrial partner, and on research on enterprise systems. This section is discussion-oriented, and uses an informal tone.

1.10.1. The upgrade problem, gathering our wits

The greatest common factor The upgrade problem can be framed in the context *reuse*. Many OO concepts, such as interfaces, abstract classes, inheritance, dynamics binding, etc., can all be seen as facilitators of reuse. People go to great lengths to promote reuse and evolvability: some are willing to endure the complex class hierarchies caused by *multiple inheritance* [134], others write large books that document *design patterns* [83]; some carefully design *binary packages* that can be changed individually [189], others create metrics to *measure* reusability [125], etc. In our humble opinion, the greatest common factor behind those techniques is *anticipation*. One has to anticipate whether a class should be abstract or concrete; whether an interface should be constructed or not; whether a method should be virtual or non-virtual; whether a factory pattern should be used or not; whether an event should be raised or not; whether a post-condition or an invariant should be weak or strong; whether two types should be in the same binary package or be separated, etc. Once *anticipation*, this pillar of software engineering, is challenged, many of those well-accepted concepts are shaken.

The ability to anticipate is useful Programmers need *local-reasoning* to be able to fight the complexity of software development. Evolvability has been traditionally the main criteria that drives *decomposition* [158]: by *programming to an interface* a client is not dependent on a concrete implementations [83]; through the use of *events*, subscribers can be added and removed at will [47]; by using a smart *binary packaging* [189], one can update only a small part of its software product; by specifying that a method is virtual and using a factory pattern, one can leverage *dynamic-binding* [83]; by using the proper *post-conditions* and *invariants*, one can ensure that some properties are maintained by subclasses [92]; by using an abstract class, one can enforce concrete subclasses to implement newly added abstract methods [47]; by using a *visitor pattern*, one allows future addition of operations on a stable class hierarchy [125], etc. Since decomposition has to take place anyhow, one can try to make an educated guess [158] with respect to future needs for customization and evolutions.

Modern times Standard software development methodology requires anticipation [158, 159]. Interestingly, modern development practices are *more cautious* in this respect. For example, the Agile development community recommends to *"resist premature abstraction"* [125]. Jeffries discusses the cost what he calls *"anticipatory design"* [101] and defend the Extreme Programming idea, in his words, to *"do the*

simplest thing that could possibly work". The same subject has been discussed by the community under the term *"premature generalization"* [34]: an abstract class with only one descendant, or a library which is used by only one program, are possible inappropriate generalizations. Of course, when anticipation is possible, abstraction should be used, and dependencies should be avoided [125] – what should be avoided are arbitrary attempts at anticipation. The philosophy of those modern methodologies is to keep the design as simple as possible, and to *embrace change*, helped by refactoring tools. It seems difficult to apply this strategy with ERP systems: a partner cannot undertake the responsibility to refactor such a large software product for his needs. And if he did, it is likely that he would only make his job harder upon upgrade.

Stable API, or not Given a standard plug-in or add-in model, one of the core aspect of the problem comes back to having a stable interface. Since the foundational work by Parnas and others [159, 158], the basic notion of an interface has now been extended with the notion of a behavioral specification [92]. As long as the interface is supported by the vendor, the add-in will not break upon upgrade (or at least is not suppose to break). Dynamics is *not* a standard plug-in model: changes are made *inside* the implementation, *not against* the Dynamics API, see section 2. (From this perspective, the Dynamics development model is actually closer to collaborative developments using version control systems). The dual of a stable API, having a "stable Dynamics product" would mean no evolution at all, or at least no evolution of the customizable part of the software product.

Evolution payoff The Linux kernel deal with *internal* API in an interesting way: the Linux kernel has *no stable internal interface* [115]. "If your driver is in the [Linux source code] tree, and a kernel interface changes, it will be fixed up by the person who did the kernel change in the first place. This ensures that your driver is always buildable, and works over time, with very little effort on your part" [115]. Could Dynamics leverage the same development model? Partners would check-in their customizations, and it would be up to the base software maker to take responsibility for his evolutions, meaning fixing existing customizations. It *seems* un-realistic, for example due to intellectual property rights issues, but the idea should not be completely dismissed. In particular, it would have the important effect of increasing the cost of an evolution, and hence it would deter the base software maker from making many changes. This strategy is referred to, in game theory and political science, as *changing the payoffs* (to promote co-operation) [15, page 133].

Quantification and decomposition Decomposition using quantification (see definition in section 1.3.3) comes on-top of the existing static decomposition, see section 1.5; it changes the *stage* of some decomposition decisions: decomposition is not completely done at design time but some of it is delayed for *customization-time*. What code query by example tries to achieve is to give a simple mean to express quantification,

using a language that does not work at the meta-level, or at least that does not look like a meta-language, see section 4.3.2.

Code representation The current version of our prototype works by performing code queries on the bytecode representation. An earlier prototype worked on a relational representation: the idea was to leverage existing relational database technology to implement code queries. We changed our design decision since performance was too low. It was partly due to the complex schema that is necessary to properly model assemblies, but also due to the performance of the database engine itself (SQL Server), which was far below expectations (despite proper indexing and the use of materialized views [91]).

A discipline of change The upgrade problem is important for customers, for partners, and for the base software makers. But is it of highest priority? Would partners, customers and base software makers be willing to surrender their powerful general purpose programming language for a restricted domain-specific language [77]; would they be willing to sacrifice time-to-market for well-thought out customizations that are in-line with planned evolutions of the base software product? Would base software makers be willing to commit to a planned evolution of their software products, say 5 years ahead? It seems that a complete solution to the upgrade problem would require all those actors to give up some of the flexibility they currently enjoy. Software engineering benefited from structured programming [53], which restricted programmers' freedom by imposing *a discipline of programming* [54]. Perhaps the same thing could be achieved with *structured evolution*: giving up a bit of freedom to get more security, by imposing *a discipline of change*.

1.10.2. Reflections on ERP systems

Toolchain While studying ERP systems, one of the early observation we made, is the contrast between the development toolchain around ERP systems and the one found in regular software development. A prime example is *continuous integration* [78]. Tools that facilitate continuous integration have been well explored, being promoted among others by the Agile software development community [78, 125]. More generally, the software engineering community has come up with a number of tools that speed up software delivery, that improve quality and ease maintainability. Consider, among others, build automation [10], unit test frameworks [151], but also modern *issue trackers* [103], modern *package management systems* and *deployment infrastructure* [12] as found in Linux environments, etc. ERP systems are typically lagging behind those well accepted, but relatively recent, tools. *Eventually*, of course, those tools are integrated into ERP systems. For example, Dynamics AX has announced a complete integration with SourceSafe [162], Microsoft version control system. Unfortunately, the delay between the adoption of those tools by the community at large and by the ERP makers is non-negligible.

Juggling frogs One could argue that the task of modernizing an ERP system is akin to juggling frogs: an ERP system typically has on the order of several million lines of code, and the user base counts in thousands world-wide (see section 2). It is true that a large user base can make evolution problematic: for example, the base classes of the .Net framework are notoriously difficult to evolve because any major change would break a large number of clients [47]. On the other hand, one should note that many tools can be adopted incrementally without breaking existing code, test frameworks for example [151]. Concerning modern tool support, we conjecture that partners that perform customizations and upgrades of ERP systems would benefit from a general purpose development environment such as Visual Studio [135] or Eclipse [64], since these versatile development environments are better fit to integrate external tools. Also, since those development environments are used by a larger number of developer, tools makers have more incentive to integrate their software. (Recall that Dynamics currently uses a *dedicated* development environment, see chapter 2.)

Back to Dynamics Typically, ERP systems are the result of many years of evolution, company merges, etc. (see section 2), which partly explain the reason behind those proprietary development environments. A devil's advocate may further argue that the *specific needs* of the ERP domain call for *specialized tools*. Microsoft undergoes some efforts to port their ERP systems to Visual Studio, a general purpose development environment [135]. The task is large, and a complete integration will take time. We think that integration is worth the effort, an open development environment would be an important asset. Note finally, and ironically, that once the integration with Visual studio has happened, Dynamics will have to keep itself up to date with the evolutions of the development environment: the upgrade problem once again.

On the absence of a large regression test We made another observation early, as we first learned about the development and business model of Dynamics: why Dynamics does not distribute a large *regression test suite* together with its products? A regression test suite would give some confidence to the partners that they did not break part of the ERP core functionality. As it is well known, the regression test case would *not* provide a proof that the modified system is behaving as expected [53] – but half a loaf is better than none – and the regression test could be made more or less extensive depending on the resources available, or the emphasis on quality. The main answer that we were given, by the Dynamics team, was that a regression test would commit the ERP maker to support the test suite. Another answer was that conceiving such a regression test would be a too large task for such a large system. We find these answers not convincing, since (a) the regression test can be produced and released incrementally to the market, and (b) the development teams of the ERP maker already use and maintain, internally, some kind of regression test – in which case releasing this test suite would mainly require some refactoring to make it fit for external developers. One should note that the test cases themselves would have to be customized because the product is customized. Furthermore, the customization of

test cases would have to be carried over to new versions of test cases when upgrading the product.

1.10.3. Reflections on enterprise systems research

Research on ERP systems The field of ERP systems is peculiar because it is dominated by big industry players, such as Oracle [155], SAP [177], Microsoft [136], etc. The software products are usually closed-source, and the signing of a non-disclosure agreement does not necessarily secure source access to critical parts of the system. Moreover, only well-known consulting companies [84, 9] can perform large empirical studies thanks to their close relationship with a large number of customers. Those consulting companies sell those reports – at a high price. Obviously, these facts hinder research.

The positive side There is a small number of online forums that gather ERP experts [140, 62]. It is through one of these forums that we could find experienced practioners willing to answer our online survey [62], see chapter 3. Note that a simple post to this forum would have been most likely ignored. In our case, we were lucky to meet at a Dynamics conference an influential member of this community, the founder of the site, that backed our request. Without his support, it is likely that our efforts would have been for naught. Hence, one of the lesson learned is that within closed circles, like the ERP systems community, it is important to get support from a recognized leader.

The open source wave Big industry players, like Microsoft, are still oriented towards a closed source model. Nonetheless, one has to acknowledge that there is a trend to open source more and more material. For example, the MEF framework [127] that we used in our implementation (see section 1.7), was released under an open source license. Finally, there are a number of small open source projects that implement ERP-like functionality, for example Ofbiz [153]. These projects can be used to perform research on enterprise systems.

Empirical research This work takes place under the umbrella of a larger project on *Evolvable Software Products* [71]. Part of the research agenda, that was already established when we joined, was to perform qualitative empirical research (see chapter 3). Although the study allowed us to uncover some of the characteristics of the ERP field (such as the fact that correctness is typically not a priority for many partners), we missed important measurements, quantitative data, on which we could have built our work. For example, we did not benefit from statistical data on customizations hot-spots in the source code. Based on our experience, our recommendation is to focus on a *quantitative* approach for related projects, and to use a *qualitative* study to complement it. In the context of the same research project, Rhiger has recently undertaken quantitative analysis of customizations on Dynamics NAV code using

tree alignment [102]. The study presents interesting preliminary results [169], such as the fact that a non-trivial number of customization points are used by many Global Development Localization teams (teams within Microsoft that perform country specific customizations). From a more general point of view, we refer the reader to d'Ambros et al. [49], for an example of tool support for software repository analysis, and to Fernández-Ramil et al. [72] for a summary of quantitative empirical studies of open source projects.

1.11. Related work

Organizational challenges Unphon studies the re-engineering of software products to allow for better evolvability [199]. This study suggests that the existing organization of a company should be carefully taken into account before embarking on a large scale re-engineering project. In our work, we focused mostly on the technical perspective – we barely touched upon the human and social aspects, mainly in our empirical study (see chapter 3). Hence, we find that the work by Unphon is quite complementary to ours.

The REA Model The REA model for accounting, an important functionality of ERP systems, proposes to improve the traditional approach based on double entry bookkeeping. For example, it can express exchanges where money is not involved (barter trade) [98]. It would be interesting to evaluate the upgrade problem in the context of the REA model, and study how alternative ERP models impact upgradability. More generally, we worked on an upgrade concept that may be adopted incrementally, on top of the existing model, whereas new models typically require a complete and profound refactoring of existing software.

Outsourcing Looking at the upgrade problem from a business perspective, one may suggest to simply outsource some of the upgrade activities. This solution does not solve the technical problem *per se*, but it might indeed reduce the upgrade cost. Existing research on this topic focus, among other things, on *knowledge management*: the challenge is that (a) the outsourcing resources must have the required knowledge to successfully perform the upgrade activities and that (b) the knowledge they create during the upgrade is stored and made available to the company [70]. The problem here is very similar to the problem of employees leaving partners' companies that we touch upon in chapter 3. There is evidence that the total life cycle cost is difficult to predict when outsourcing is used [154].

Code query by example Code query by example has been used extensively in the database community [184]. A few related projects have explored some variants of code query by example with respect to a general purpose programming language, for example the work by Cohen et al. on JTL [44]. One of the main difference with our framework is that they do not use an embedded language: their language is a variant of Java, and hence cannot be compiled with a standard compiler. Similarly, De Roover et al. use Java code patterns embedded in a logic query language [173]. Finally, Martin et al. use a dedicated query language, called *Program Query Language*, to embed code patterns [124]. Again, contrary to our approach their host language is a dedicated language, and not a general purpose language such as C#. In the Aspect-oriented programming (AOP) community, some variants of CQE have been proposed; the work by Noguerra et al. came recently to our attention; building on the Spoon framework [186], they propose to use source code templates to denote static

pointcuts [149]; their proposal is, to the best of our knowledge, the closest to CQE, since their pointcut language is embedded in pure Java. Their implementation works at the abstract syntax tree level, whereas we perform matching and instrumentation at the bytecode level.

The fragile pointcut problem Still in the AOP community, a number of researchers have worked on the *fragile pointcut problem*: upon evolution of the base program, existing pointcuts might not denote the proper join points. The fragile pointcut problem is closely related to the upgrade problem; as illustrated in chapter 5, our work can be framed in the context of AOP. Stoerzer et al. propose to analyze changes in matching behavior upon evolution, and to produce a so-called *pointcut delta* [187]. The idea is similar to what we described in section 1.6: upon evolution, when new join points appear, developers can browse through the relatively small list of join points (what we called positives), and check whether the join point is correct or not. Similarly, Kellens et al. face the fragile pointcut problem using *intensional views* [106] (sets of programs entities that share some properties) : pointcuts are defined in terms of views, not directly using the concrete program entities, and therefore some join points can be ported from one version of the base software product to the next .

Dependency injection and Inversion of Control Containers *Dependency injection*, and its closely related subject *Inversion of Control Containers* (IOCCs), are very active topics in the industry, see for example the Castle project and their Windsor container [37]. In the words of the Castle project: "[Inversion of control] is the opposite of using an API, where the developer's code makes the invocations to the API code. Hence, frameworks invert the control: it is not the developer code that is in charge, instead the framework makes the calls based on some stimulus". Similarly to our work IIOCCs often rely on injection of dependencies. There is no clear-cut separation between what is AOP and what is dependency injection: one can be used to implement the other. Nonetheless, the emphasis with IIOCCs is more on complex configuration capabilities [165]; on the other hand, using IIOCCs, obliviousness is typically not strictly respected: anticipation is often required in the form of a predefined interface and explicit calls to a container.

.Net extensibility frameworks There are four *extensibility frameworks* currently supported by Microsoft: Managed Extensibility Framework [127] (MEF), on which we build (see section 1.7); Managed Add-In Framework [122] (MAF, also called by its namespace `System.AddIn`); the Prism project [167] (also called Composite Client Application Guidance); the Unity project [198]. Those projects overlap, which currently create confusion in the .Net community. They are also closely related to *dependency injection* that we just mentioned. MEF focuses on simplicity of the programming model using attributes; MAF concentrates on isolation of extensions using `AppDomains` [135]; Unity is presented as a "lightweight extensible dependency injection container" [167]; Prism is dedicated to graphical applications. The main difference with our

work is that, using those frameworks, the extensibility points are typically fixed: their location is hard-coded in the base code; CQE offers quantification which enable to port *some* customizations to the next version of the base software product.

Pattern matching and unification Programming language support for *pattern matching* has been studied extensively by the functional language community, see for instance Sestoft [181] for a study of efficient implementation of pattern matching. Our matching requirements can be related to what the functional community refers to as non-linear patterns: patterns where the same variable can appear multiple times. Pattern matching itself can be seen as a special case of unification where there is no free variable in the target: unification is *two-way matching* [60].

Software product lines We study software products lines (SPLs) using AHEAD and Multidimensional separation of concerns in chapter 2. The goals behind SPLs bears some similarities with our work: both allow for the customization of software systems by third-parties. Nonetheless, there is one fundamental difference: SPLs have a closed-world assumptions. That is, the SPL community usually assumes that the set of possible customizations is well-known in advance, by a central agent such as chief architect, see chapter 2. In other words, the closed-world assumption provides a very precise anticipation of all possible customizations. ERP systems cannot rely on this assumption, see section 1.4.3. From this perspective ERP systems are not SPLs.

Open classes Existing work on *open classes* shares some objectives with our work, namely to avoid dependence on anticipation, and to be able to add (and sometimes change) behavior of existing classes – without creating distinct subclasses or editing existing code. MultiJava allows a programmer to add new methods to existing classes [42]. Similarly, in Ruby, the implementation of a class is not closed, methods can be added and redefined [174]. Open classes are similar to the system of layers of Dynamics AX, described in section 2.3.9.

Code generation for embedded systems Code generation is a well-known approach to variability [48], we will give a specific example. Wąsowski studies software product lines in the context of embedded systems, using *statecharts* that characterize *reactive synchronous* systems [211]. A code generator, called SCOPE, is proposed to generate C code from those models. Wąsowski uses the notion of *color-blind environments* (the inability of some environment to distinguish certain responses to the system), to parametrize the output of the tool [211]. Obviously, the domain of embedded systems is different from the domain of ERP systems: the embedded systems community typically emphasize resource usage and correctness, whereas ERP systems emphasize ease of use and time to market. Furthermore, Wąsowski starts from an abstract model and goes toward a concrete program, whereas we work directly on a concrete program.

Updatable views The database systems community faces a problem which can be related to our upgrade problem: the view update problem. This problem is concerned with how modifications to a database view should be reflected in the database. Bohannon et al. propose a language to deal with this problem, based on the concept of relational lenses, where a lense is defined as a bi-directional transformation [29]. Since the database system is an important component of ERP systems, it would be interesting to study further how their approach could be used in the concrete setting of an industrial enterprise system.

Related surveys and taxonomies To the best of our knowledge, there is no technical survey on the upgrade problem that spans a large number of ERP systems. Our survey, see chapter 2, is confined to Dynamics AX and Dynamics NAV. Nonetheless, there exists a number of surveys and taxonomies on related fields. We refer the reader to Mens et al. for a taxonomy of software evolution [131], and to Mens and Demeyer for a recent collection of papers on software evolution [132]. Czarnecki and Eisenecker describe a wide range of generative techniques for software variability [48]. Conradi et al. give an introduction to software configuration management systems [46]. State-of-the-art in behavioral specification is summarized by Hatcliff et al. [92]. Filman et al. give an overview of AOP [74]. Finally, software reuse is surveyed by Krueger [116].

1.12. Conclusions

Back to the beginning The problem statement (section 1.1.2) asked what were the characteristics of the upgrade problem within the field of enterprise systems. It then inquired how to improve existing customization techniques with respect to the upgrade problem. The hypothesis (section 1.1.4) claimed that the characteristics of the upgrade problem within the field of enterprise systems were that (a) little anticipation is possible with respect to future customizations (b) any solution to the upgrade problem should be simple enough for domain experts (c) adoption of a new customization technique should be incremental. Then, the hypothesis proposed the concept of *code query by example* (CQE) to complement existing customization techniques given those characteristics.

Claims concerning the characteristics of the domain are supported by our empirical study, see chapters 2 and 3; the claim concerning the simplicity of CQE is grounded in queries being specified *by example*, see section 1.5, and supported by comparisons with other code query tools, see chapter 6; the claims concerning the portability of a subset of customizations with CQE, of our non-reliance on anticipation, and of possible incremental adoption are grounded in the quantification and obliviousness characteristics of our approach (see the definition of those terms in section 1.3.3), detailed in sections 1.5 and 1.6, and chapters 5. Table 1.5 summarizes the claims and the related argumentation.

Table 1.5.: CLAIMS AND ARGUMENTATION SUMMARY

Claims	*Argumentation*	*Chapters or sections*
Characteristics of the domain	Empirical study	Chapters 2 and 3
Simplicity of CQE	Code queries given by example; comparisons with other code query tools	Section 1.5, chapter 6
CQE requires little anticipation	Code quantification and obliviousness	Section 1.5, chapter 4
Portability of some customizations using CQE	Code quantification and obliviousness	Section 1.6
CQE allows for incremental adoption	Code quantification and obliviousness	Section 1.5, chapter 5

Table 1.6.: SUMMARY OF CUSTOMIZATION TECHNOLOGIES EVALUATED

This table references the 20 software customization techniques discussed in this dissertation. Each particular "technique" is bound to a particular implementation (for example, there is a wide range of variants among version control systems). Furthermore, the techniques are not mutually exclusive, and can sometimes be considered a variant of one another (for example, code query by example can be seen as a variant of aspect-oriented programming, see chapter 5; covariance is related to traditional inheritance mechanism, etc.). Software engineering is concerned with reaching compromises and do not claim purity nor tries to attain a grand unifying theory that some computing sciences aim for [96].

Technique	*Section*	*Implementation*	*References*
Aspects	2.5.7	Yiihaw	[104, 105]
Assembly versioning	1.9.4	.Net	[135]
AX Layers	2.3.8	Dynamics AX	[88]
Code query by example	1.5	Eggther	-
Covariance (formal parameters)	1.9.2	-	[36]
Events	2.5.4	C#	[65]
Factory pattern	1.9.5	C#	[125]
Information hiding	2.5.2	C#	[159, 65]
Inheritance	2.5.1	C#	[65]
Mixins, Traits	2.5.6	Scala	[61, 152]
Parametric Polymorphism	2.5.3	C#	[65]
Partial methods	2.5.5	C#	[65]
Procedural abstraction	1.9.1	C#	[3]
Service locator	1.9.5	C#	[79]
SPL using AHEAD	2.5.8	AHEAD	[20]
SPL using MSC	2.5.9	Hyper/J	[157]
Strategy pattern	1.9.5	C#	[125]
Template method	1.9.5	C#	[125]
Version Control Systems	1.9.3	Subversion	[45]
Visitor pattern	1.9.5	C#	[125]

What was accomplished

- We characterized the upgrade problem, and we showed that the problem is not fully addressed by existing customization techniques (see chapter 2). Furthermore, we conducted an empirical study to ground our work (see chapter 3 and section 1.2.6).
- We introduced the concept of *code query by example* (see section 1.5), and we showed how customizations can be performed using this approach; we described the implementation of a prototype, and presented some design alternatives (see section 1.7).

- We studied how code query by example can help in some cases during upgrade (see section 1.6), and illustrated that it is not a perfect solution.
- We studied the limitation of code query by example in the context of enterprise systems, and of the prototype (see section 1.8).
- We studied other customization techniques in the context of the upgrade problem (see chapter 2 and section 1.9).
- We proposed to apply code query by example to lightweight static analysis, and to aspect-oriented programming (see chapter 6 and chapter 5).
- We reflected on lessons learned during our research and discussed related topics (see section 1.10)
- Finally, we discussed related work (see section 1.11).

All things considered The upgrade problem is challenging. The core of our contribution was to propose a customization technique, that complements existing approaches. What we consider the main benefits of our approach are its simplicity, its non-reliance on anticipation, its capacity to be incrementally adopted, and thanks to our embedded language, its leveraging of existing development tools. What we consider the main disadvantages of our approach are the weak expressiveness of the query language, its absence of support for runtime instrumentation, and its absence of support for fine-grained customization.

Further work Our approach can be improved; we see a lot of potential for further research on code query by example. One could look at ways to improve the expressiveness, while trying to retain the concept of code query by example. For example, code matching could be done at *the class level* and not just at the method level. Regarding the implementation of code matching itself, we see many opportunities for further innovation, for example trying to exploit F# (a functional language for .Net) and its *meta-programming* capabilities [193] to implement code matching. One could also try to leverage recent work on *dynamic languages* [148] to provide a more flexible approach to customization. Furthermore, one could exploit the fact that queries are valid programs to attempt behavioral matching, by *executing* queries instead of just analyzing them statically. Finally, from a more general point of view, we think that further research is needed to understand how all the various customization techniques that we introduced *interact* (table 1.6 summarizes those techniques). This last point is particularly challenging – but also particularly inspiring.

Part II.

Collection of Papers

2

Technologies for evolvable software products: The conflict between customizations and evolution

Peter Sestoft
IT University of Copenhagen
sestoft@itu.dk

Sebastien Vaucouleur
IT University of Copenhagen
vaucouleur@itu.dk

Published in: Advances in Software Engineering, Lecture Notes in Computer Science vol. 5316. Springer-Verlag 2008 [182].

Abstract

A *software product* is software that is built for nobody in particular but is sold multiple times. A software product is typically highly customizable, or adaptable, to particular use contexts; moreover, such a software product can typically be thought of as a common *kernel* plus a number of *customizations*, one for each use context. A successful software product will be used for many years, and hence the kernel must evolve to accommodate changing demands and environments. The subject of this paper is the conflict between the *customizations* made for each use context and the *evolution* of the kernel over time. As a case study we consider Microsoft Dynamics AX and Dynamics NAV, highly customizable enterprise resource planning (ERP) software systems, for which upgrades are traditionally costly. We study the challenges related to the customization/evolution conflict and present some software engineering approaches, programming language constructs and software tools that attempt to address these problems, and discuss whether they could be brought to bear on the conflict.

2.1. Introduction and definitions

A successful software product is typically released in many versions over many years; it *evolves* over time. Also, a software product is typically *customized* to permit

effective use in many different applications and contexts. In this work, we are interested in the problems and conflicts that arise from the combination of evolution and customization; we call this the *upgrade problem*.

In this section, we define the most important terms used in the next sections. Then we discuss the relation to the concept of a software product line as it is currently used in the literature. Finally, we outline the contributions and the structure of the paper.

Definitions A *software product* is software that can be used in many different contexts, such as a shared calendar system for organizations, or a text processing system. Such software products should be contrasted with software that has been developed in a project for a particular purpose, for instance for the securities trading desk of a particular bank. One may view a particular instance of a software product, deployed in a particular context or organization, as consisting of a software *kernel*, plus *customizations* (adaptations of the kernel to the context), plus possibly further *configurations*, whether organization-wide or for the individual end-user. In this paper we shall distinguish *customization*, which can add new and possibly unforeseen features to software, from *configuration*, which enables or disables features that are already present in the software, change their behavior, or affect the way they appear to the end-user[1]. *Software evolution* is the phenomenon that software must change over time to stay useful: errors must be fixed, security holes must be plugged, new functionality must be supported, and changes in the environment must be accommodated [131]. Finally, software composition is the construction of software applications from existing software parts.

Software product lines The software products considered in this paper are clearly related to software product lines [185]. Software product lines typically have a closed-world assumption in which a central agent (such as a chief engineer) has a clear idea of all the variations that are required. We will consider this approach to customization and compare it with approaches that have an an open-world assumption – that is, where no central agent has a clear understanding of all the possible customizations that may be needed for a software product.

For software product lines with a closed-world assumption, the construction of a particular member of the product line comes down to choosing from a predefined set of features, that is, to configuration. In that case, kernel evolution and customization are hard to distinguish.

Contributions The contribution of this paper is two-fold. First, it gives a more precise characterization of what we call the upgrade problem. Domain experts and practitioners have previously claimed that the upgrade problem is an important one, but surprisingly it has never been thoroughly studied from a technical angle to

[1]Note that this terminology is not universally accepted. For instance, the Microsoft Excel 2003 menu `Tools > Customize` performs what we would call configuration: it determines which toolbars to display in the user interface, and so on.

the best of our knowledge. Second, we give a subjective evaluation of some of the most commonly used customization techniques and study how they can be used to mitigate the upgrade problem. We support our conclusions by using an explicit set of criteria, as well as a simple running example.

Road-map The next section gives a detailed explanation of the upgrade problem. Then, section 2.3 gives a concrete example of this problem, through a study of two widely used ERP systems. Section 2.4 gives a list of criteria that will be used in section 2.5, the core of the paper, to give a subjective evaluation of some of the most widely used customization technologies.

2.2. The upgrade problem

The focus of this work is the interaction of two distinct dimensions of change, namely *customization* and *evolution*. When the kernel of a software product evolves, and an organization wants to upgrade to a new version of this kernel, the customizations of the deployed software product must be carried over to the new version. In simple cases this may just involve copying the customizations over unchanged, but in general it may involve a rewrite of the customizations and also require comprehensive knowledge of the customizations as well as the old kernel and the new kernel; see section 2.3.2. This work incurs considerable cost and often causes end-user companies to postpone the upgrade as long as possible.

2.2.1. Customizable software

Almost all software is configurable. Even the most mundane of applications, the Minesweeper game delivered with Microsoft Windows, has several levels of difficulty, sounds on or off, and so on. Also, there is hardly a Unix (or Linux) program without a configuration file somewhere in /etc/, or a .foorc configuration file in the user's home directory.

Moreover, much software is customizable, in the sense that it admits subsequent extensions of its functionality, unforeseen at the time the software itself was designed, implemented and shipped. For instance, Web browsers such as Firefox support add-ons that enable the browser to display new media types; spreadsheet programs such as Microsoft Excel support add-ins that enable the spreadsheet program to solve optimization problems and other specialized tasks; and integrated development environments such as Eclipse support plug-ins that enable the development environment to support new programming languages, graphical modeling tools, and so on. These add-ons, add-ins and plug-ins are what we call customizations.

In all the above examples the additional functionality is provided via software components that can be dynamically loaded into a running application on demand. A more static approach would permit customization when the software is built, by importing (or not) third party features into the software when it is compiled or linked.

Operating systems such as Linux support both static and dynamic customization: drivers for particular network devices, file systems, and so on can be added to the Linux kernel when compiling it, and in addition some modules (e.g., wireless network drivers, support for USB devices) can be loaded and unloaded on demand while the operating system is running.

In the above examples, there are well-specified interfaces between the kernel and the software (add-ins, add-ons, plug-ins, modules) that implement the additional functionality. As shown by the case study in section 2.3, for some software systems it is difficult or impossible to foresee what kinds of customizations are needed, so it is impossible to design interfaces that are both general enough and specific enough. Instead, (some) customizations require edits to the kernel software itself. We shall call the latter a white-box approach to customization of the kernel.

2.2.2. Software evolution

All software will change from time to time, if it is used at all, as evidenced by the all too familiar and increasingly arcane version numbers: C# compiler version 3.5.21022.8, Eclipse version 3.3.1.1, Oracle database version 9.2.0.6.0, Linux kernel 2.6.23.1-42.fc8, and so on.

Software evolution is a research topic in its own right, pioneered by Lehman in an empirical research setting three decades ago [128, 129], and now having its own conferences, terminology and methods, as well as a Journal of Software Maintenance and Evolution. Lehman's original software evolution research made several observations: We all too often believe that the system we are currently building will be the final one, and hence we fail to plan for change, whether foreseeable or unforeseeable. Also, the very purpose of some kinds of software systems, called E-programs by Lehman [128], is to cause a change in the context in which they are deployed, and hence those are even more prone to change, as the context changes and feeds back change requirements on the software system. Current research on software evolution and maintenance attempts to classify the various reasons for evolution [131], to propose theoretical means to understand software evolution and to find practical mechanisms to help maintain software during evolution.

Probably the strongest drivers of software evolution are:

- commercial pressure to support additional functionality
- organizational changes, such as company mergers
- legal changes, such as additional audit requirements
- changing technical environments, such as evolving operating systems
- demand for distributed and mobile access and new user interface technology
- co-evolution for interoperability with other software

2.2.3. The evolution of specifications

The problem of supporting customization as well as evolution cannot be addressed without taking evolving specifications into account. In an ideal software architecture, every software kernel component is accessed only through a well-defined specified interface. If a customization modifies a component, but the component continues to satisfy the specified interface, then obviously the software system continues to work correctly, using the traditional relative notion of correctness (satisfaction of a specification).

However, the point of software evolution is often that the specification must change, not just the implementation. Changes to the specification are usually caused by changes in the environment, such as new business processes or user needs, as outlined in section 2.2.2. When the specification changes, the black-boxing of the implementation behind the specification provides little help in the upgrade of customizations.

The more interesting and challenging case is when the software kernel evolves due to a changing specification, not the case where its implementation changes but the interface specification remains the same.

2.2.4. Upgrade problems in operating systems

In early versions of Microsoft Windows, upgrade problems would be experienced almost daily, a phenomenon that was known under the name "DLL hell". Most applications would rely on dynamically loaded libraries (DLLs), which were typically shared system-wide between multiple applications. This caused problems because at any time there could be only one installed version of each DLL, and newer versions of a DLL were not necessarily backward compatible. For instance, installing a new version of the Internet Explorer web browser might require an upgrade also of a DLL, and the deletion of the old version. Subsequently one would discover that the accounting software installed on the same computer had relied on that old version and was incompatible with the new version, and hence stopped working. At its core, the problem was that multiple applications relied on a common resource (DLL), and that one application would affect the others through unwanted modification of the common resource. Another variant of this problem would be that manipulation of the path environment variable caused by installation or upgrade of one application would mean that other applications could no longer locate their DLLs and therefore stopped working.

The same problems could be observed in early Linux distributions, where an upgrade of the gcc C compiler and its associated libraries might break some other part of the system. In more recent versions of Microsoft Windows as well as Linux, such problems are addressed by allowing multiple versions of the same library to coexist. For instance, in a current Linux installation one may find both versions 0.9.7a and 0.9.7f of the libssl.so library.

Modern programming platforms, such as Microsoft's .NET, address these problems in an even more powerful way, by allowing one library (called an assembly) to

express its versioned dependencies on other libraries. A forthcoming version of the Java platform is expected to support versioning of libraries (called modules) and versioned dependencies in a similar way [89]. In the experimental language Fortress being developed at Sun Microsystems for DARPA, the basic program module is the trait (see section 2.5.6), and the language aims to provide upgradable program components in the form of versioned collections of traits [6, 7].

2.2.5. Conclusion on the upgrade problem

The upgrade problem is found in many contexts and can be addressed in many ways. In the remainder of this paper we focus on software products, and in particular on the conflict between customization of a software kernel and subsequent evolution of that kernel. In particular, we consider this problem in relation to highly customizable enterprise software systems.

2.3. Case study: Dynamics AX and NAV

To get a more concrete setting for discussing upgrade problems, we now present Microsoft Dynamics AX [137] and Dynamics NAV [138, 144], two enterprise resource planning (ERP) systems from Microsoft Corporation.

For short, the term "Dynamics" will refer to the Dynamics products (AX and/or NAV), and the term "Dynamics developers" will refer to the core Dynamics development teams at Microsoft.

2.3.1. Add-ons and configurations

Both Dynamics AX and Dynamics NAV are highly customizable and configurable, and customization takes place in several stages. Microsoft builds and sells a kernel system, consisting of runtime environment, database system, development environment and a number of core packages, e.g., for sales tax reporting in a particular country. A large number of partners, also called independent solution vendors (ISVs) or value-added resellers (VARs), sell add-on solutions and customizations.

An add-on solution may be targeted to a particular industry (a vertical solution area), such as apparel and textiles, or address a particular activity within an organization (a horizontal solution area), such as customer relationship management. Several add-on solutions may be used together in a Dynamics installation. Simply put, in ERP parlance an add-on is a set of customizations. Further customizations may be created on top of the kernel and the add-ons, thus tailoring the ERP system to the needs and processes of a particular company. Some end-user companies even make such customizations themselves.

Add-ons are written as additional modules or by modifying parts of the kernel modules, using the development environments. Hence, Dynamics AX and NAV are software products developed over a long time and sold in many copies, with a

wide range of customizations, to many different customers. They also exhibit the upgrade problem outlined in section 2.2 above, in a particular way: The add-ons and customizations are developed primarily by partner companies, whereas the kernel evolution is controlled primarily by the Dynamics development team.

This section will provide details about the Dynamics software products, the upgrade problems experienced, and some current practices to alleviate them.

2.3.2. Dynamics NAV versus Dynamics AX

Before we dive into the upgrade problems in more detail, let us consider the characteristics of the Dynamics NAV and Dynamics AX enterprise resource planning systems. Both systems are partially model-driven and partially programming language based. Namely, database tables, runtime data structures, and the user interface (forms) are described by meta-data, not built using programming language declarations. On the other hand, behavior is described using traditional programming language constructs, called code units, which correspond to functions or methods.

The two systems have distinct organizational and technical characteristics:

- Dynamics NAV mostly targets smaller organizations, for which pre-developed add-ons mostly suffice, so they only require minor customizations. A large number of organizations run Dynamics NAV. The integrated development environment is called C/SIDE, and the programming language, C/AL, is a relatively simple language with a Pascal-like syntax. The developers employed by NAV partners usually focus on the customer's business and many do not have a strong background in software development. Code unit customizations are made simply by editing the required code units in the C/AL language.

- Dynamics AX mostly targets larger and more complex organizations, that often require extensive customizations. Fewer organizations use Dynamics AX than NAV. The integrated development environment is called MorphX, and its proprietary programming language X++ is an object-oriented language with a Java-like syntax. The developers employed by AX partners often have a good background in software development. The Dynamics AX model is structured into a number of layers, with layers for the kernel, layers for partners' customizations, layers for further customizations in the end-user organization, and so on; see section 2.3.8. A code unit customization is made by copying the code unit from the layer at which it was originally defined and then adding and editing at a higher layer. The higher layer version will then be used instead, and is said to shadow the lower layer code unit; see 2.3.9.

We present both systems here, because their different organizational and technical characteristics cause different kinds of upgrade problems.

2.3.3. The Dynamics developer ecosystem

Microsoft and its partner companies form an ecosystem in which the partners depend on Dynamics developers for providing a kernel that is robust, comprehensive, easily customizable, and up to date. Conversely, Microsoft depends on the partners for marketing its kernel, for developing add-ons that make it valuable to customers, for making customizations, and for deploying the customized solutions in customer organizations.

There is a delicate balance in relation to the evolution of the system kernel: If the kernel changes by frequent small steps, then the partners will find it difficult to sell all these upgrades (of kernel and customizations) to their customers; but if the kernel changes by infrequent radical steps, partners or customers may find upgrade so complex that they can just as well switch to a competing product (such as SAP, an Oracle-based system, or software as a service). Also, if the kernel evolves too slowly or not at all, advanced customers may find that it no longer inter-operates well with other software they use, or does not support new reporting standards or functionality that they need, such as visualization, business intelligence, electronic trade, etc.

2.3.4. What constitutes an upgrade

Common to Dynamics AX and NAV is that an upgrade to an installation involves upgrade of kernel and customizations as well as conversion of the end-user organization's production data. The data conversion poses interesting challenges itself. First, it is highly time-critical because the end-user company usually cannot conduct business while the data conversion is being done, so the conversion must take place over a weekend or an extended weekend. Second, the data conversion must be fully reliable, or it would disrupt the business. Third, full-scale testing of the scripts that perform the data conversion cannot be conducted until a test environment consisting of the entire upgraded ERP system (new kernel and upgraded customizations) is available, which is usually late in the process; see also section 2.3.9. The code and meta-data migration can be done in advance of the actual data conversion upgrade; only the data conversion is time-critical in this sense.

Nevertheless, we shall say no more about the data conversion process in this paper, but focus on the problems caused by upgrade of code customizations.

2.3.5. Upgrade problems in Dynamics NAV and Dynamic AX

It is clear from a survey of partners [58], from talking to the Dynamics AX and NAV core development teams, and from various online forums and blogs, that upgrade of customizations in Dynamic AX and NAV are problematic. For instance, a public video from a Dynamics AX core developer [163] acknowledges that upgrade of customizations can be costly: "Our research shows that an average upgrade costs as much as 30% of (the original cost of) the customizations". As further evidence, a Google search for "dynamics nav upgrade" gives 114,000 hits (January 2008). There

are companies, such as Liberty Grove Software in Illinois, USA, that specialize in doing NAV upgrades for other partners at a fixed price quoted after a preliminary upgrade diagnostic. Also, partner-oriented materials from Microsoft itself suggest that care is needed when customizing the systems to minimize future upgrade problems (see section 2.3.10).

2.3.6. Constraints on a solution to the Dynamics upgrade problem

Although a kernel upgrade affects both add-on solutions and partner-made customizations (see section 2.3.1), in this paper we focus on the problems caused by partner-made customizations, because fewer resources are available for upgrading those than for upgrading add-ons, which are usually sold more than once.

A potential solution to the upgrade problem should work with the current ecosystem (see section 2.3.3), and should provide a plausible upgrade path from the technologies currently used (the existing code base is very large, therefore incremental technology adoption is important). Ideally the solution, especially for NAV, should support the short edit-compile-run cycle that developers are used to. Developers add, modify and experiment with customizations in the development environment, and then immediately switch back to the running enterprise application without a lengthy build phase and without restarting the enterprise application and loading data anew. Finally, the solution should support some form of static checking: there should be tool support to discover which customizations may be affected by changes from the old version of the kernel to the new version.

2.3.7. Handling upgrade in Dynamics NAV

Here we consider how the modest size and complexity of some NAV customizations mean that the upgrade of customizations can be handled by rather simple techniques. A particular Dynamics NAV partner, Logos Consult in Denmark, reports [144] that most of their original customization projects are small, on the order of 50–500 man hours, and involve only one or two developers. While doing the original customization, developers simply mark each change in the customized code using stylized change comments with date and developer's initials, like this:

```
// >> 07.FM
DtldCVLedgEntryBuf."Document Date" := "Document Date";
DtldCVLedgEntryBuf."Job No." := "Job No.";
// <<
```

These stylized comments are easy to search for in the source base, and indicate who made the change and when. Because customization projects are so small, and because developers stay long with Logos Consult, this information is enough for the developer to understand how to upgrade the customization when subsequently the kernel gets upgraded; no special tools are used to assist in the upgrade. Program comments might also be used to indicate why the change is made, but often this is not needed.

Table 2.1.: THE LAYERS OF A DYNAMICS AX APPLICATION

Layer name	*Meaning and purpose*
USR	User: Individual companies, or companies within an enterprise, can use this layer to make customizations unique to customer installations.
CUS	Customer: Companies and business partners can modify their installations and add the generic company-specific modifications to this layer. The layer is included to support in-house development without jeopardizing modifications made by the business partner.
VAR	Value-added reseller: Business partners use this layer, which has no business restrictions, to add any development done for their customers.
BUS	Business solution: Business partners develop and distribute vertical and horizontal solutions to other partners and customers. A vertical solution targets a particular line of business such as brake pad manufacturing. A horizontal solution addresses a particular task that is similar across multiple businesses, such as car fleet management.
LOC	Local solution: For strategic local solutions developed in-house.
DIS	Distributor: For critical hot-fixes.
GLS	Global solution: For country-specific functionality.
SYS[a]	System: The lowest application element layer and the location of the standard Dynamics AX application.

[a]The LOS, DIS and GLS layers are developed by the Dynamics development team but their application elements can be customized by partners. Only Dynamics developers have access to the element definitions at the SYS layer.

The Dynamics NAV approach sketched above is simple and suffices for NAV applications that do no differ too radically from the NAV kernel, but it is unlikely to scale to applications that require extensive customizations.

In the rest of this paper we will focus on Dynamics AX, for which customizations are usually much more elaborate.

2.3.8. The layered structure of a Dynamics AX application

The Dynamics AX layering system supports multi-stage customization and extension. The architecture has eight layers [88, page 50], shown in table 2.1. An application element (also called model element) at a higher layer hides one with the same name on lower layers. This supports multi-stage customization because a lower-layer application element may be customized at a higher layer, and that customized application element may be further customized at a yet higher layer.

For each of the eight layers shown in table 2.1 there is a patch layer directly above it, used for small delta updates, for instance to avoid redistributing a slightly changed version of the entire 472 MB SYS layer file.

2.3.9. Customization using AX layers

To customize or extend an application element from a lower level (say SYS) at a higher level (say LOS), the developer copies the entire application element to the

LOS level and makes the desired edits to it there. Henceforth the system will use that customized application element. A subsequent upgrade to the application element at the SYS level is not automatically carried through, but must be handled manually in an upgrade project.

In response to a subsequent kernel upgrade, at least the following tasks must be performed:

- Find all those lower layer elements that have changed in the new kernel version and have been customized in the current installation.
- In each case, decide whether
  - (a) the new lower layer functionality makes the customization unnecessary; if so, remove it
  - (b) the customization continues to work; if so, copy it to a new customization of the lower layer code
  - (c) the customization no longer works; if so, design and implement a new one

These steps require insight into both the old and the new version of the Dynamics AX kernel, into the old customizations, and into the reason for making those customizations in the first place. Hence this work must be done by an expert, preferably the same developer who made the old customizations.

A shadow is an application element from the standard application that has been modified at a higher level. The cost of an upgrade (of the standard application, say from AX 3.0 to 4.0) is to a high degree determined by the number of shadows [88, pages 464-467].

A partner-oriented textbook on Dynamics AX distinguishes the various environments in which a version of the system may execute [88, pages 466]: production environment, test environment and development environment. It also distinguishes the following phases of the upgrade process, from Dynamics AX 3.0 to AX 4.0, say:

1. Test AX 3.0 layer files (customizations) in test environment
2. Create a production environment with AX 3.0 and the layer files
3. Modify layer files to work in AX 4.0; [that is, upgrade the customizations]
4. Write data migration code and migrate data from AX 3.0 production environment to AX 4.0 development environment
5. Perform functional test of the AX 4.0 application with migrated data
6. Move AX 4.0 layer files to production environment and migrate up-to-date AX 3.0 data files; this is the time-critical step mentioned in section 2.3.4
7. Start production on the AX 4.0 application

2.3.10. Mitigating code upgrade problems in Dynamics AX

A public video called "Smart Updates" from a Dynamics AX core developer [163] gives some advice on upgrade in Dynamics AX. Its main messages are:

- One should customize small application elements such as class methods, and avoid big ones such as forms: "Once you customize an application element, a copy of the entire original element is placed in the customization layer". The larger application elements one customizes, the more future upgrade liabilities are incurred.
- One should avoid gratuitous customization: "It is tempting to customize everything" but then later the "customer upgrades the kernel application" and "you'll have to resolve all conflicts" that is, "whenever you're overlayering an element that has changed"
- One should avoid, whenever possible, code unit customizations that could cause a conflict at a later upgrade. Instead one should use "class substitution".

"Class substitution" simply exploits that the Dynamics AX language has object-oriented features, unlike the Dynamics NAV language. The idea is to (1) make a derived class of the to-be-customized lower layer base class, overriding the method that should be customized; (2) to introduce a factory method, for instance called "Construct()" that returns an object of the derived class instead of the base class object; and (3) to make sure this Construct method is called everywhere the base class constructor would otherwise be used. Section 2.5.1 below further explores this approach to customization, which is a classic object-oriented idea. The point is that a customization based on "class substitution" is much easier to upgrade than a customization that consists of arbitrary edits to the source code of a code unit.

2.3.11. Another case study

As another case study we considered an advanced collection library for C# and .NET, and whether one could build a generator of specialized libraries while preserving maintainability of the base library. Writing such generators is an old dream, first made explicit by McIlroy in 1969 [126]. An approach using static aspects in explored in a companion paper [104]. That case study has very different properties. In particular, it works with closed-world assumption, where the universe of versions is known from the outset: singly-linked versus doubly-linked lists; constant-time or linear-time size property; update event listeners or not; fail-early enumerators or not; slidable list views or not; hash-indexes or not; and so on. This is in contrast to the Dynamics case, in which the space of customizations is unbounded, because new business models and new legislation may pose completely unforeseen challenges. Hence the library specialization case lends itself much more readily to a feature-oriented approach (section 2.5.8), which is unlikely to be adequate for the Dynamics case because of the open-ended customization needs of businesses.

2.4. Evaluation criteria

This section describes the four central criteria that we will use in section 2.5 to evaluate a range of customization technologies. The criteria are:

- Need to anticipate customizations. (A kernel developer concern.)
- Control over customizations. (A kernel developer and partner concern.)
- Resilience to kernel evolution (An end-user concern.)
- Support for multiple customizations (A partner and end-user concern.)

Table 2.2summarizes the evaluation results.

2.4.1. Need to anticipate customizations

Many software engineering techniques for software customization are based on some degree of anticipation of future changes. When the designer can foresee some future needs for customization and evolution of the software system, he will choose a software design that can accommodate these with as few changes as possible. Unfortunately, it is not always possible for the designer to foresee well enough the broad class of possible future customizations. In general, there is a trade-off between control and flexibility. For instance, a customization technique that permits arbitrary source code edits offers little control but high flexibility. Conversely, a customization technique that permits only a choice between a number of predetermined options offer high control but little flexibility.

We distinguish approaches that:

- **Require no anticipation.** The customization technique does not require anticipation of the customizations, whether of the customization points nor of the customization kinds.
- **Require anticipation of the customization points.** The customization technique requires the anticipation of the customization points – that is where customizations can be applied in the source code.
- **Require anticipation of the kind of customizations.** In this case, the customization technique expects the developer to foresee the content of the customizations that will be potentially applied.

2.4.2. Control over customizations

When a developer is customizing a correctly functioning software system, he takes the risk that his changes break the coherence and correctness of the current implementation. Hence, a customization technique should help in preserving the intent of the original software maker. The customization techniques typically offers control

over customization at two different staging times: design-time and run-time. We will categorize the customization techniques according to the following categories:

- **Design time control over the customizations.** Customizations can be constrained during the design stage of the software product's kernel.
- **Run-time control over the customizations.** The customization technique gives explicit support for controlling customizations at run-time (for example, activation and deactivation of certain customizations).
- **No control over the customizations.** The technique provides no explicit support for controlling the customizations.

2.4.3. Resilience to kernel evolution

A software product that has been customized will eventually need to be upgraded to a more recent version. Since the kernel will have evolved, it is likely that the customizations cannot be ported automatically to the new version. Different customization techniques have different weaknesses in this respect and require different amounts of intervention from the developer to customization to the new kernel. The third criterion is the resilience of customizations to the evolution of the kernel. We will differentiate the following three categories of explicit support for resilience to evolution of the kernel:

- **Some resilience to evolution.** The customization technique provides some resilience even to evolution of parts of the kernel related to existing customizations.
- **Restricted resilience to evolution.** Resilience only to evolution of parts of the kernel unrelated to existing customizations. Existing customizations may rely indirectly on some part of the kernel that has changed, which may affect the behavior of those customizations. In some cases this will be intended – after all, the point of changing the kernel is to change the system's behavior – but in some cases it will be unintended. We assume here that it is impossible to distinguish those two cases by automatic means.
- **No support for resilience to evolution.** The customization technique provides no explicit support for resilience to evolution of any parts of the kernel. Any part of the kernel may have been altered by some customization, so any change to the kernel may conflict with somebody's customization. Inspection (manual or tool-supported) is needed for each customization to detect whether it conflicts with a change to the kernel

2.4.4. Support for multiple customizations

Very often customizations are not made by the same company. The challenge is that those multiple customizations must be gathered together into a single product. We

will distinguish three categories of techniques with respect to support for multiple customizations. First, those who support parallel development (customizations can be independently developed and brought together at a later stage, possibly by another company). Those who support only sequential development: customization are conceived one after the other. Finally we distinguish the techniques that provide no explicit support for multiple development. We summarize those three categories:

- **Support for parallel development of customizations.** Multiple customizations can be independently developed and then subsequently applied to the same customization point in the kernel. There is still a risk that the customizations have unintended interference, for instance by updating some data structure in the kernel.
- **Support for sequential development of customizations.** If one customization is made after, and has access to the other one, then both can be applied to the same customization point in the kernel.
- **No support for multiple customizations.** No support for multiple customizations without breaking the abstractions that are used for the customizations.

2.4.5. Runtime performance penalty

Runtime performance can be an important criterion, especially for computation intensive software systems and for core software such as collection libraries. However, all the customization technologies considered in this paper have acceptable runtime performance overhead, typically comparable to a few indirections or a virtual method call per customization point reached during execution. This should be contrasted with reflective method calls, which are typically one or two orders of magnitude slower.

Since all the technologies considered here have satisfactory performance, we will not discuss this criterion further.

2.4.6. Illustration of the criteria

We describe further the last three criteria through an illustration, see figures 2.1, 2.2, and 2.3.

- Figure 2.1 illustrates our second criterion: a software product P_1 is being customized by a third-party programmer and is again customized by another programmer, resulting in a software product P_3 . The concern here is the staging time of the control for customization: design-time, runtime, etc.
- Figure 2.2 illustrates our third criterion: again, a and b are two successive customizations of an original software product P_1. The original kernel P_1^1 will eventually evolve into a new version P_2^1 . The concern here is the ease with which customizations can be ported to the evolved kernel.

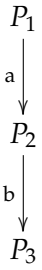


FIGURE 2.1.: FURTHER CUSTOMIZATION

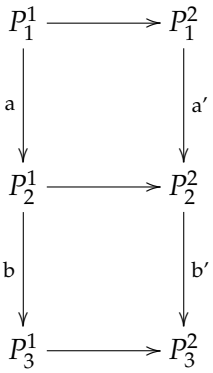


FIGURE 2.2.: RESILIENCE TO KERNEL EVOLUTION

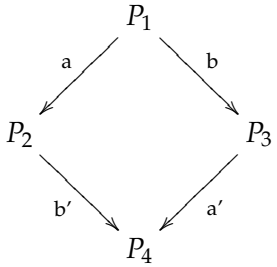


FIGURE 2.3.: SUPPORT FOR MULTIPLE CUSTOMIZATIONS

- Figure 2.3 illustrates our fourth criterion: here a and b are independently conceived customizations of an original software product P_1. Those two customizations are then used by another company to compose the software product P_4. Informally, the concern here is that the two customizations can be developed independently and brought together at a later stage, ideally yielding an equivalent software product whether one applies a then b', or b then a'. Note that this equivalence is a design goal, not a theorem – to prove such a thing would require a clear definition of the notion of equivalence.

2.5. Survey of software customization methods

Software customization is a recurrent theme within the software engineering community. Software extension in particular has received much attention from the researchers working on software reuse. Software reuse is important for economical reasons: instead of developing software from scratch one hopes to save effort and obtain better quality by reusing an existing software module, or sometimes an entire software system. They are many different ways to implement customizations. In this section, we review some of these customizations techniques, and we categorize them with respect to the criteria defined in the previous section.

2.5.1. Inheritance

Inheritance and dynamic binding are heavily used within object-oriented programming to create families of software systems. Virtual methods allow for customization by subclassing. This is essentially the "class substitution" approach for Dynamics AX customization described in 2.3.10.

For example, assume we need an `Invoice` class with a `GrandTotal` method that is customizable in the sense that the computed grand total may be modified by a customization. Then we can define a base class `Invoice` with a virtual method `After`, like this:

```
public class Invoice {
  protected virtual void After(ref double result) {
   /* do nothing */
  }
  public double GrandTotal(int input) {
   double total = ...;
   After(ref total);
   return total;
 }
}
```

If we want to customize `Invoice` to give a 5 percent discount on grand totals over 10,000 Euros, we declare a subclass in which `After` has been overridden to do just that:

```
private class CustomizedInvoice : Invoice {
 protected override void After(ref double result) {
  if (result >= 10000) result *= 0.95;
 }
}
```

Basically, as is usual in object-oriented programming, the `After` virtual method is a parameter (of function type) of the `Invoice` class, and that parameter may be (re)bound in subclasses. This particular example is a variant of the well-known Template method design pattern [83].

To ensure that all clients use this customization of Invoice one can require them to obtain Invoice instances only through a central factory method, using the Factory design pattern [83]:

```
public static Invoice Construct() {
 return new CustomizedInvoice();
}
```

Then only one place in the code needs to be changed when a new customization is created. As a further precaution against clients creating un-customized `Invoice` instances, one could declare the `Invoice` base class abstract.

Hence, customization of methods can be done by method redefinition. Dynamic binding allows for run-time selection of the method body to be executed depending on the actual type of the target object. Multiple dispatch systems such as CLOS claim to be more flexible in that they allow for the selection of the methods upon the types of all of their arguments.

- **Need to anticipate customizations.** This technique requires anticipation of the needed customization points. In the Invoice example, as in any use of the Template method pattern, the abstract template method is basically a (function-type) parameter of the class, and one needs foresight to determine which template methods are needed and where they need to be called. Also, the designer of the software system must foresee that the Factory pattern might be required to create an instance of a specific implementation of the `Invoice` class.

- **Control over customizations.** Correctness in statically-typed object-oriented languages is mainly supported by the type system. The compiler will enforce at design-time that the method to be called exists (no "Method not found" exception at run-time) and that the formal and actual parameters are type-compatible. Hence the control is done at design-time. Other languages (such as Spec#, JML, etc.) allow for behavioral specification by the use of contracts. Contracts are assertions that can be be checked at run-time, or, in some specific cases, verified at compile-time. As an example, we could add a post-condition

to the `After` virtual method to ensure that the customized variant of Invoice returns a non-negative value.

```
public abstract class Invoice {
  protected abstract void After(ref double result)
    ensures result >= 0;
    ...
}
```

- **Resilience to kernel evolution.** When the abstract class `Invoice` evolves, customized version of the software system might stop functioning correctly or not even compile any longer. For example, using C#, if the type of the formal parameter result in the abstract method `After` in class Invoice is changed from double to int, the compiler will reject the existing customized versions. The current version of C# does not allow any form of variance in the redefinition of formal parameters in subclasses. Now consider the case that the signature of the abstract method `After` does not change in the new version of that base class, but that its post-condition now requires that the result is positive. We say that the post-condition of the abstract base method was strengthened in its new version. Existing customized version of the `Invoice` class that assign zero to result now fail to satisfy the post-condition specified in the abstract method. This is likely to only be discovered at runtime, typically resulting in an exception. One may argue that this is the only acceptable output in such a case.

- **Support for multiple customizations.** Single inheritance here restricts the customizations to sequential development. More complex design patterns are required to support the composition of independently developed customizations of `Invoice`. The decorator design pattern for example will allow for more flexibility than does inheritance, allowing responsibilities to be added and removed at runtime [83]. Also, a variant of the proxy pattern allows to chain proxies, which provides support for multiple successive customizations. Note that the order in which proxies execute can be crucial for correctness.

The chief advantage of the virtual method approach to customization is that it is well understood and supported by mainstream programming languages such as Java and C#. Evolution of the base class does not require any changes to the customizations (subclasses) so long as no base class customization points are removed and no customization point data types are changed. In particular it is not necessary to edit the same section of source code, so one avoids the attendant risks of one customization overwriting another one, and difficulties in upgrading that section of source code.

The chief disadvantages of this approach to customization are that it requires foresight as to which customization points may be needed, and that multiple serial customizations of the same class cannot be developed independently of each other: one customization must be a subclass of the other customization, and hence must be aware of the existence of that other customization.

2.5.2. Information hiding using interfaces

Interfaces allow one to hide some of the design decisions that are not relevant to clients. Since implementation details are unknown to clients, they do not become dependent on them, and it is much easier to evolve the specific implementation – hence the popular slogan, "Program to an interface, not to an implementation" [83]. Also, by combining information hiding and inheritance, programmers can extend existing interfaces in a subtype with new operations without breaking existing clients, this is the traditional approach to evolution in a object-oriented setting.

Even if interfaces support evolution of their implementations, one has to keep in mind that the interfaces themselves may need to evolve. Even if some design decisions can be hidden behind an interface, as proposed by Parnas [159], the published interfaces themselves cannot be changed without taking the risk of breaking a large number of external software systems that depend on them. An apparently harmless modification, such as adding a new operation to an interface in C#, can cause great trouble: all the existing classes that implement the previous version of the interface will have to be modified to support the new operation. Abstract classes, as found for example in Java and C#, are more interesting in this respect as they can sometimes meaningfully provide a default implementation for a new operation. Consider the following abstract class Invoice:

```
public abstract class Invoice {
 public abstract ICollection<Item> Items { get; }
}
```

It is possible to add a method `GrandTotal` to this abstract class without breaking the existing concrete subclasses:

```
public abstract class Invoice {
  public abstract ICollection<Item> Items { get; }
  public virtual double GrandTotal() {
   return Items.Sum(item => item.Price * item.Quantity);
  }
}
```

Note that if there is already a (non-virtual) method with the same name in the subclass, the compiler will give a warning that the subclass implementation of `GrandTotal` hides the inherited member. Note also that the default implementation provided by Invoice can be sub-optimal. For example a subclass that maintains the current total in an instance variable will gain from overriding `GrandTotal` and directly returning the instance variable.

```
public class InvoiceImp : Invoice {
  ...
  public override double GrandTotal() {
  return currentTotal; // instance variable
  }
}
```

The problem with abstract classes is that a class can only have one base class (in Java and C#), whereas it can implement multiple interfaces. This is not the case for languages that support multiple inheritance. But multiple inheritance tends to be criticized for its complexity and the problems that it brings along – such as the infamous diamond inheritance problem [134].

The Component Object Model (COM) [172] uses interfaces to support evolution of components as well as client programs. A component can be used only through its functions (operations, methods) as originally advocated by Parnas [158]. An interface is a set of functions, where each function is described by its signature: its name, its parameters (number, order and types), and its return type.

The following restrictions on COM components and their interfaces help mitigate evolution problems:

- An interface (with a given interface identifier) must remain forever unchanged once it has been published.
- A component may support any number of interfaces, and the set of interfaces it supports may change over time.
- A client program can, at runtime, ask a component whether it supports a particular interface (using its interface identifier) and hence whether the component supports particular methods.

The restrictions support evolution of components, because an updated component may exhibit new functionality through an additional interface, while continuing to support its old interfaces. The updated component will continue to work with existing client code, because such code will continue to ask the component for its old interface and will be unaffected by new functionality.

The restrictions also support evolution of the client code. Obviously, any change to the client that does not require new component behavior, will just work with old and new components alike. If a client is updated so that it would prefer to get some new behavior from a component, but can work with old client behavior (only less efficiently, say), then the updated client simply asks the component whether it supports the most desirable new interface that exhibits new behavior, and failing that, asks it whether it supports the second-most desirable interface, and so on. Hence this supports any number of steps of evolution.

If an updated component stops supporting some functionality (for instance, because it has been deprecated for security reasons), it will have to stop supporting some old interface. Client code will discover that at runtime when asking for the interface. Depending on the robustness of the client design, and the amount of foresight that went into the design of the interfaces, the client may be able to fall back on some other interface supported by the component; if not, it must give up.

The latter scenario shows one drawback of the COM model: mismatches in component evolution will not be discovered at compile time or deployment time, only at runtime, when the client asks the component whether it supports the requisite interfaces.

- **Need to anticipate customizations.** Following the concepts of information hiding, the designer has to come up with a list of design decisions which are likely to change. Hence there is a strong requirement to anticipate changes.
- **Control over customizations.** One of famous epigrams by Perlis [160] reads: "Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication". Types allow for a limited form of checking. Contracts, mentioned previously, are sometimes used to extend checking – but most of the control over customizations is typically done at design-time, through the use of static type checking.
- **Resilience to kernel evolution.** As long as the new version of the kernel conforms to the published interface, the program will still compile. Of course more guarantees than just type-conformance are typically needed to ensure correctness of the software system (as explained in the criteria 2.4).
- **Support for multiple customizations.** There is no direct support for independently developed customizations, since the implementation of a specific interface is provided by a single class. Using a combination of inheritance and information hiding would allow for multiple sequential customizations (in the context of single inheritance), but using information hiding alone will not.

2.5.3. Parametric polymorphism

Parametric polymorphism supports evolution because it can decouple some design decisions. For example, the designer of a new class `Stack<T>` will not have to foresee the possible kinds of elements that will be contained in the stack, and yet can enjoy type safety. Without parametric polymorphism, the designer of the class Stack would have to either make a new version of the class for each possible kind of element contained, such as `StackOfPerson`, `StackOfInt`, and so on, or he would have to compromise type safety by losing type information and using type casts, as in `Person p = (Person)myStack.Top`.

However, with parametric polymorphism or generic types as in Java, C# and ML, the behavior of a parametrized type or method is the same for all type parameter instances – as implied by the term "parametric". Hence parametric polymorphism may support evolution but not really behavioral customization. This is in contrast to templates in C++ [188] and polytypic programming and generalized abstract data types in Haskell and extensions of C# [107], but we shall say no more about those mechanisms here.

- **Need to anticipate customizations.** In the previous Stack example, parametric polymorphism does not depend on anticipation of customization of the classes of the various element that will be stored in the stack – if the class `Person` changes, the class `Stack` does not have to change. But very often, we have to do more that just storing and retrieving objects from a collection: we need to

use constraints on the formal generic types. For example if a class `Invoice` is seen as container of priced items, it is reasonable to require the first generic type to be constrained by an interface `IPriced`. But if such a constraint is used on the formal type parameter, then we are back on the some problem as for information hiding: the interface `IPriced` can evolve. (Also one should note that the choice of using a generic type for a specific type declaration represents a form of anticipation itself.) For example, using C#:

```
public interface IPriced { double Price { get; } }
public class Invoice<T> : Stack<T> where T : IPriced { ... }
```

- **Control over customizations.** Similarly to other language based techniques presented above, the type system ensures some degree of correctness. The control over customizations is performed at design-time.
- **Resilience to kernel evolution.** A class `Stack<T>` with an unconstrained type parameter, as above, need not change when the item type `T` changes. However, a generic type `Painting<U> where U : Drawable` with a constrained type parameter `U` may need to change to be applicable to a new argument type.
- **Support for multiple customizations.** Parametric classes can have several formal type parameters, each of which can act as a placeholder until a runtime type is used [65]. One could devise a solution where each of these placeholders is used for a different customization.

2.5.4. Synchronous events

In C#, so-called synchronous events, or callbacks, provide a flexible way to customize behavior when one can foresee where customizations are needed. To add a customization point, one first declares a suitable delegate type (that is, function type), such as `After`:

```
public delegate void After(ref double result);
```

Then to prepare a class for customizations, we add an event field such as after to the class, and insert a conditional call to that event at the customization point:

```
public class Invoice {
  public static event After after;
  public double GrandTotal(int input) {
  double total = ...;
  if (after != null)
    after(ref total); // Event raised here return total;
  }
}
```

Now assume we need a customization to give a 5 per cent discount on invoices over 10,000 Euros. The customization is added as a suitable anonymous method to the static event field of the `Invoice` class:

```
Invoice invoice = new Invoice();
Invoice.after += delegate(ref double result) {
  if (result >= 10000) result *= 0.95;
};
```

When the `GrandTotal` method of the `Invoice` class reaches the customization point, it will raise the event and call the anonymous method, which will reduce the total variable by 5 percent if it exceeds 10,000 Euros.

In the above example we associated the event with the class (as a static field) and hence obtain class-level customizability as in the object-oriented approach in section 2.5.1. Alternatively, one might use an instance field to obtain instance-level customizability.

- **Need to anticipate customizations.** There is a strong need to anticipate customizations, because one must create the necessary events and raise each event at all appropriate places, in the right order. Also, the type of the event being sent requires some insight into the forthcoming customizations.
- **Control over customizations.** On one hand, the event argument types impose restrictions that support design-time control over customizations. One the other hand, triggering of events can be turned off at run-time providing a form of run-time control over customizations.
- **Resilience to kernel evolution.** The event model is quite fragile under changes to the base program: existing events may have to be raised at more or fewer places.
- **Support for multiple customizations.** Multiple customizations can be made simply by attaching multiple event handlers, so simultaneous development of customizations is straightforward. This of course does not prevent unwanted interactions between customizations as mentioned in the criteria section 2.4. Moreover, the order of event handler invocation may be significant, yet it may not be feasible to control the order in which handlers are invoked.

The chief disadvantage is that the event model is very dynamic events can be attached and removed at runtime – so it is difficult to determine statically the properties of a system built with event listeners.

A less obvious disadvantage is that it is difficult to provide a complete specification of the contract between the listened-to object (the one raising the event) and the listening objects (those installing the event handlers). Namely, the installation `y.Event+=x.h` of an event handler `x.h` on object `y` is the beginning of a potentially long-lasting interaction between objects x and y. Hence to understand and correctly use an event model, one must consider at least the following questions:

- What data can an event handler read, and what data can it modify? In Microsoft's Windows Forms framework, unlike Java's Abstract Window Toolkit, it is customary to pass the entire "sender" object y to the event handler, which seems to invite abuse by the event handler.

- What can the event handler assume about the consistency of data in the sender y when it is called, and what must it guarantee about the state of data in y when it returns?
- Could an event handler, directly or indirectly, call operations that would cause further events to be raised, and potentially lead to an infinite chain of events?
- At what points should an event be raised? This central design decision should be based on semantic considerations, since it strongly influences the correctness of upgrades of the kernel. For instance, it is better to specify that "the event is raised after a change to the account's balance" than to say that "the event is raised after one of the methods `Deposit` or `Withdraw` has been called". The former gives better guidance when new methods are added, or when considering bulk transactions such as `DepositAll(double[])` whose argument may be an empty array and hence perform no change to the account at all.
- What is guaranteed about multiplicity and uniqueness of events? For instance, consider a class Customer derived from class Entity, where method `Customer.M()` calls `base.M()`, and both implement an interface method specified to raise some event E. Should a call to `Customer.M()` raise the event once or twice?

2.5.5. Partial methods as statically bound events

The partial types and partial methods of the C# 3.0 programming language offer a statically bound alternative to events. Wherever there would be a call to an event handler, a call to a partial method is made instead. For instance, we may declare a partial method called after and call it as in this example:

```
public partial class Invoice {
  partial void after(ref double result);
  public double GrandTotal(int input) {
  double total = input * 1.42;
  after(ref total);
  return total;
 }
}
```

If the method call is needed, that is, if there is a customization at the call point, the partial method's body may be declared in a different source file:

```
public partial class Invoice {
  partial void after(ref double result) {
    if (result >= 10000) result *= 0.95;
   }
}
```

Then the two source files simply have to be compiled together, like this:

```
csc PartialMethod.cs PartialAfter.cs
```

If no customization is needed at the `after(...)` call point, one simply leaves out the `PartialAfter.cs` file when compiling `PartialMethod.cs`, and then the `after(...)` call will be ignored completely.

- **Need to anticipate customizations.** There is a strong need to anticipate customization points, because one must create the necessary partial methods and call them at all appropriate places.
- **Control over customizations.** The partial method argument types impose restrictions that supports control of customizations at design-time to some degree.
- **Resilience to kernel evolution.** Similarly to events, the partial method customization model is rather fragile under changes to the base program: existing partial methods may have to be raised at more or fewer places.
- **Support for multiple customizations.** Partial methods offer no explicit support for multiple customizations since there can be only one implementation of a given partial method.

The chief disadvantage of partial methods, however, is that they are not dynamically configurable; unlike events they cannot be added and removed at runtime under program control. This provides poor support for the fluid way in which developers prefer to interact with e.g. Dynamics NAV, mentioned in section 2.3.6.

There is a position between that of dynamically-bound events that may be added and removed under program control (section 2.5.4) on the one hand, and the partial methods that require recompiling and reloading the application (as described above) on the other hand. Namely, one may use meta-data to specify the association of event handlers with events, and prevent the running program from changing this association. This is the approach taken by Dynamics NAV. The approach would enable the development environment to tell which event handlers may be executed when raising a given event, and to discover potential event cycles by analyzing the meta-data and the code of the event handlers. However, the other concerns and questions about events listed in section 2.5.4 must still be addressed.

2.5.6. Mixins and traits

A mixin provides certain functionalities to the classes that inherit from it. It is sometimes said that the mixin "export its services" to the child class. When mixin composition is implemented using inheritance, mixins are composed linearly. Ducasse et al. [61] report several problems traditionally associated with mixins. For example, it is reported that class hierarchies are often fragile to changes since simple changes may impact many parts of the hierarchy. Traits can be seen as an attempt to solve some of the problems caused by mixins. A trait is, simply, a set of methods. A trait is not coupled with the class hierarchy. Traits can be composed in arbitrary order (in their original definition) and can be used to increment the behavior of an existing

class. Ducasse et al. emphasize that, using traits, the two roles of "unit of reuse" and "generator of instances" can be respectively assumed by traits and classes, whereas both roles are traditionally assumed by classes in object-oriented languages [61]. And since traits are divorced from the class hierarchy, they do not suffer from the problems associated with multiple inheritance.

Scala uses both mixins and traits to solve the code reuse limitations posed by single inheritance [152]. Its mixin class composition mechanism allows for the reuse of the delta of a class definition. The following example defines a trait `Invoice` with an abstract method `GrandTotal`. The class `InvoiceImpl` will provide the implementation for this abstract method. Note that the two are, for now, completely unrelated: `Invoice` and `InvoiceImpl` can be compiled independently. For the sake of simplicity for the example, the method implementation returns a constant.

```
trait Invoice {
  def GrandTotal: double // Abstract definition
}
class InvoiceImpl {
  def GrandTotal: double = 10 // Candidate implementation
}
```

A different developer (for example, in a partner company), can provide a customization of the method `GrandTotal`.

```
trait DiscountInvoice extends Invoice {
  abstract override def GrandTotal: double =
    super.GrandTotal * 0.95
}
```

Note that the developer implementing this customization does not have to know about the concrete implementation; his customization extends the trait `Invoice` and not the implementation class `InvoiceImpl`. Method `GrandTotal` is declared above as abstract since it overrides a method which is not defined. Similarly, another developer, (e.g., at another partner company), can define another customization implementing a simple 1 Euro tax rule:

```
trait OneEuroTax extends Invoice {
  abstract override def GrandTotal: double = super.GrandTotal + 1
}
```

Finally, a customer might want to combine the implementation `InvoiceImpl` with the two traits `DiscountInvoice` and `OneEuroTax` that customize the behavior of `GrandTotal`:

```
class DiscountFirst extends InvoiceImpl with DiscountInvoice with
   OneEuroTax object Test {
  def main(args : Array[String]) : Unit = {
  // (10 * 0.95) + 1
  println("Total " + (new DiscountFirst).GrandTotal)
  }
}
```

Note that in this particular example, the order of the with clauses is significant, due to the linearization of the super calls. In this case, the discount will first be applied on the grand total, and then the one Euro tax will be added.

One of the problem with traits is that they usually do not give direct support for state. Traits must be stateless, which imposes some strict limitations on their use. Note that the traits community is actively working on stateful traits but the current proposals also have some limitations (instance variables are local to the scope of traits, with some exceptions), see [26].

- **Need to anticipate customizations.**Traits are attractive in our case since they allow for fine-granularity code reuse. But some foresight is required to design the collection of traits in a way that will be be most convenient for the person performing the customizations, especially the specific grouping of methods into traits.
- **Control over customizations.** The compiler ensures type correctness. Using traits, the control over customizations is performed at design-time.
- **Resilience to kernel evolution.** We showed in our example that the customizations are decoupled from `InvoiceImpl` since they do not even need to know about its existence. One the other hand, if the base trait Invoice changes, the customizations will have to be adapted.
- **Support for multiple customizations.** The previous example demonstrated that `InvoiceImpl`, `DiscountInvoice` and `OneEuroTax` can all be developed independently, and finally composed together by the end-developer.

2.5.7. Aspect-oriented programming

Aspect-oriented programming [110] provides an alternative to the event models described in section 2.5.4 and section 2.5.5. Although some realizations of aspect-oriented programming restrict the insertion of extra code to the beginning or end of a method body, others allow code to be inserted at arbitrary (but previously identified) places in a method body [63]. Clearly the latter is equivalent to raising events at those places in the method.

One concern that speaks against this approach is that a well-designed method should encapsulate a state change that results in a coherent object state, so it seems to go against software engineering principles to permit arbitrary modifications to a method's body. This concern is similar to the concern that an event handler should not modify the event sender object in arbitrary ways; see section 2.5.4.

Here we consider only a rather special case of aspect-oriented programming, namely aspect-like static program rewriting. We use Yiihaw, a static aspect weaver for C# that works by rewriting of bytecode files [104]. It reduces runtime overhead relative to event-based customization and permits static checks. However, while Yiihaw's pointcut language permits some quantification, it is not particularly expressive. Other aspect weavers, such as AspectJ [111], would provide more fine-grained

customization, which would be an advantage compared to event-based customization.

Customization using aspects Consider again customization of the Invoice example already seen in section 2.5.1 and section 2.5.4. Assume the `Invoice` class is declared on a lower layer with an instance method `GrandTotal`:

```
public class Invoice {
  public virtual double GrandTotal() {
  double total = ...;
  return total;
  }
  ... other members ...
}
```

As before, assume that at the higher layer we want to customize this to give a discount when the grand total exceeds 10,000 Euros. To do this, we separately declare an advice method as follows:

```
public class MyInvoiceAspect {
  public double DoDiscountAspect() {
   double total = JoinPointContext.Proceed<double>();
   return total * (total < 10000 ? 1.0 : 0.95);
  }
}
```

and compile it, and then write an interception pointcut:

```
around * * double Invoice:GrandTotal() do MyInvoiceAspect:
    DoDiscountAspect;
```

The target assembly and the advice assembly are compiled using the C# compiler and then woven by an aspect weaver. In the resulting woven assembly, the `GrandTotal` method of the `Invoice` class will behave as if declared like this:

```
public class Invoice {
  public virtual double GrandTotal() {
   ... complicated code ...
   return total * (total < 10000 ? 1.0 : 0.95);
  }
  ... other members ...
}
```

The resulting woven method has the exact same signature as the original target method.

Sequential customization by further weaving The woven method can be used as target for further weaving. For instance, we may want to further modify the `Invoice` class and its `GrandTotal` method to count the number of times the `GrandTotal` method has been called. This involves adding a field `int count` to the class and making further advice on the method.

The additional pointcut file must contain an introduction and an interception:

```
insert field private instance int MyNewInvoiceAspect:count into
    Invoice;
around * * double Invoice:GrandTotal() do MyNewInvoiceAspect:
    DoCountAspect;
```

We need to declare an advice class with a field and an advice method as follows:

```
public class MyNewInvoiceAspect {
  private int count;
  public double DoCountAspect() {
    count++;
    return JoinPointContext.Proceed<double>();
  }
}
```

After compiling the advice and weaving it into the previously woven assembly, we get a class `Invoice` that will behave as if declared like this:

```
public class Invoice {
  private int count;
  public virtual double GrandTotal() {
   count++;
   ... complicated code ...
   return total * (total < 10000 ? 1.0 : 0.95);
  }
   ... other members ...
}
```

Evaluation of aspects for customization

- **Need to anticipate customizations.** Aspect-orientation does not require foresight as to where events need to be raised, but there is an analogous though less stringent need for foresight. Namely, customization points must be expressible as join points. In the case where only "around" interceptions are expressible , foresight is needed to factorize the kernel so that all customization points are methods, but it is not necessary to foresee which ones will be customized. In contrast to events needed to factorize methods so that they are meaningful units of customization.

- **Control over customizations.** The type system of the implementation language, combined with weave-time checks performed by the aspect weaver, give some assurance that customizations are meaningful, and can point out incompatible changes when one attempts to upgrade the base system.

- **Resilience to kernel evolution.** Aspect-oriented customization is fairly insensitive to evolution of the base code so long as the names and parameters of

methods remain unchanged. However, if customized methods or their parameters get renamed, then the weaving may fail to customize a method it should have, or may wrongly customize one that it should not.

- **Support for multiple customizations.** Aspect-oriented customization supports independently developed customizations just as well as do events.

Some research indicates that an aspect approach to cross-cutting concerns makes software evolution harder, not easier, at least based on theoretical considerations [195]. It is not clear that those results extend to our use of aspects. When using aspects for cross-cutting concerns, join points are likely to be described by quantification, using only few pointcuts. However, when customizing software products, the join points are customization points and are more likely to be explicitly enumerated, using many pointcuts. Which gives more resilience to evolution is unclear.

Aspects for customization in Dynamics AX Static aspect weaving, as outlined above, offers a plausible way to perform customization of Dynamics AX applications (2.3.3):

- It preserves the layer model of Dynamics AX. This in turn offers several advantages. First, the overall philosophy will be readily understandable to the current developers at the Dynamics core development team, as well at partners and customers. Second, there is a likely upgrade path from the current AX implementation to an AX implementation based on layers and aspects.
- The aspect weaver can check, at weave time, the consistency of the modifications of upper layers with lower layers.
- Aspects can be statically woven so that they incur no performance penalty at all, and hence would perform no worse than the existing source code based customizations.

To express customizations as aspects we have used the Yiihaw aspect weaver [105] described by another paper in this volume [104]]. Although several aspect weavers for .NET have been proposed, Yiihaw seems to be especially suited: it introduces no runtime overhead at all, it statically checks aspect code ahead of weave-time, it statically checks consistency of weaving, and it can further weave an already woven assembly as indicated above. This is necessary in the Dynamics AX scenario where lower layer code gets customized in a higher layer, and the result gets further customized in an even higher layer; see 2.1. The limitations of the Yiihaw pointcut language and its notion of aspect mean that some will consider it a tool for feature composition rather than a full-blown aspect weaver, but it seems adequate for the purposes considered here.

2.5.8. Software product lines using AHEAD

Feature-oriented programming has been developed over many years by Batory and coworkers [22, 23, 21, 166]. Part of the motivation for this work is the insight that future software development techniques will synthesize code and related artifacts (such as documentation) extensively. The research efforts have focused on structural manipulation of these artifacts. These ideas can be seen as part of the meta-programming research field: programs are treated as data, and transformations are used to map programs to programs.

These ideas gave rise to concrete tools, among which GenVoca and AHEAD [20] are prime examples. These tools were used to synthesize product lines for various domains such as database systems and graph libraries. More concretely, using a product line, a user can select among a set of predefined features and the tool will combine artifacts to generate a program that implements the desired functionality. The user typically uses a declarative domain-specific language to express the feature selection he wants.

Among the various artifacts handled by these tools we henceforth focus our attention on source code. The mixin is one of the core object-oriented concepts that underpin this approach to code composition. In this context, a mixin is a class whose superclass is specified as a parameter. Using the variant of Java proposed by AHEAD, we can write a customization for the invoice example from section 2.5.6:

```
layer tax;
refines class invoice {
  overrides public double grandTotal() {
    return Super().grandTotal() + 1;
  }
}
```

This customization adds one Euro, a “tax”, to the grand total computed in the base code (omitted for the sake of brevity). Note that the customization is defined in a named layer “tax”. The discount customization, that we saw previously, can be programmed similarly in a layer “discount”. The discount is unconditional in this case to make the example a bit shorter.

```
layer discount;
refines class invoice {
  overrides public double grandTotal() {
    return Super().grandTotal() * 0.95;
  }
}
```

To compose the base code invoice with the customizations, the programmer can choose between two tools. The first one, called “mixin”, will transform the composition into a class hierarchy. Using this tool, each customization will be turned into an abstract class that extends another abstract class, with the exception of the last customization, discount in our case, which is turned into a concrete class. Each class name in the hierarchy is a mangling of the name invoice with the name of the

originating layer – again with the exception of the class that corresponds to the last customization (since it is the one that will be instantiated).

```
package invoice;
abstract class invoice$$invoice implements invoice {
  public double grandTotal() { return ...;}
}

abstract class invoice$$tax extends invoice$$invoice {
  public double grandTotal() {
    return super.grandTotal() + 1;
  }
}

public class invoice extends invoice$$tax {
  public double grandTotal() {
   return super.grandTotal() * 0.95;
  }
}
```

The other tool, called "jampack", offers a more compact encoding of the code composition. In this case, the base code and the customizations are turned into static methods, with the exception of the last customization which is mapped into a non-static method. The name mangling for method names is very similar to the name mangling for class names performed by the other tool.

```
package invoice;
public class invoice {
  public final double grandTotal$$invoice() {
    return ...;
  }
  public final double grandTotal$$tax() {
    return grandTotal$$invoice() + 1;
  }
  public double grandTotal() {
    return grandTotal$$tax() * 0.95;
  }
}
```

Mixins are often not conceived in isolation, but rather "carefully designed with other mixins and base classes so that they are compatible" [20]. It is easy to see in the above example that overriding `grandTotal` might break some other code that relies on its initial semantics.

A particularly interesting feature of this work is the composition algebra and design rule checking. The design rules are necessarily domain-specific, for instance, for the domain of efficient data structures. Batory's feature-oriented programming for product lines [19] seems highly relevant and makes many points of value for evolvable software products.

- **Need to anticipate customizations.** Similarly to classical object-oriented pro-

gramming, it seems that product-line engineering requires that the programmer has a good understanding of the domain. Classes must be designed is such way to accommodate for mixin composition conveniently.

- **Control over customizations.** The AHEAD tools suite will check that the types are conforming, but no guarantee is given on the semantics. It is up to the designer to ensure that the prescribed composition of code artifacts is meaningful for the domain.

- **Resilience to kernel evolution.** If we assume the closed-world assumption that is common within software product lines, all the potential customizations and their possible interactions are known. Therefore an evolved kernel can be organized in such way that any existing choice of features will continue to work as intended. This does not mean that upgradability comes for free: the kernel developer must handle these interactions and handle them.

- **Support for multiple customizations.** The product line is the family of classes created by mixin composition. As noted before, the mixin approach requires that mixins are not created in isolation, but rather carefully designed together, which basically assumes a closed world of possible customizations. Therefore there is no support for independently developed customizations.

2.5.9. Software product lines using multi-dimensional separation of concerns

The Hyper/J framework and tool developed by Tarr, Ossher and others at IBM Research [157] support multi-dimensional separation and integration of concerns in Java programs, which may be used to implement software product lines. A Hyper/J prototype implementation [100] is publicly available, but is not currently actively supported. In particular, the prototype does not seem to work with the latest version of the Java runtime environment, which seriously limits its usability. Hyper/J shares many goals with aspect-oriented programming, such as the decomposition of software systems into modules, each of which deals with a particular concern.

A *dimension of concern* is a class, a feature, or a software artifact. For example, a class in a code base represents a class concern. Each dimension of concern gives a different approach to software decomposition. Tarr and others coined the term "the tyranny of the dominant decomposition" to signify that a programming language typically supports only one (dominant) decomposition, such as classes in case of object-oriented languages. Consequently some concerns cannot be implemented in a modular manner, and the code fragments implementing them will be scattered across the modules that arose from the dominant decomposition [157, page 5]. For instance, logging (of method calls) is an example of such as cross-cutting concern, often cited in aspect-oriented programming.

Using Hyper/J, decomposition can be done simultaneously along multiple dimensions of concern: The class is no longer the main decomposition mechanism in an

object-oriented language, putting class, package, and functional decomposition on a more equal footing. The Hyper/J tool takes care of the interaction across those different decompositions. The goal is to encapsulate into new modules those concerns that were previously scattered over the classes.

By combining selected concerns into a program, a programmer can create a version of the software containing only selected features, even if the original software system was not written with separation of features in mind [157].

Units are organized in a multi-dimensional matrix, where each axis is a dimension of concern, and each point on the axis is a concern in that dimension. The main units in Hyper/J are functions, class variables, and packages. Concern specifications are used to specify the coordinate of each unit within the matrix, using the notation:

```
x: y.z
```

where `x` is a unit name, `y` a dimension and `z` a concern.

We now give a Hyper/J solution to the invoice example from section 2.5.6. Once again there is a base implementation of Invoice, now in Java. The method `GrandTotal` computes the sum of the items of the invoice, and another method called `GetTotal` will return that total.

```
package lipari.base;
public class Invoice {
  private double total; public void GetTotal() {
   return total;
  }
  public void GrandTotal() {
    total = 10; // Dummy implementation
  }
}
```

In another package, a developer defines a discount as a customization of the base implementation by writing the following class:

```
package lipari.discount;
public class Invoice {
  double total;
  public double GrandTotal(double x) {
    total = total * 0.95;
  }
}
```

Note that the name used for the method and for the instance variable mimic the ones from the base code, but the package name is different. The "one Euro tax customization" can be specified similarly to the discount customization above, in a separate package. Note that both the customizations and the base class can be compiled completely independently.

A programmer can then compose the base code with the two customizations by writing the following Hyper/J specification (some parts were omitted for brevity). First, we specify the concerns:

```
-concerns
  package lipari.base : Feature.Base
  package lipari.tax : Feature.Tax
  package lipari.discount : Feature.Discount
```

In this case the mapping is simple since each concern is implemented by its own package. Then we specify that we want to compose a software system, here called LipariHypermodule, using the concerns specified above:

```
-hypermodules
 hypermodule LipariHypermodule
 hyperslices: Feature.Base, Feature.Discount, Feature.Tax;
 relationships: mergeByName;
 merge class Feature.Base.Invoice,
             Feature.Discount.Invoice,
             Feature.Tax.Invoice;
 end hypermodule;
```

Note the composition relationship `mergeByName`, which indicates that units in different hyperslices that have the same name will be fused. Using the composition specification above, the tool can generate a new software system with the selected features. The code below will correctly display the expected total, 10 Euros with a 5% discount, followed by a one Euro tax – that is, 10.5 Euros.

```
package lipari.base;
public class Main {
  public static void main(String[] args) {
   lipari.base.Invoice i = new lipari.base.Invoice();
   i.GrandTotal();
   System.out.println("Total = " + i.GetTotal() );
  }
}
```

- **Need to anticipate customizations.** Some foresight is required to identify the dimensions of concern because they determine how concerns can be combined into systems. It seems that concerns may be added to a dimension as needed.
- **Control over customizations.** Type provide some protection against meaningless compositions at design-time.
- **Resilience to kernel evolution.** If we have a closed-world assumption, similarly to what was mentioned in section 2.5.8, the evolution of the kernel can be done is such way that any existing choice of features continue to work. Of course, the same constraints mentioned in section 2.5.8 apply here.
- **Support for multiple customizations.** Again as it was mentioned before, under a close-world assumption there is no support for other independently developed customizations other than those that could be foreseen when designing the kernel.

Table 2.2.: SUMMARY EVALUATION OF CUSTOMIZATION TECHNOLOGIES

Technique	*Sec.*	*Impl.*	*Refs.*	*Need to anticipate customizations*	*Control over customizations*	*Resilience to kernel evolution*	*Support for multiple customizations*
Inheritance	2.5.1	C#	[65]	(2)	(a)	(ii)	(II)
Information hiding	2.5.2	C#	[159, 65]	(2)	(a)	(ii)	(III)
Parametric Polymorphism	2.5.3	C#	[65]	(2)	(a)	(ii)	(II)
Events	2.5.4	C#	[65]	(2)	(a) and (b)	(ii)	(I)
Partial methods	2.5.5	C#	[65]	(2)	(a)	(ii)	(III)
Mixins, Traits	2.5.6	Scala	[61, 152]	(2)	(a)	(ii)	(I)
Aspects	2.5.7	Yiihaw	[104, 105]	(1)	(c)	(iii)	(I)
SPL using AHEAD	2.5.8	AHEAD	[20]	(3)	(a)	(i)	(III)
SPL using MSC	2.5.9	Hyper/J	[157]	(3)	(a)	(i)	(III)
AX Layers	2.5.10	Dynamics	[88]	(1)	(a)	(iii)	(II)

Legend: Need to anticipate customizations: (1) none, (2) customization points, (3) customization kinds. Control over customizations: (a) design-time control, (b) run-time control, (c) none. Resilience to kernel evolution: (i) some resilience, (ii) restricted resilience, (iii) no resilience. Support for multiple customizations: (I) for parallel development, (II) for sequential development, (III) no support.

2.5.10. The Dynamics AX layer model

The source-code based layered customization models of Dynamics AX was described in section 2.3.8. Here we just give a brief assessment of it for comparison with the other technologies surveyed in the following sections.

- **Need to anticipate customizations.** There is no need to anticipate customizations, since any lower layer application element can be copied to a higher layer and customized there.
- **Control over customizations.** A customization can include any edits, so there is no support for controlling customizations.
- **Resilience to kernel evolution.** The customizations are very fragile to base program evolution; it is entirely up to the developer to identify what changes need to be made to the customizations.
- **Support for multiple customizations.** The support is very good if the changes are made sequentially, for instance, if a customized component is further customized at a higher layer.

2.5.11. Summary evaluation

Table 2.2 summarizes the properties of the technologies surveyed.

2.6. Conclusion

We defined the upgrade problem as the conflict between customization and evolution of flexible software products. We have presented the Dynamics enterprise resource planning systems as prime examples of such software products, and discussed how they are structured and customized, underscoring that the upgrade problem is a real one and the focus of much attention also in industrial contexts.

We then considered a number of software technologies and practices that are traditionally used for customization and for creation of families of related software systems. For each one, we have given a description, an example, and an evaluation in relation to four criteria: need for foresight, support for correctness, fragility in view of kernel evolution, and support for merging of independent customizations. A tentative conclusion of this investigation is that all the technologies considered offer adequate runtime performance. Static aspects (in the Yiihaw guise [104]) and traits offer good static correctness guarantees and good support for independent customization. They fit well with the structure of Dynamics AX (section 2.3.9) but rely too much on build-time software composition to fit well with the development practices around the Dynamics NAV (section 2.3.6). Also, they both require some foresight in defining the customization points, which must be classes and methods, and they are rather fragile in case class names or method names in the kernel are changed as a consequence of kernel evolution.

Software product lines offer some interesting potential to deal with the upgrade problem but their closed-world assumption does not fit the domain of enterprise resource planning (ERP) systems that we took for a case study here. Such systems must be customizable to unforeseeable legislation and new business models, and this poses additional upgrade challenges.

Acknowledgments

Thanks to the anonymous referees whose comments led to many improvements and clarifications. This work is part of the project Designing Evolvable Software Products, sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen. For more information, see http://www.itu.dk/research/sdg.

3

Customizations and upgrades of ERP systems: An empirical perspective

Yvonne Dittrich
IT University of Copenhagen
dittrich@itu.dk

Sebastien Vaucouleur
IT University of Copenhagen
vaucouleur@itu.dk

Reprinted from the technical report [57] (long version) with minor corrections and clarifications. Short version published in the Workshop on Cooperative and Human Aspects of Software Engineering, ICSE [58]. Complemented version accepted for publication in the IEEE Software journal [59].

Abstract

An increasing number of software systems are developed by customizing a standard product that provides the major part of the functionality. The customizations of *Enterprise Resource Planning* (ERP) systems are examples of such a practice. Nonetheless, little empirical research on the specific characteristic of this kind of software development is available. Do the recommendations for "normal" software development also apply in this case? We present an empirical study on ERP customization practices based on video recordings, interviews and a survey. The observed and reported practices challenge some of the principles of software engineering acknowledged as good practices. Based on the analysis, we discuss essential challenges and identify directions to take when addressing specific difficulties. Besides bearing the potential to influence the development of future generations of enterprise systems, the presented research provides insights in software development practices changing and amending a software product *from within* rather than developing the central functionality from scratch, or re-using components *from without*.

3.1. Introduction

We present an empirical study investigating customization and upgrade practices around *Enterprise Resource Planning* (ERP) systems. ERP systems aim at supporting most of the administrative and management processes of an organization, e.g. finances, human resource, supply chain management, manufacturing, customer relations. *Customization* means changes to the functionality of the base program itself that become necessary when the flexibility provided by configuration facilities is not sufficient. (See section 1.3.3 for a definition of the terms *customization* and *configuration*).

Software products A substantial part of today's software development builds on standard systems like ERP systems. Such software products provide functionality that would be expensive to develop from scratch. The vendors can spend more resources on domain analysis and interaction design than a single company could afford. New releases allow to keep the infrastructure up-to-date. Often, these software products have to be customized to fit with the organization they should support. Customizations are considered problematic in the related Information Systems literature, especially as they require additional effort when they should be ported to a new release [31]. However, we have found little research on ERP customization practices or research investigating tool support for practitioners when customizing software products and upgrading the customizations.

Motivation To better understand and support this kind of software engineering, empirical research is needed. What kind of customizations are developed? How do developers proceed? What tools and conventions do they use to keep track of their changes? What are the relevant skills to develop add-ons, and to fit them into a program providing the major part of the functionality? How can this kind of software development be supported? Are there ways to better support upgrades of customizations technically? In section 3.5, we highlight the challenges our observations provide for software engineering tools and techniques.

Road-map The remainder of the article is structured as follows: the next section introduces the ERP systems which are subject to the observed and reported practices. Thereafter, we discuss the methods we used for the empirical research. Section 3.4 contains the analysis of our research material. First, we present how a developer solves a typical customization problem, thereafter, we present the analysis of our field material with respect to a number of dimensions reaching from project organization over testing and quality assurance to cooperation practices. Finally, section 3.6 concludes.

3.2. The ERP systems considered

The empirical study concerns the ERP systems *Dynamics AX* and *Dynamics NAV* whose development organization now belongs to Microsoft Dynamics [88, 139]. Both are developed for small and medium size companies. As all other ERP systems they have to be configured to fit with the structure of the specific organization. This is done through a configuration interface inserting base data, e.g. defining roles and rights for every single user. For the case that customization is required that is not supported through configuration possibilities, an integrated development environment is provided. The design philosophy behind both systems is to open up for customizations rather than to provide an overhead of unused functionality.

3.2.1. Dynamics AX

Dynamics AX, previously named Axapta, addresses medium size companies. Of the two ERP systems discussed here, it provides a more flexible model and more substantial customization possibilities. Customizations are done using a proprietary object oriented language, called X++, similar to C# or Java.

Support for customizations On top of the normal object oriented functionality of classes and inheritance, the construct of layers is provided. These layers are meant to support customizations. A so-called *object*, in AX parlance, in an upper layer *shadows* objects with the same name in any of the lower layers. An object can be for example a method, a table, or a form. The development environment that is part of the product gives access to a large part of the application code. Though developers are not prohibited from editing code in the lower layers distributed by the provider, copying the code to one of the customizations layers before adapting it is regarded as good practice. The original version of the business object can then be called from within the customization code.

Development experience An important point in the development experience is that changes are immediately effective once they are saved and the code is compiled. The application does not need to be restarted, and the developer can try the modification without leaving the development platform. Even the development platform can be extended. As we will see below the development of custom tools by development organizations is not unusual.

3.2.2. Dynamics NAV

Dynamics NAV, previously named Navision, is targeted to small companies. This ERP product offers an alternative to the sophisticated technological choices made by Dynamics AX. NAV is sometimes qualified as simpler and easier to use than AX. Accordingly, it is typically used by companies that need few customizations.

Support for customizations Customizations are done by first defining business objects, for example classes, table and forms, in an interactive environment. The behavior of these business objects is then customized by editing methods of a class defined in this manner using a proprietary procedural language. The business objects and their relation to other objects are stored as meta data. Only the procedures are compiled.

Development experience Similarly to Dynamics AX, the Navision development environment allows for incremental development and iteration between editing and running the application.

3.3. The research method

Empirical research on ERP system customization provides a number of challenges. First, real world customizations touch on the intellectual property rights of three independent organizations: the ERP vendor, the consultancy selling, implementing and customizing the ERP system and the customer paying for implementation and customization of the system. Second, it is unsure whether – and when – the learnings from the empirical research will result in new features in a future version of the ERP system. So the practitioner's contribution in form of time might not pay back at all (in other words, practioners have little *incentive* to participate in this study). Third, to understand the complexity of the tasks and to be able to formulate relevant questions, the structure of the ERP systems under discussion has to be understood to some extent.

Process We addressed this challenge by combining different data collection methods, applying a flexible approach [171] that allows adjusting the research instruments to a developing understanding of the domain: first, we reviewed existing empirical material (video recordings) and discussed the issues around customization and upgrade with our industrial partner, Microsoft. Based on the initial knowledge acquired, we devised an online survey targeted towards ERP practitioners. Finally, we conducted face-to-face interviews with lead developers and managers working in the ERP customization business. With this triangulation, we aimed at improving the confidence in the results of an empirical study. We used multiple sources of data, multiple observers, multiple methods to complement the respective short comings of each method and to counter a possibly biased interpretation.

3.3.1. Video recordings

Data collection Existing video recordings and empirical reports came from empirical studies previously conducted within Microsoft, whose subject was not the upgrade problem. The video recordings were part of a user experience study addressing the functionality and interaction design for the development environment. They

consisted of several hours of screen captures of customization and about six hours discussion around previous recordings between practitioners, interaction designers and developers. The agreement of the subjects was acquired prior to analysis. We logged the discussion videos as they provided both walkthroughs through specific customizations and discussion of the functionality of the development environment.

Analysis The analysis of the video recordings provided the basis for the survey and for the interview guidelines. The accounts of development practices from the interviews could be supported by parts of the video material. The example introducing the analysis part is based on one of the videos. We complement the analysis with findings from the video analysis when suitable.

3.3.2. Survey

Data collection The survey was done online using a dedicated survey tool. We asked practitioners through online forums to kindly help us by answering the questions according to their experience in the field. The questions addressed mainly topics around upgrade practices, difficulties and the use of tools for upgrades. 42 persons answered the survey. We asked the participant as an optional question to leave their contact details, and approximately 2/3 of them did so. Using the specialized forums (where developers discuss bugs, workarounds, best practices, etc.) seems a-posteriori a good choice, the persons who answered the survey are involved in the intricacies of ERP systems on a daily basis.

Analysis Figure 3.1 gives an overview of the tasks that people who answered the survey worked with. The question allowed multiple answers: especially in small companies, employees often have more than one responsibility. Table 3.2 indicates the experience of the practitioners answering the survey. The sample is well versed in the Dynamics NAV product, but seems less experienced in the Dynamics AX product. From a statistical point of view, the number of participants is low. We use the survey answers where suitable to support our analysis based on the interviews.

3.3.3. Semi-structured interviews

Data collection Three interviews of approximately an hour and a half were performed on practitioners (excluding the test interview). All of them were managers and lead developers in consultancies, focusing on customization around Microsoft Dynamics ERPs. They all had extensive technical experience with the ERP systems. We developed an interview guideline based on the analysis of the video recordings. We addressed technical issues as well as cooperation and knowledge sharing. The guideline was tested by performing a test interview with a senior Microsoft engineer. The test interview confirmed that the interview guideline was appropriate for our objectives: the questions had the right level of technicality and the interview lasted

Table 3.1.: TASKS OF THE SURVEY RESPONDENTS IN THEIR RESPECTIVE COMPANIES

Tasks of the survey respondents (multiple answers allowed)	
35	Customization of ERP systems
31	Upgrade of ERP systems
29	Customer site implementation
23	Requirements and specifications
23	Managing a development team
19	Customer support
8	Sales
3	Marketing

for a bit more than an hour, which was the time frame we aimed at. The interviews were recorded and transcribed.

Analysis The transcripts were coded in a grounded theory open coding manner [171] independently by both authors. The coding provided the basis for the structure of the analysis part of the article.

3.3.4. How valid are our findings?

Data Both from a quantitative and a qualitative perspective, the data is rather limited. The number of respondents of the survey does not allow for a statistical analysis; three interviews are not enough to be sure to catch relevant aspects of a practice. More interviews and also observations would be needed. Given these limitations, we also see indicators that the findings are generalizable beyond the specific organizations we studied. The three interview partners reported different levels of formality in their approaches, but there is a high level of agreement regarding the challenges of customizations and upgrades, the concrete practices, and conventions. Central aspects are confirmed by the data from the video recordings and the survey as well.

Relevance to other fields Whether the findings apply to the customization of systems like content management systems, simulation systems, or games is an open question. Here, more research is needed.

Sample Another important factor is the experience of the respondents in the respective ERP systems under study. The answer to this question indicates that the sample is well versed in the Dynamics NAV product, but seems less experienced in the Dynamics AX product, see table 3.2. Note especially the high number of respondents who had 6 months or less of experience with Dynamics AX. Table 3.2 is the result of two independent questions, respectively concerning AX and NAV; 7 respondents are experienced with both products.

Table 3.2.: EXPERIENCE WITH THE ERP SYSTEMS CONSIDERED

Years of Experience	*0*	*0 - 0.5*	*0.5 - 2*	*2 - 4*	*> 4*
Dynamics AX	25	3	5	2	7
Dynamics NAV	9	1	6	2	24

3.4. Business and work practices for customization and upgrade of ERP systems

Before starting with the analysis of our field material, we present an example of customization practices. Then, we start with a general characterization of the companies studied. Section 3.4.3 prepares the presentation of the practices. Thereafter, we focus on tools and conventions and on learning and knowledge sharing between practitioners. We anonymized our interview partners using symbolic names for the persons and the companies, company X (Jack), company Y (Albert), and company Z (Herbert). Similarly, we give the name Finn to the developer that we observed on the video recordings.

3.4.1. An example: logging all actions related to customers

In one of the video recordings, Finn tells about a Dynamics AX customization task that turned out to be challenging. The task was to provide an overview for the users over all transactions in which one of his customers was involved.

History table For this functionality, a new customer history table has to be created containing information from and references to different tables that held data related to customer related transactions. A form has to be created to view and navigate from that table, and a respective class is needed to collect and access the information. The new table should be populated whenever data in the other tables is updated. That means that for every relevant transaction a method call has to be inserted that in turn creates a new entry in the customer history table.

Problem solving To solve this problem, Finn discusses with some of his more experienced colleagues. The first hint he gets is to look at a similar module logging changes to one specific table. However, this does not solve the problem. Another colleague hints at a system function that collects data for a central archive – a functionality needed for public organizations in the Scandinavian countries. Finn explores the related objects, and uses the module as a template to implement the customer history functionality. Finn goes about it in an iterative way. First the table, and the respective class and form are created. The method creating the table entry is added, and a call is inserted in one of the relevant business objects. This is tested by creating a new customer and manipulating the system so that the respective data entry should be inserted in the customer history table. During this process, the developer switches

between a program editor, a table viewer and the user interface of the system. The user interface is also used to trigger transactions that then become visible as entries in the table, as well in the newly developed form. This process is continued until method calls in all relevant places are inserted, and the creation of the respective entries in the customer history table are checked.

3.4.2. The companies and the people

The companies we have been interviewing are small and medium size consultancies. Two of the three companies specialized in specific business segments, a practice which provides a head start when bidding on tenders.

> "When we started, we took every kind of company, but now we are taking production plants. We have a special *vertical*, QA [Quality Assurance] which we use to [get] into the companies." [Albert]

> "We have four segments we work with. One is called *Consultancy Business*. That is normally other software companies, other consultant companies, engineers, architects and so on. That is our segment 1. Then we have a segment 2, which is the *Public Sector* but with a speciality regarding project oriented organization within the public sector. [...] Then we have a segment called *Membership*, which is membership organized organizations. [...] And the last segment is called *Trading Service*." [Herbert 00:04:24]

Business focus More established consultancies focus on a specific market segment, developing so-called *verticals*, add-on modules complementing the basic functionality of the software, again providing competitive advantages. All the consultancies we interacted with aim at keeping customers over longer periods. Especially when developing custom solutions – like for example a module for administrating flights for a small carrier or a module administrating royalties for a film distributor; changes and developments on the business side result in new contracts for customization of the base software.

Required expertise All interviewees referred to two kinds of expertises needed to implement ERP Systems in customer organizations: the ability to understand the customer's business and configure the software, and the ability to customize the system and develop additional functionality. In two of the companies, the roles were distinguished.

> "In the old days, we had XAL and a small product. The employees were [doing] both things. But our experience with Axapta is, it is so complex that if you are going to be good at something, you cannot be good at all [aspects]. You have to either focus on developing or on the application. But of course you have to know some of the other part." [Albert]

Developers As we were focusing on customization and upgrades of customizations we focused on the development side of this cooperation. But even in the technical area, few employees hold a MSc degree in computer science. Employees often come with an engineering degree, a business degree or a professional education comparable to a bachelor in computer science. Hiring and educating developers was mentioned as one of the main challenges: general programming skills are just the basis; on the one hand, developers have to learn to understand an existing rather complex application. On the other hand, they must have enough understanding about business administration to understand the rationale behind the base application and understand the business impact of changes. New developers start by taking courses offered by the ERP provider and work with documentation and minor developing tasks under guidance of a more experienced developer.

> "[They start with doing] the simple things, reports and forms, and after half a year they are going to do more complex business logic." [Albert]

Experienced developers often have worked with different ERP systems, but most consultancies specialize in one of them.

3.4.3. Project organization and documents

The long time relationship between the consultancies and their customers results in three different ways of working. Our interviewees talked about projects for new customers, evolution and maintenance tasks with existing customers, and upgrade projects.

New projects A project starts with the consultants or "application experts" of the company discussing with the customer what is needed. Part of this initial phase is to align the expectations of the customer with the functionality the system provides. The goal is to use as much as possible of the existing functionality without customizations.

The requirements document Though the different companies follow different methodologies for analysis and design, some document – called "bill of functionality", "requirement specification" or "analysis report" – is the base for the agreement with the customer, the starting point of the development process and the base for the acceptance test by the customer.

Two processes Two different processes are used depending on the complexity of a customization: larger customization projects are using flowchart-based models of the work processes in the area under discussion. Based on that description, an example application is set up, and sample parts of the customization are prototyped. An analysis report is developed, and is signed by the customer. Simpler customizations only use a "bill of functionality" as a starting point.

Design document and tests Based on the analysis report or the bill of functionality, a design document is developed; the customizations are implemented; sometimes internal test cases are specified, and – based on the analysis report or the bill of functionality – the acceptance test is defined. The rigor of the tests depends on the quality requirements of the customer.

Developer centric versus customer centric The interview partner of Company Z tells that the format for the analysis report and the bill of functionality changed from *developer centric* notation to a format that the customer can understand. To compensate for any lacking information, the main developer associated with the project takes part in the workshops and the meetings with the customer to get a firsthand idea of the requirements.

Custom tools Some of the companies use custom tools – sometimes internal customizations of the ERP system's development environment – to keep track of all communication and documentation around the customizations for a specific customer. For example, company Z uses a project management tool developed by another consultancy from the same consortium. Such tools are further described in section 3.4.5.

Code comments All three companies use standards for code comments, both to identify own code and to document the implemented customizations.

Testing Acceptance tests normally result in a list of issues that are to be resolved before the project is signed by the customer. All interviewees problematize testing. How much and how rigorously the customizations are tested depends on the willingness of the customer to pay for the additional effort. Verticals and parts with high reliability requirements are tested more rigorously than simple reports.

Small changes Often small changes are done at the customer site: for example, to adjust the reports to a local printer set-up and to add keyboard short-cuts for often used functionality.

Customer contact Customer contact continues throughout the project. Normally that implies that both the business expert and the developers are involved.

Evolution and maintenance As previously mentioned, the consultancies also take care of evolution of the installations and customizations for their customers. Sometimes several developers are working full-time for one customer. Often, a developer is responsible for maintenance and evolution for a number of customers. In such long term cooperation, the developers often develop a more direct contact to the customer.

Ensuring continuity At company Z, each customer site installation is taken care of by one consultant and one developer in order to ensure continuity for the customer. This scheme maintains knowledge about relevant discussions and rationales for the implementation which are not always documented. "[...] Normally, we have one developer, and also, we prefer to have one consultant at least who owns the customer, who knows what is going on. Because we have a long history and the amount of documentation is also too low. So we need to know the customer's business." [Herbert].

Upgrade processes According to our interviewees, upgrades are handled similarly to a new implementation, though the existing implementation makes the process easier. The consultant goes through the existing customizations together with the customer and discusses one by one whether to keep the customization, or whether to use the default implementation in the base system – in partners parlance, *"to go standard"*.

Deferred customizations New customizations that are caused by the new functionality not being sufficient any longer or by the need of additional adaptations are postponed to a dedicated maintenance project thereafter.

Use of documentation and testing If the upgrade refers to own customizations, the documentation developed in the original design process is referred to. The resulting implementation is then tested against the old installation.

Complications The upgrade process is often complicated by the fact that some customers have had several consultancies customizing their installation. None of the interviewees appreciated to work with or upgrade customizations not developed within their own company. However, this is regarded as an unavoidable problem.

Upgrades: who and why? If upgrades are cumbersome and expensive who then initiates an upgrade and for what reasons? Figure 3.3, from the survey answers, reveals interestingly that the partners have a strong influence on the decision to upgrade. Table 3.4 shows the survey answers to the question regarding the motivation to start an upgrade project. Interestingly, getting bug fixes for the standard system scores even higher than new technologies and features.

3.4.4. Customizing ERP systems

The concept of customization is not clearly defined. Practitioners distinguish between different categories of customizations according to their complexity and the difficulties connected with each of them.

Table 3.3.: WHO INITIATES AN UPGRADE?

Who initiates an upgrade? (multiple answers allowed)	
24	The customers, on their own initiative
34	The partners (vendors) suggest the upgrade
4	Other

Categorization of customizations and structuring of tasks Starting with a bill of functionality or with a requirements specification, how do you start to work with the customizations, how do you split up the work in small chunks? We did not get a clear answer to this question. The reason might be that the term *customization* covers a range of different changes to the application:

- Most simple are customizations of existing reports (invoices, order confirmation, etc.): for example, hiding one of the fields. Often, custom reports are required to provide additional information for management. Here, additional data has to be collected either by accessing the database directly or by calling specific procedures provided by the standard system. This is often a task for new developers, who have to get familiar with the system to be able to take care of more complex tasks.
- The existing functionality of the ERP system can be enhanced: a simple enhancement would be to capture additional data for some entity, requiring the change of a form, the change of some code unit, and the extension of a database table. Our introductory example falls into this category. A more complex example would be changes to the business logic of the general ledger, a central accountancy module [Herbert]. Under this category, the integration with other systems is treated as well.
- Add-ons comprise a third category. So called "verticals" consist of more independent complementary functionality, like a contract management system

Table 3.4.: REASONS TO UPGRADE

	A^a	B^b	C^c	D^d	E^e	F^f
Typically not a reason at all	7	12	1	13	8	13
Good to have but not essential	21	21	21	22	24	15
Typically the main reason	14	9	20	7	10	14

[a] **To benefit from a new technology**
[b] **To ensure support from vendors**
[c] **To get bug fixes**
[d] **To make it easier to upgrade later**
[e] **To get compatibility with external software**
[f] **Customize and upgrade at the same time**

mentioned by Herbert, a module to administrate royalties for a film distributor [Jack] or a flight model for a small carrier [Albert]. Such add-ons are less intertwined with the standard application.

Complex features For a complex feature, some developers start up with implementing the necessary changes to the database scheme; then they add adjustments of the data model; and finally the respective forms in order to be able to view and add data are implemented. Finally, the reports are adjusted.

Challenges of customizations All our interview partners confirmed that the main challenge when customizing standard systems is the understanding of the base ERP system. According to our interview partners, even very experienced developers do not understand the whole system. Experts for specific modules are consulted if the customizations in this area are of the more complex kind.

Code exploration One of our interviewees reported how he goes about developing an understanding of an unknown part of the system: he explored the code and in parallel ran trials through the interface in order to see what tables are affected and in which way.

Lack of proper documentation Interviewees reported that a proper documentation of the code base is lacking. Often changes impact distant parts of the system. In some cases the order of procedure calls influences the result. In accordance with that, experience with a specific ERP system is reported as the main skill needed for the customization of ERP systems.

3.4.5. Upgrading customizations

As mentioned above, one strategy for upgrading customizations in many cases is to get rid of them, *"to go standard"*. If that is not possible the customizations have to be ported to the new version of the standard system. Below, we first discuss various reasons that make upgrades of customizations cumbersome. Thereafter we report different strategies for an upgrade.

3.4.5.1. Difficulties when upgrading customizations

Intrinsic difficulties

- Many custom features require related modifications of a number of objects or code units. The related modifications have to be found, their interaction needs to be understood. Depending on the changes in the new standard version, the whole set of modifications may have to be redesigned.

- Some objects and code units are not only modified to implement one feature but tend to be subject to a number of modifications related to different custom features. For example, the postings to the general ledger are often modified to mirror specific business rules. Here each modification has to be identified, analyzed and re-applied or redesigned.

Beginners mistakes One of the survey questions addressed the problem of mistakes made by junior ERP programmers that have little experience. Table 3.5 summarizes the answers. The issue was brought up in the interviews as well.

- The most critical issue is *undocumented customizations*. Our interview partners mostly complained about customizations made by other consultancies. As the documentation is not available, they sometimes have to perform a serious re-engineering job.
- *Poorly structured adaptations* are often due to the lack of experience of junior developers. Our interview partners mentioned reports where the database is accessed and complex algorithms are implemented instead of calling the standard function that implements that functionality. Another typical *faux pas* is to code directly into the standard functionality instead of developing an independent object and only calling the additional functionality.
- Changes to the core/system layer of Dynamics AX are considered dangerous and risky.

Difficulty levels Different kind of changes to the standard system are of course the reason for the upgrade. However, our interview partners distinguished between different levels of difficulties depending on what kind of changes conflicted with their upgrades.

- Changes to the data model that a custom feature relies on, require a total redesign of the feature.
- When business logic has changed, the new business logic has to be inspected. Either it replaces the customization and can be used as it is, or the old functionality has to be re-implemented, as part of the customization.
- The layout of forms and reports is often tailored to company specific standards. If the standard of the forms and reports module changes, this code has to be re-adjusted, often at the customer's site.

3.4.5.2. Upgrade practices

Strategy One of our interview partners explained his strategy when upgrading customizations for Dynamics AX. He takes a combined strategy; he begins with going through the logical layers of the system. First, he ports changes to the database

scheme. Then, he is doing the same for the business logic, the classes or business objects. And finally, he upgrades the customizations to reports and forms. When he gets to a modification belonging to a more complex customization, he switches to another mode: "And, of course, sometimes you have to stop with a feature, if you can't make the upgrade the easy way [...] Then you have to understand what the purpose of the feature is. Then you have to focus on: this feature links to this class and this class, [you need] to have a whole overview." [Albert] When available, the documentation of the old customization is accessed.

Knowledge Knowledge about major changes in the new version informs the upgrade process: "For instance, if you are upgrading from 3.0 to 5.0. There are a lot of changes in the project module, and if you have some customization there, you nearly have to start over with customization. Because all the data model is changed, the class hierarchy is changed. Then you have to understand what was the meaning of the old customization. Then you have to, in some way, make it anew." [Albert] All interview partners talked about the difficulties of understanding old complex modifications.

Verticals The "verticals" were not mentioned in the discussion of the upgrade difficulties. They seem not to provide major problems for the upgrade. Probably this is because normally the language does not change. Of course the parts interfacing with the standard system have to be upgraded.

3.4.6. Quality control

All of our interview partners emphasized the importance of documentation and conventions for code and documentation of the code. We specifically asked about testing, and about the tools they used both for customizations and upgrade, beyond the development environment. All interview partners mention badly documented and bad code as one of the main reason why upgrades can be difficult.

Coding conventions All our interview partners agreed that it is part of good practice to separate one's own code as much as possible from the code of the standard system. That means for Dynamics AX that additional functionality is implemented in

Table 3.5.: MOST IMPORTANT FACTORS THAT COMPLICATES AN UPGRADE

	Adaptations poorly structured	*Changes to the system layers*	*Missing documentation about existing customizations*
Minor	2	11	3
Moderate	29	23	16
Critical	11	8	23

additional classes. The interface to the standard system is done via call of standard functionality or through small insertions, where the new code has to be called from the standard functionality. The disciplined usage of the layer system also helps to separate customizations. That way, "verticals" can be further customized, as they are often sold to more than one customer.

Documentation conventions All companies used conventions for documenting code: normally the name of the developer, a date, some reference to the relevant requirement or feature in the analysis document. Two of the companies we interviewed have conventions for the overall documentation: they require that the requirements specification or the analysis report is annotated with the parts (tables, business objects and classes, code units, forms or reports) that are used to implement each required feature.

Tools The development environment was often enhanced with custom tools developed by the company itself or purchased. In one of the video sessions, a developer describes such an internally developed documentation system: each requirement is split up in a number of "programs"; for each such "programs" additional text and figures describing the design can be added; the customized and the new developed "application objects" – e.g. classes, tables, forms – are linked in; a number of test cases are defined (both program driven and manual ones).

History add-on One of our interview partners reported about a similar add-on: for a given customer, the whole communication history around the customization is accessible for the consultants and developers as well as – through a web interface – for the customer.

Upgrade tools We asked specifically for upgrade tools. A tool highlighting differences between a customized version and an standard version was mentioned as the main tool for upgrades. The documentation with the links to changed objects was only used when more complex features required additional analysis.

Linking features and modifications One of our interview partners asked for a tool that connected the different changes belonging to one feature: "That is one of the problems, because if we had some kind of tool saying that for doing this task, I make this and this and this modification. Then I could more or less isolate what has been done previously. Because one modification could take some tables, it could take some forms, it could take a code unit, and some data ports. And then you do not know how it is linked together. [Of course] we have the version tool, but it is not done properly." [Herbert]

Table 3.6.: TESTING (MULTIPLE ANSWERS ALLOWED)

What kind of tests are used to ensure that the software is correct after an upgrade?	
37	Running the GUI and performing sanity checks
26	Running in parallel with the old system
24	Unit tests
14	Extensive regression test suites
0	Machine generated test cases
3	Other

Testing Rigorous testing depends on the willingness of the customer to pay for the extra effort. Testing is not always prioritized by the customers. The standard system is taken as tested and correct with the exception of known bugs. Another difficulty mentioned in the interviews is that it is often difficult to test changes to the standard system in isolation.

System level tests Tests are normally done on a system level. That means, the feature is tested through the respective forms, and the tester eventually checks the database tables through a dedicated viewer to see whether the data is saved in the right way in the right place. The test cases are often part of the documentation.

Acceptance tests Customers check the features that are subject to the contract in an acceptance test. Normally, errors surfacing during the first few weeks of use are corrected by the consultancies without extra fees. Figure 3.6 shows the distribution of answers to the question regarding the testing of upgrades. The results confirm the analysis based on the interviews.

3.4.7. Peer learning and knowledge sharing

Developing software based on an existing large software product is often referred to by practitioners as a very different experience from writing traditional software. One must build on top of a large code base – so large that one cannot hope to comprehend it fully. The newly developed features have to be integrated with the existing code base. So how do new-comers acquire the necessary skills and the comprehensive knowledge about the software? Our interview partners reported a number of measures used by their companies to help both to develop and maintain the expertise of their employees.

How does expertise show? Expertise shows in the overview a developer has of the base system. Such an overview allows him for example to make use of functionality provided by the standard system instead of accessing the database to collect the data to be presented in various reports. It allows him to fit customizations smoothly

into the standard application. You have to know "Where to do [something] and the consequences exactly of what you are doing." [Herbert] The encapsulation of as much as possible of the implementation in own classes or objects is another indication of experience. The more the business logic can be isolated the easier it is to debug and also to upgrade customizations.

A new release is a challenge Expertise, and with it peer recognition, depends on the knowledge of the base system. Hence, on one hand, new releases are considered problematic by the developers since part of their knowledge become obsolete. On the other hand, a new version is welcomed as a challenge, providing new possibilities, new smart ways to do things.

Apprenticeship A more comprehensive project requires customizations of different levels of difficulty. Such a project offers possibilities for educating new developers "on the job". The lead developer on a project supervises new developers, starting with documenting customizations and doing simple changes, e.g. developing customized reports. As visible in the introduction of this section, this peer education continues even when the developer is able to take on customization tasks independently. Knowledge about how to solve more intricate problems is acquired by asking colleagues, and applying their hints in a trial and error fashion.

Knowledge sharing Peer education continues even among well-experienced developers. Often experts for specific modules are consulted when these modules have to be adjusted as part of a more complex customization. In one of the companies subject to the study, developers assigned themselves to investigate different modules of the new release in order to act as expert support for their colleagues. Additionally, two of the companies subject to our interviews organized more formal seminars, where topics of interest are presented and discussed.

3.5. Topics for discussion

In this section, we take up some aspects of the development practices observed and discussed in the interviews. The purpose of this discussion is to highlight some issues worth further discussion. When comparing customization of ERP systems with custom software development, a number of challenges become visible. What on a superficial glance can be seen as idiosyncrasies can actually be related to the very characteristics of adapting an existing product.

3.5.1. A different kind of development

Any programming task is dependent on the implementation technique, for example in the form of a programming language. Nonetheless, many of the most influential methods and approaches like structured programming, stepwise refinement and the

popular versions of the waterfall model, take for granted that the development team has full control over the central model, the architectural structure, and the code base of the core system.

Structured design A clearly structured design, good interface specifications, a complete set of test cases belong from this point of view to the cornerstones of good practices. In this perspective, black box is the favored approach to re-use. The aim is to encapsulate functionality so that it only interferes with the existing design in a limited way. These methods and principles contributed much to the ability to manage the complexity of software development. However, their underlying assumptions have to be reconsidered in the context of ERP customization: they might be valid when designing independent "verticals" where a developer designs new functionality. For small and medium size customizations the developer tend to in-line many small code patches to the existing code base.

Anticipation Because the use of frameworks and libraries can be anticipated, documentation supporting the anticipated use and configuration is possible. Research on documentation of frameworks and libraries like [112] emphasizes the anticipated usage. As ERP system customizations cannot be anticipated, documentation to support the customizations cannot easily be devised. Product specific research is necessary to understand typical customization scenarios and develop relevant documentation.

ERP systems The implementation of a framework or library is meant to be hidden from the developer using it. This is not possible when customizing ERP systems: the configurations are changes to the very implementation of the code. Customizations require an understanding of the existing program in order to be able to estimate the implications of a change both regarding the functionality of the program and regarding the domain the program should support. Accordingly, knowing the respective ERP system is one of the central skills of an experienced developer. One could argue that the difference between software development "from scratch", the use of frameworks and the customization is not a fundamental one. Our observations indicate that the degree of constraint the ERP system provides for the development results in radically different requirements both to the skills practices of the developers and the tool support for these practices.

3.5.2. Implications on Testing

As customizations are not always changes to isolated parts of the ERP system, testing also becomes problematic. Systematic testing of the customization would have to be based on a set of test cases for the base system. It would be a major endeavor to define such a test base: the functionality of an ERP system depends not only on the input but also on the configuration, the base data and the production data in the database. This results in some systematic challenges:

- Running such a test base, that would cover a large number of different configurations and possible combinations of production data and base data would not be possible as part of the incremental development we observed. Maybe a full regression test would require more than a night or a week end.
- To help that situation, one could try to select a relevant subset of test cases. Static analysis of the source code is probably not enough to identify this subset, since meta data play an important role in defining the behavior of the system.
- Parallel to the customizations of the source code, the relevant test cases would have to be customized. How to navigate the test base in order to identify the test cases to be changed with a customization? It is not clear whether the support provided by the test cases would compensate for the complexity they add to the implementation task.

These difficulties are mirrored in the reports on testing and quality assurance provided in the interviews. Testing is regarded as cumbersome and expensive. Test automation, e.g. in form of unit tests, is only fully applicable for the so-called verticals. The quality of the validation thus depends on the requirements of the customer and the customer's willingness to pay for the time more systematic testing requires in this environment. Instead of simply applying verification and validation methods developed for custom development where the software engineer has full control over the design or the software, the respective methods have to be adapted to the specific challenges of ERP customizations.

3.5.3. Development groups as communities of practice

The preferred way of working seems to be based on exploration and experimentation rather than on reading documentation. This way of working is enabled by the flexibility of the development environment. The knowledge the developers gather that way is shared informally. Team work is important and is taken care for in the manning of the projects.

Communities of practice A group of developers can be described as *community of practice* [207]. Informal knowledge sharing and peer learning is the main way of developing skills. Throughout all our field material, this side of the development practices was emphasized. All of the companies we interviewed were actively addressing the sharing of knowledge:

- Employees took on the exploration of specific modules of new versions.
- Newbies and skilled developers teamed up in projects. That way informal apprenticeship relations were established.
- Informal knowledge sharing was accepted and encouraged.

- Skilled developers implemented and documented their customizations according to accepted standards thus supporting maintenance and upgrades as much as possible.
- Custom tools support the commonly accepted standards of good development.

With respect to "normal" software development, knowledge and knowledge management is emphasized as well [55]. However, in the reported cases the emphasis is on method related knowledge, specific technologies, and the maintenance of project specific knowledge. Though all of our interview partners also emphasized the importance to support the development of project specific knowledge – through the organization of the projects and through adequate documentation – the knowledge that was emphasized as most important was the knowledge about the functionality and implementation of the ERP system.

3.5.4. Making customizations first order inhabitants in the development environment

The implementation of one feature or requirement often requires changes to different parts of the ERP system. Changes to the same class or code unit might belong to the implementation of very independent features. Many of the documentation guidelines and tools addressed this problem: how to keep track of a set of logically related changes, made to different parts of the ERP system.

Unit of work The programming environments use structures defined through the programming language as the unit of work. However, the unit of work when customizing ERP systems cuts across the structures provided by the programming languages. One solution to this problem would be to allow the developer to indicate that different changes belong the the implementation of one feature and highlighting them in a unique way when one of the code change is selected. Maybe, aspect-oriented techniques can be adapted to cluster related code changes and provide more control e.g. regarding type safe adaptations [182]. The development of suitable support for the developers – both regarding functionality of the development environment and the interaction with it – requires more detailed observational studies of different customization practices.

3.5.5. Supporting practices of artful integration

Artful integration In the area of computer supported cooperative work, *tinkering* and *bricolage* were recognized as organizationally viable ways to deal with heterogeneous technical infrastructures [33, 170]. In 1994, Lucy Suchman proposed, based on a rich body of empirical research around the use of document handling systems, to conceptualize design as *artful integration* rather than continue "designing from nowhere". Designers have to relate the new piece of technology to the existing technical and social context. Whereas traditional design approaches teach and develop

methods that disregard existing infrastructures and work practices [192]. The very existence of the practices we describe supports Suchman's proposal. ERP systems need to be adjusted to be useful, and that to an extent that a larger and larger number of consultancies can make a living out of it.

Skillfull integration The necessity to develop "within" a given application requires skillful integration between new and standard functionality. At the same time shortcomings of the tools and techniques developed for "normal" software engineering become visible: the development environment focuses on programming language features as the unit of work rather than clustering and navigating modifications belonging to the same feature. Verification and validation techniques are another issue where problems with traditional approaches become visible in our study. Taking the above described practices serious might lead to a more artful integration of software development methods in software development practices.

3.6. Conclusions

To better understand and support the customization and upgrade of ERP systems, we implemented an empirical study. What we found were development practices far from what is taught as good practices in software engineering. However they mirror the conditions of this kind of development. Developers have to adjust to an existing code base and design. In the discussion, we propose to take these practices as a challenge to adjust software engineering tools and techniques to support these practices. We give concrete indications how some improvements can be implemented. Software development does not necessarily take place the way developers and teachers of techniques and tools anticipate. The findings here refer to one of many development practices that do not fit the ideal promoted. End user tailoring, end user software engineering, game modding, opportunistic software development are terms denoting related practices. These diverging development practices are becoming more and more important. They will need suitable support. Adapting and integrating software engineering tools and techniques with these practices might result in more flexible and therefore more usable support for "normal" software development as well.

Acknowledgments

Thanks to all ERP system developers that offered us their time answering the survey and our interview questions. The Danish Council for Strategic Research and Microsoft Dynamics sponsored the research in the context of a Research project on *Evolvable Software Products*. Thanks to the colleagues that helped us with their generous feedback.

4

Customizable and upgradable enterprise systems without the crystal ball assumption

Sebastien Vaucouleur
IT University of Copenhagen
vaucouleur@itu.dk

Accepted for publication: IEEE International Conference on Enterprise Computing, EDOC 2009 (main conference) [201].

Abstract

Most software engineering techniques that deal with software products customization are based on *anticipation*: The software designer has to foresee, somehow, the future needs for customization so that other programmers can adapt the software product with as little modifications as possible (programmers hide implementation details behind previously defined interfaces, or alternatively, they refine some pre-defined properties). While practical, this approach is unfortunately not completely satisfactory for *Enterprise Resource Planning* systems (ERPs). These software products have to be customizable for numerous and various local contexts; they cover a very large domain, one that cannot be fully comprehended — hence accurate anticipation is difficult. To solve this problem, an extreme measure is to give the programmers the means to do modifications in place, directly in the source code. This approach trades control for flexibility. Unfortunately, it also makes the customized software product very sensitive to upgrades. We propose a more mitigated solution, that does not require accurate anticipation and yet offers some resilience to the evolution of the base software product through the use of code quantification.

We introduce the Eggther framework for customization of evolvable software products in general and ERP systems in particular. Our approach is based on the concept of code query by example. The technology being developed is based on an

initial empirical study on practices around ERP systems. We motivate our design choices based on those empirical results, and we show how the proposed solution helps with respect to the upgrade problem.

4.1. Introduction

4.1.1. Enterprise resource planning systems

First, we will recall some details about *Enterprise Resource Planning Systems systems* (ERPs), but only those that are directly relevant for the discussion (we refer the reader to Shanks et al. [183] for a detailed treatment). ERP systems are usually defined as being business support systems, that deal with the management of the various functions found in modern companies, such as manufacturing, financial, human resources and customer relationship management. ERP systems are data-oriented: the back-end database is typically seen as the central element of the infrastructure (while there is a trend to give better support for processes, ERP systems remains heavily data-oriented). When a company purchases a license, it can decide to use an ERP system as it is — the ERP system is fully functional and can be used off-the-shelf. Nonetheless, many companies prefer to customize their newly acquired ERP system to the local context in which it will be deployed. The local context can be for example related to the company's unique business model, or to some local regulations: state regulation, industry specific regulations, etc. An alternative is to adapt the company to the ERP system — which does happen in practice.

Customizations can be made directly by customers, but usually they are done by small software houses that specialize in this activity. The competencies of these software houses consist in their knowledge of the ERP system, but most importantly in their knowledge of the vertical domains (accountancy, transport industry, etc.). Their mastery of both the ERP system and of the vertical domains allow them to quickly develop customizations that fit the needs of their customers. They typically charge high fees for their services, hence time-to-market is important to the customers.

Our work targets evolvable software products in general, and ERP systems in particular. We grounded our work in the study of two existing ERP systems, Microsoft Dynamics AX and Microsoft Dynamics NAV, that we will call collectively Microsoft Dynamics. Section 4.2 provides a summary of the empirical study. We refer the reader to [182, 190, 88] for a more complete treatment about Microsoft Dynamics. Note that our work does not directly address database evolution, another common problem in the field of enterprise systems. Finally, we do not address disruptive organizational changes required by the adoption of an ERP system, commonly referred to as the *misalignment problem* [205].

4.1.2. Definitions

Customizations Customizations add some new and un-foreseen features to a software system. We contrast customizations with *configurations* (configurations enable

or disable features already present in the program).

Software products A software product is software that can be customized for a specific context. Successful software products are evolving on a regular basis [182].

Base software product The base software product is the software product *before* customizations.

Software product maker The software product maker is the company that designs and implements the base software product. In the case of Dynamics, Microsoft is the software product maker.

Partners Partners are software houses that specialize in making customizations for other companies (their customers could be partners themselves or regular customers). The term *partner* recalls the privileged business relationship that they have with the software product maker[1].

Customers Customers will simply refer to the companies which are purchasing an ERP system for their own needs. Typically, customers are also purchasing customizations services or customization code from partners.

Microsoft Dynamics follows the following business model: Microsoft sells the base software products, the ERP. Partner companies specialize in providing some customization to the original ERP systems. Although many partner companies provide these services, users can decide to use the software product as it is, or to perform the customization themselves if they have the competencies in-house (a large part of the source code is provided). Partners can also provide customization solutions to some other partners, who can in-turn perform some further customizations. Using this unstructured scheme, Microsoft can scale the scope of their product to a very large market around the globe, and yet can keep its focus on its main competency: the core horizontal functionalities of the ERP system (for example the transactional sub-system, a convenient graphical interface, web services, etc.). On the other hand, reasoning about the intent behind a specific part of the system can be particularly challenging — especially since the code base can be very large, more than l million lines of code, with no explicit specification (modulo some informal documentation).

4.1.3. The Crystal ball assumption

At the core of many approaches to evolution and customization within the field of software engineering is, in one form or another, what we call the *crystal ball assumption*:

[1]The terminology around Microsoft Dynamics sometimes makes a distinction between companies that implement customization solutions for a particular vertical domain, and companies that make customizations for a single company. For the sake of simplicity, we will ignore this distinction in this paper, and use the generic name of *partner*.

software designers are supposed to be equipped with a rather accurate *crystal ball*, by which they can anticipate the future needs for customizations and evolution of their software product. A concrete example in object-oriented programming is the use of virtual methods and factory patterns that are positioned in strategic positions of the code base to deal with *likely* variations. The same need for anticipation (the need for a "crystal ball") can be found in most approaches, whether object-oriented or not [182].

We would like to argue that the problem is particularly prominent in the field of ERP systems: the domain covered by ERP systems is very large and diverse. For example, tax rules are obviously not the same in Denmark as in France. But what about this particular region of France; what about the rules for a particular industry (say textile); what about the rule for this particular time of the year, and what about the combination of any of these special cases? Not only the domain is very large, it is also evolving very quickly: tax rules typically come and go as new governments are put in place. In short, the domain is very large, and evolving — full anticipation is not an option.

ERP systems can be contrasted with software products that deal with more stable domains. Consider for example graph libraries: graphs have been thoroughly studied and the concepts and design alternatives around graphs are well comprehended, extensively explored and have been well documented. Of course new important variations around graphs do surface once in a while — but it is relatively rare. Typically, that new variation would simply be incorporated in the next version of the library: what is required in this case is not code customization but *code evolution*.

4.1.4. Anticipation is not a panacea

Several lines of research have tried to address explicitly the anticipation problem. For example, the work on *Multi-Dimensional Separation of Concerns* by Tarr et al. at IBM Research lead to Hyper/J [156, 182], an ambitious framework for multiple decomposition of existing software products. Quoting Ossher and Tarr:

> "Anticipation causes ulcers : Deeply ingrained within software engineering is the notion of anticipating and designing for the *most likely* kinds of changes, towards the goal of limiting the impact of future evolution. [...] We believe in anticipating and planning for changes whenever possible. Anticipation is not, however, a panacea for evolution. It clearly is not possible to anticipate all major evolutionary directions. Further, even if it were possible, building in evolutionary flexibility always comes at a price: it increases development cost, increases software complexity, reduces performance, or often all of the above." [156].

The stand of Ossher and Tarr is very close to ours: anticipation is definitively useful, but it is not a panacea. As noticed in the previous section, we believe that the problem becomes prominent in the field of ERP systems: the domain that they cover is very large and is constantly evolving, hence the software maker cannot *comprehend* it. Even if he could, finding a convenient and useful safe approximation (an abstraction) of

all those local variations is likely to be difficult. We will show how we address the anticipation problem in section 4.3.

4.1.5. The upgrade problem

We now present the upgrade problem. We focus on code upgrade; data migration is another important problem that we do not discuss here. ERP systems are being customized by numerous independent partners. Eventually, the software product maker will release a new version of the product. Figure 4.1 is an abstract and simplified representation of the upgrade problem. Nodes represent a variant of a software product. A directed edge from a node X towards a node Y denotes that Y is an evolution or a customization of X. Horizontal edges denote evolutions, and vertical edges denote customizations. A particular version of a software product P_y^x can be customized by a partner, leading to the software product P_{y+1}^x. This software product can be itself further customized by another partner, leading to P_{y+2}^x, etc. On the other dimension, P_y^x will eventually evolve to a new version P_y^{x+1}, forcing the partner that produced the software product P_{y+1}^x to adapt its customizations in order to come up with P_{y+1}^{x+1}.

Imagine that a partner wants to implement a feature called F as a customization. To do this, he identifies a particular code fragment C whose behavior needs to be modified. In the next version of the software product several problems can surface when the customization (made the previous version) need to be ported to the latest version of the base product. First, C might have been modified, for example a loop was rewritten to use recursion, or C was moved to an other location in the code base. One can also consider the extreme case: the code fragment C might have disappeared all together in the new version. Or maybe the feature F is now provided by default by the software product maker, in which case the customization should not be ported to the new version of the software products but should be simply discarded. Yet another problem can surface: in the new version of the software product, C was un-touched, but in addition to C another code fragment now needs to be modified to support the partner's intent.

Road-map

Section 4.2 will give a summary of the empirical study, focusing on the most relevant results for the discussion. Section 4.3 will introduce the Eggther framework for customization of software products, and will carefully motivate the design choices based on the results from the previous sections (section 4.3 also makes the connection with Aspect Oriented Programming). Section 4.4 will propose to use two well-known measures to quantify exactness and completeness in the context of our framework. Important implementation details are addressed in section 4.5. Section 4.6 describes some future work. Some of the most frequent questions around this work are discussed in section 4.7; the same section will briefly point to related work. Finally section 4.8 will conclude.

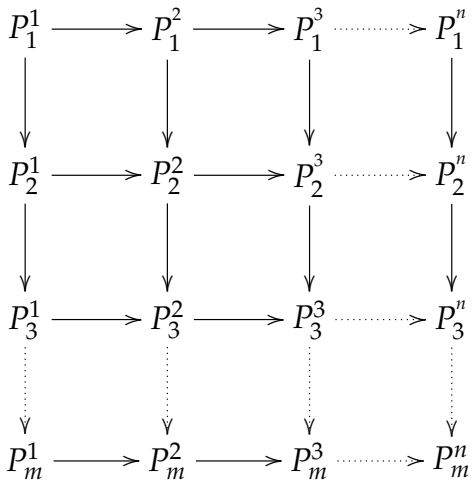


FIGURE 4.1.: THE UPGRADE PROBLEM

4.2. Empirical grounds

Our work on software products is grounded on an initial empirical and qualitative study that focused on customizations and upgrades of Microsoft Dynamics ERP Systems [57, 58]. We shortly summarize here the most relevant results for the discussion.

Cost of the upgrade problem It is difficult to precisely measure the cost of the upgrade problem described in section 4.1.5. Our empirical survey points to a range between 10 and 15% of the price of the original customization for a single upgrade. These numbers are derived from informal discussions with ERP practitioners and were not collected using a rigorous statistical approach. Upgrades have to be operated on each deployed project. Customers typically consider that upgrades are almost mandatory: they fear that they will not benefit from the latest bug fixes if they do not upgrade. They also avoid to jump a version because upgrading will be harder later on. Upgrades happen approximately every two years.

Partners are domain-experts When recruiting new staff members, partners tend to favor domain experts to highly skilled software programmers. Ideally, they try to form groups of two persons, who work together on the same customization project: one is the domain expert (for example an accountant), and the other one is a more technically minded person. Typically, even the "technically minded" staff members have little formal computing science education: they either learned programming through internal company training, or are autodidacts. Staff members share their experience with their co-workers through informal discussions and pair programming.

Partners rely extensively on code examples Staff members working for partners acquire their knowledge about the ERP system using code examples. They look at the existing application code (given by the software product maker), or to some code previously written by a colleague, and imitate the practices. When it is available, documentation is of course used from time to time, but we would like to stress that knowledge is mainly acquired by looking at code examples.

Mainly incremental Customizations are described by the partners as being "mainly incremental": partners typically avoid to remove functionalities, and prefer to simply hide unused elements at the graphical user interface level. It is commonly perceived as "adding functionality" or "adding features". Note that the terminology used by partners is informal: their customizations do have side-effects and impact the semantics of the existing code base (obviously a customization that has no impact is of little value).

Exit points Customizations are done by inserting exit points to some hook methods. That is, they avoid to the extent possible to insert a lot of code in the existing code base and simply insert external calls to their newly defined methods. This offers a form of textual modularization, albeit not a perfect one since these method-calls are intrusive.

4.3. Eggther framework

This section introduces the Eggther framework for customization of software products. We show that our approach (a) Allows for non-anticipated customizations, and (b) Provides some resilience to upgrades. The work is grounded in the empirical work described in section 4.2, and builds on the .NET framework [65].

4.3.1. Overview of the approach

We summarize the approach, and we will come back in details on each step in the rest of this section.

First, the software product maker designs his software product using the dominant decomposition mechanism in object-oriented programming: data decomposition. This decomposition is based on his knowledge of the vertical domains, and on his capacity to anticipate future needs for customizations, see section 4.1.3.

Second, the software product maker gives to the partners part of the source code. The first difference with the scheme described in section 4.1 is that partners only have read-access to the source code: we say that this is a glass-box approach to code reuse. If the decomposition of the software product fits their needs, the partner just use the traditional object-oriented extensibility mechanisms (sub-classing, method redefinition, etc.); if the decomposition is not convenient, the partners write code queries to denote the variability points, see section 4.3.2.

The partner then writes customization code. The binding between the variability points and the customization code is described in section 4.3.3. Customizations can be further customized by another partner. Upon upgrade, queries are re-applied, and eventually modified. Customizations code is eventually modified.

4.3.2. Code queries

We now enter the core of the problem. We mainly deal here with the limitations posed by the crystal ball assumption and how we can approach these limitations in a way that allows us at the same time to deal with the upgrade problem.

From white-box to glass-box As noticed previously, a white-box approach is not satisfactory since this will make upgrades difficult. A pure black-box approach is not satisfactory neither since it requires anticipation. Therefore, we move from a white-box to a glass-box approach: partners are given part of the source code, but with a read-only access. The first step for them is to denote the variability points, the locations in the code where customizations should happen.

From extensional to intensional definitions The programmer needs to define the set of code fragments that have to be customized. We distinguish two ways to define sets: *by extension* and by *intension*. An extensional definition would have the form

$$CodeFragments = \{c_1, c_2, c_3, ...\}$$

whereas an intensional definition would have the form

$$CodeFragments = \{c \mid P(c)\}$$

where P defines a property of the source code fragment — this is also known as *set comprehension*. Using our framework, the variation points will be described, to the extent possible, by *intension* rather than by *extension*. This will allow the denotations to be more resilient to evolution of the base code because the variation points are not mentioned explicitly.

One way to express variation points by intension is to use a reflective language with a set of user-defined and pre-defined relations, that could include for example an implementation of the predicates

$$IsMemberPublic : Member \rightarrow \mathbb{B}$$

$$IsMemberName : Member \times String \rightarrow \mathbb{B}$$

Using set comprehensions, the partners could then express that all public members called “F” should considered as variation points:

$$VariationPoints = \{x \in Method \mid IsMemberPublic(x) \wedge MemberName(x, "F")\}$$

This is a perfectly valid approach and it also maps almost directly to a rule-based engine, such as Prolog. Unfortunately, it requires partners to think at a higher level of abstraction, at a meta-level. This conflicts with the fact that partners are not computing science specialists: they are domain experts, see section 4.2. We want to give to partners simple and yet convenient programming primitives: primitives that are close to how they approach their daily programming tasks.

A simple programming primitive: CQE As observed in our empirical study, programmers like to work with code examples. This is how they work when they face their daily programming tasks: this approach seems natural and appealing to them. Hence, we propose to use the concept of query-by-example to denote variability points. Query-by-example is a well-known concept in the field of database systems, we adapt it here to the domain of code query: what partners are effectively querying is the code base of part of the software product.

Query methods .NET attributes can be used to annotate class members [65, 66]. Class members marked with the `[Query]` attribute will denote code queries. In the case of methods, we will simply call them *query methods*. Several query methods can participate in the same query as we will demonstrate. For a given query, the framework will look for matching code fragments in the existing software product. The matching code fragments will be the variability points. The framework will insert method invocations to the customization code at the variability points (we will describe customizations in section 4.3.3).

Formal arguments of query methods Variables available at the customization points will be bound to formal arguments of the query methods. Query methods formal arguments of type delegate [65] are used to denote an arbitrarily large well-formed code fragment whose static type is the return type of the delegate: for example, `Func<int>` denotes an arbitrarily large expression of static type int. A delegate with a `void` return type denotes arbitrarily large well-formed sequence of instructions.

An example In the following example, a partner wants to perform a customization in all the code locations where a transaction is performed. Using a code query example, the partner describes his concept of a transaction as a BankAccount being debited from a given amount, some further action taking place, and a BankAccount being credited, within the same procedure. In the following listing, the formal argument `action` is of static type `Action` (a delegate with a `void` return type), hence any arbitrary large sequence of instructions would match the delegate call, line 8. (Note that all example compiles with a standard C# compiler.) The query is given a name, here "Transaction":

```
1 [Query("Transaction")]
2 void SimpleTransaction(double amount,
3                         BankAccount b1,
4                         BankAccount b2,
5                         Action action)
6 {
7   b1.Debit(amount);
8   action();
9   b2.Credit(amount);
10 }
```

For example, the query "Transaction" would match the following code fragment from line 5 to line 7:

```
1 void F(double x,
2        BankAccount a,
3        BankAccount b,
4        double tax) {
5    a.Debit(x);
6    x -= tax;
7    b.Credit(x);
8 }
```

Note that here the identifier `x` is not required to be bound to the same value for the debit and credit operations: In this case, the action applies a tax, and rebinds `x` to a new value. Note also that nothing prevents the identifiers `a` and `b` to be bound to the same object (aliasing). The action can refer to an empty action, hence the following code fragment would match our first code query:

```
a.Debit(x);
b.Credit(x);
```

Disjunction of query methods Continuing on the same example, if the partner wants to cover the case where a Credit operation is done first, then a new query method must be added, see listing 4.1. Note that both query methods below are given the same query name, hence both of them participate in the definition of the query "Transaction". Informally, the code query can be interpreted as the disjunction of two cases: code fragments of the form `DebitFirst` *or* code fragments of the form `CreditFirst`.

Summary of code-queries To summarize, we moved from a white-box to a glass-box approach ("see but don't touch"), by adding a level of indirection: code queries. The query language is based on the concept of query-by-example. This simple programming primitive makes the approach accessible to partners (non-programmers experts), as they do not have to think at a meta-level. The code queries are a form of code quantification. This code quantification allows us to break procedural abstraction and hence to deal with fine-grain unanticipated customizations. Quantification allows

LISTING 4.1: DISJUNCTION OF QUERY METHODS

```
[Query("Transaction")]
void DebitFirst(double amount,
                BankAccount b1,
                BankAccount b2,
                Action action)
{
  b1.Debit(amount);
  action();
  b2.Credit(amount);
}

[Query("Transaction")]
void CreditFirst(double amount,
                 BankAccount b1,
                 BankAccount b2,
                 Action action)
{
  b1.Credit(amount);
  action();
  b2.Debit(amount);
}
```

us to textually localize the definition of the variability points outside of the base code of the software product.

4.3.3. Customization code

Once the variability points are defined using code queries, the actual customization code can be expressed using a regular .NET language, such as C#. Methods annotated with the attribute `[Customization]` are called *customization methods*. Classes that contain customization methods are called customization classes. The customization attribute takes as an argument the name of the code query that denotes a set of variability points that should be customized.

Example of customization code Continuing on the transaction example, now that the variability points are defined, the partner wants to log all transactions before they take place, see listing 4.2.

Binding Formal arguments of customization methods are bound to available identifiers at the scope of the variability point. The framework builds a sequence of available identifiers within the scope of the variability point $\langle i_1, i_2, i_3, ... \rangle$ respectively of static types $\langle I_1, I_2, I_3, ... \rangle$. The signature of a customization method M defines a sequence of formal arguments $\langle f_1, f_2, f_3, ... \rangle$ respectively of types $\langle F_1, F_2, F_3, ... \rangle$. For

LISTING 4.2: EXAMPLE OF CUSTOMIZATION

```
[Customization("Transaction")]
public void LogTransaction(double amount,
                           BankAccount b1,
                           BankAccount b2) {
  Log("Transaction from account {0}
      to account {1}, amount {3}",
      b1.Number,
      b2.Number,
      amount);
}
```

each formal argument f_n the framework looks sequentially in the sequence of available identifiers for an identifier i_m, such that i_m was not already bound to a formal parameter of M, and such that $I_m <: F_n$, where $<:$ denotes the usual subtyping relation[2]. If the framework cannot find such an identifier the customization method will not be called. Note that only a prefix of the available identifiers at the scope of the variability point is necessary in the signature of the customization method. For example, if the customization would simply need to log the amount of the transaction, the following method signature would be sufficient:

```
[Customization("Transaction")]
public void LogTransaction(double amount) {
  Log("Transaction amount " + amount);
}
```

On the other hand the following customization would not be called (otherwise it would be ambiguous which BankAccount we refer to).

```
[Customization("Transaction")]
public void LogTransaction(BankAccount b) {
  Log("Transaction account " + b.Number);
}
```

The current object Sometimes, it is useful to have access to some members of the current object (the object where the customization is called). To support this, the partner can annotate a formal argument f_x of a customization method with the attribute `[Current]`, in which case the framework will bind the current object to f_x. If the variability point is in a static scope, f_x will be bound null. If the current object is not a subtype of F_x (the static type of f_x), the customization method will not be called. Note that using this attribute is optional feature, and can be safely omitted if it is not required. For example suppose that we want to log the identifier of the TransactionManager when a transaction takes place:

[2]Type compatibility is defined by the ECMA standard [65, 66]

```
[Customization("Transaction")]
public void Log(double amount,
                [Current] TransactionManager tm)
{
  Log("Transaction amount {0}
      executed by transaction manager {1}",
      amount,
      tm.ID);
}
```

Note also that we do not break encapsulation here, since the property Number of the class TransactionManager should have public access, or at least the member should be accessible from the customization class.

Before versus after customizations By default customizations are triggered just before the matched code fragments execute. If partners want a customization to be called just after the variation points, he can simply set to true the `After` property on the `[Customization]` attribute. For example the following will log the balance of the debited account after the transaction has taken place:

```
[Customization("Transaction", After = true)]
public void LogDebitedAccount(double amount,
                              BankAccount b) {
   Log("Balance after transaction: {0},
       b.Balance);
}
```

Side effects in code customizations So far our customizations mainly added behavior. This corresponds to a great part of customizations tasks performed by partners, see section 4.2. Given the techniques that we already covered, a partner can already introduce side effects in customization methods, for example by writing `b.Debit(1)` in a customization method.

Nonetheless, it is useful to be able to modify the binding of identifiers within the scope of the variability points. To do this, partners can annotate the formal arguments of customization methods with the standard `ref` modifier. The value of a reference parameter is the same as the argument in the method member invocation [65, 66] (they represent the same storage location). The framework will take care to bind the variables of the matched code fragments by reference. In the following customization method, a partner wants to convert the transaction amount from Euros to Danish Kroners:

```
[Customization("Transaction")]
public void ConvertToDKK(ref double amount) {
    amount *= EuroToDKK ;
}
```

The effect of this customization is that the `amount` identifier will be rebound to an new value.

Interrupting the flow of control Note that the approach that we propose is mainly incremental: functionality is added to some specific places in the code base. This fits closely with the result of the empirical study on customization practices, see section 4.2. Nonetheless, it is possible to interrupt the flow of execution at the variability points by throwing an exception in the customization code. We take advantage of the fact that exceptions are unchecked in .NET [65]. Throwing an exception to interrupt the flow of control is, arguably, a reasonable thing to do: we want to warn callers (at run-time), up in the call-stack, that the expected execution of part of the base code did not take place. Continuing on the transaction example, a partner wants to forbid transactions of more than 1.000.000 Euros:

```
[Customization("Transaction")]
void NoLargeTransaction(double amount)
{
  if(amount > 1000000)
    throw new TransactionException();
}
```

4.3.4. Unit testing

Customization methods are directly amenable to the usual unit testing procedures. For example, a unit test for the `NoLargeTransaction` customization would look like:

```
[Test]
[ExpectedException(typeof(TransactionException))]
public void TransferWithInsufficientFunds()
{
  var tm = TransactionManager.Instance;
  var b1 = new Account(2000000);
  var b2 = new Account(0);
  tm.Transaction(b1,b2,2000000);
}
```

4.3.5. Stateful customizations

As mentioned in section 4.1, data and state are an important part of modern ERP systems, therefore it is important to support stateful customizations: the framework must allow the partners to preserve some state across several invocations of the same customization method. In our framework, customization classes can simply declare some class or instance variables that will preserve state across invocations of customization methods. The following example counts the daily number of debit operations (made as part of a transaction) for all bank accounts, and throws an exception if a threshold is crossed:

```
// Daily initialize NumberOfDebitOperations
// to 0 for all existing accounts
Dictionary<BankAccount, int> NumberDebitOp
   { get { ... } }

[Customization("Transaction")]
void CheckNumberDebitOp(double amount,
                        BankAccount account)
{
  var num = NumberOfDebitOperations[account];
  if(num > 100)
    throw new TransactionThresholdException();
  NumberOfDebitOperations[account] = num + 1;
}
```

4.3.6. Aspect-oriented programming characterization

According to Filman and Friedman aspect-oriented programming (AOP) is quantification and obliviousness [75]. The code-query by example of our framework provides quantification: the result of the code queries are the joint-points in AOP parlance. Obliviousness is achieved when the programmers should not be required to insert join-points markers into they source code. Our approach provides obliviousness since no special markers are introduced. According to this definition, our framework is AOP, and code query by example is a pointcut language. Among the most popular AOP tools, one can cite AspectJ [14], PostSharp [164], and CaesarJ [74]. Contrary to those frameworks, our pointcut language is completely embedded in a host programming language (for example, C#). Also, to the best of our knowledge, there is no other AOP tools that uses the concept of code query by example.

4.4. Exactness and completeness of code queries

Upon upgrade, existing code queries might not denote *exactly* code fragments that fit the partner's intentions since the customizations were developed for the previous version of software product. Similarly, code queries might not refer *completely* to the code fragments that should be customized to fit the partner's intention. A measure of exactness and completeness of code queries would be helpful to characterize these issues. To this extent, we introduce precision and recall.

4.4.1. Precision and recall

Precision and *recall* are two measures widely throughout science used, and especially in the field of information retrieval, to evaluate the quality of results, focusing respectively on exactness and completeness [17]. We introduce precision and recall as they provide a convenient and well-defined terminology for the rest of the discussion. The two measures are traditionally defined in terms of a set of retrieved documents, and

a set of relevant documents. Precision is the percent of retrieved documents that are relevant to the search (exactness):

$$Precision = \frac{|\{Relevant\ documents\} \cap \{Retrieved\ documents\}|}{|\{Retrieved\ documents\}|}$$

In turn, recall is the fraction of documents relevant to the query that are successfully retrieved (completeness):

$$Recall = \frac{|\{Relevant\ documents\} \cap \{Retrieved\ documents\}|}{|\{Relevant\ documents\}|}$$

One can immediately observe that a recall of 1 can be easily obtained by returning all documents in response to any query. Dually, one can very easily obtain a high precision by returning no documents to any query. Hence, it is useful to use those two measures together.

4.4.2. Precision, recall and code queries

When a partner writes a code query, he wants to denote the variability points for customization purposes. We will say that precision is high when the code query returns mostly variability points that are necessary for the customization. Dually, recall will be high when most required variability points to implement the customization are returned by the code query.

When precision is lower than 1 some customizations will happen at places where they should not happen. Similarly when recall is lower that 1, then some locations in the code base where customization should happen will not take place.

4.4.3. Exactness and completeness problems upon upgrade

Upon upgrade, a partner takes the new version of the base software product and reapplies the code queries. We simplify the discussion by considering that the customization consists of only one code query. We can immediately identify two problematic cases:

(1) The query result now points to some code fragments that *should not* be customized in the new version,

(2) The query fails to match some code fragments that *should* be customized in the new version of the base code.

From a practical point of view, the first problem is less problematic: the partner is presented with the result of the code query, and can inspect whether the new customizations points fit his intentions. If he identifies a problem, he can decide to rewrite the customization query to fit his needs. The second problem seems more difficult to tackle. We identify two sub-cases to problem (2):

(i) Some new code that *should* be customized is not part of the query result,

(ii) Some old code that *should not* be customized in the previous version *should* now be customized in the new version.

Helping with (i) can be done by simply providing a diff of the new version versus the old version to partners, and requiring them to study very carefully the changes. This is highly impractical, of course. Alas, (ii) seems even more difficult to deal with; it is not clear at this point how to approach this last problem in a scalable way in the absence of any explicit specification.

4.4.4. Giving control back to the partners

Ideally, one would like to have both high precision and high recall. Nonetheless, one can sense that there is a tension between the two: design decisions that favor precision will hamper recall and vice versa. Since these design decisions directly impact the upgrade process, it would be good if partners could regain control of them. In other words, let them decide whether exactness or completeness is more important to them. Our framework allows for this: partners can abstract their code queries by making use of delegate calls to denote typed expressions. Making extensive use of delegate calls in code queries will increase recall but decrease precision. Dually, partners can add more context code in their queries (make their code query more lengthy than strictly necessary) and will thereby increase precision but decrease recall. Similarly, partners can increase or decrease the number of formal arguments in their code query, which will respectively increase precision (but decrease recall), and increase recall (but decrease precision). Section4.6, which describe some further work, details yet another way by which partners could favor precision or recall.

4.5. Implementation aspects

We quickly describe the current implementation of the Eggther framework since the design choices have some important implication on the application of the framework.

4.5.1. General design choices

An obvious design choice for our framework would have been to reuse an existing parser for .NET language (or to generate one), and to do the code queries on the abstract syntax tree instantiated by this parser. Unfortunately this has major drawbacks: first, for each language that we would like to support we would have to implement a new variant of the framework, and every time that one of the high-level language evolve we would have to evolve our framework. Second, conceiving a complete and high-performance parser for a complex high-level language is difficult. Finally, we would loose integration with current development environments. Therefore, we decided against this approach, and instead settled to work at the intermediate code level.

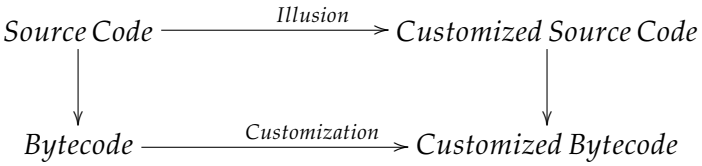


FIGURE 4.2.: CUSTOMIZATION SCHEME

Implementation details We give an outline of our current implementation:

- First, the base code is compiled to .NET intermediate code (IL) [66, 65] using one of the standard compilers for the corresponding high-level language (the standard C# compiler, the standard VB.NET compiler etc.), and the same is done with code query and the customization code.
- At design time, the framework will work at the IL level, and for each code query will look for matching IL code in the compiled base code. When a match is found the framework will instrument the bytecode by injecting a method call to a proxy method, binding the formal arguments of the proxy method to the available identifiers in the scope of the method call.
- At run-time, the framework will construct a list of the customization methods and instantiate a delegate for each them. A singleton [83] is instantiated of each customization classes. When a proxy method is called, it will look for the customization methods that subscribed to this particular customization points (the query name), and will invoke the corresponding delegates (binding of formal arguments is done as described in section 4.3).

Note that even if matching and instrumentation is done at the IL level, partners get the illusion that is performed at the same level at which they write high-level code, see section 4.2.

4.5.2. Advantages

We summarize some of the advantages of this approach:

- Our framework re-uses the standard high-level compilers, and we can rely on Microsoft and others to maintain them and evolve them as new version of the languages surface.
- The standard compilers have been well-tuned for many years and are highly performant.
- We get support out of the box not just for one high-level language for a large number of them (C#, VB.NET, Eiffel, etc.): the only requirement is that the base

software product and the code queries are written in the same language and are compiled using the same compiler.

- Partners enjoy the full development environment that is already available to them and that they know well (the interactive development environment of Visual Studio that includes a type checker, syntax highlighting, refactoring tools, etc.). We would like to stress that our use of the standards type checkers allows us to make sure at design time that the query and customization methods are well-formed.

4.5.3. User interface

The concrete user interface is an add-in for Visual-Studio that allows partners to execute the code queries and visualize the variability points in the base software product at design time.

4.6. Further work

We shortly introduce some further work that are potential extensions to the framework we introduced in section 4.3.

4.6.1. Non-Boolean matching

So far, code matching was defined as a simple predicate, taking as arguments two code fragments, and returning true or false depending on whether the two code fragments match each other. We can generalize this approach by defining a measuring function M that will give the distance between any two code fragments:

$$M : CodeFragment \times CodeFragment \rightarrow \mathbb{R}_0^+$$

We will call this distance the *code distance*. We are considering to use two well-studied algorithms as the foundation for code distance: the first is the Levenshtein distance (also called edit distance), and second, the tree-edit distance [28].

4.6.2. Partial ordering of customizations

It would be useful to be able to specify an order in which customization methods should be executed for a given variability point. A partner should be able to define a simple partial order between customizations (that includes his own customization as well as customizations from other partners), and the framework should simply compute a linearization compatible with this partial order. This seems, a priori, quite straight-forward to implement. We believe that the challenge is more on the language design side: how to allow partners to express this order relation in a convenient way, and make it at the same time seamlessly integrated with the programming primitives

that we introduced. Furthermore, it is not clear what the framework should do if it is not possible to compute a linearization given the constraints given by the partners: should it throw an exception? And if so, where and when?

4.6.3. Toward behavioral customizations

The customizations that we described were based on matching of the code structure. While it is a practical and convenient solution to many customizations scenarios, sometimes the customizations are not driven by specific code patterns but by more behavioral aspects.

Example Continuing on the previous example, consider the following customization scenario: suppose that when the balance of a bank account exceeds a threshold, the local bank branch should get an alert (for example to send to the client some commercial offers for the latest financial products). This particular scenario is more abstract than the previous ones, since here the customization is not geared toward specializing the software product at some specific places in the code text, but rather to take some action under certain conditions, irrespective of the location in the code that made the bank account cross the threshold. Moving from this informal requirement towards a more precise specification, we define predicate $BalanceTreshold : BankAccount \times \mathbb{R} \rightarrow \mathbb{B}$, which evaluate to true if given a bank account and a threshold, the balance of the bank account is greater than the threshold. Whenever the predicate first evaluates to true, the customization should be triggered, namely sending an alert to the local bank.

Possible approach Consistent with our goal of reusing existing .NET technology as much as possible we envisage to use to Spec# to our benefit. Spec# is an extension of C# that adds to the language, among other things, the concept of invariants and pre and post-conditions. Spec# is based on the foundations laid down by the work on axiomatic semantics, pioneered by Floyd, Hoare and others. The definition given by Hoare [95] is based on the concept of a triple $\{A\}B\{C\}$ that defines partial correctness: whenever A is true and B executes and terminates, then C will be true (where A and B are predicates on the state and B is a command). Similarly, but from a more concrete point of view, a method can be equipped with pre and post-conditions, where the precondition defines what has to be true at the beginning of method execution (to be satisfied by the client, i.e. a benefit to the supplier), and the post-condition what has to be satisfied by the supplier when the method terminates (to be satisfied by the supplier, i.e. a benefit to the client).

In a previous work [202], we redefined the notion of pre-condition in the concurrent case to move closer to the concept of conditional critical regions (first proposed by Hoare then championed by Brinch Hansen [90]). Similarly, we envisage to redefine the semantics of the pre-condition in the case of a customization: whenever a customization method is equipped with a pre-condition, this precondition will define a

condition for triggering the customization. More concretely, and using the concrete example introduced above, the condition is the predicate, expressed as pre-condition to the customization:

```
[Customization]
void AlertThreshold([Current] BankAccount b)
requires b.Balance > 10000;
{
  SendAlert(b.LocalBank, b);
}
```

The semantics of this customization is to be interpreted as follows[3]: for all bank accounts b that are instantiated in the runtime such that *AlertThreshold* was not already executed with b as an actual argument, whenever the balance of b is more than 10.000 Euros, then execute *AlertThreshold*.

4.7. Discussion and Related work

This section discusses informally some of the questions related to the approach that we presented, and points to some related work.

What if one is completely satisfied with the crystal ball assumption?

If anticipation can be done accurately in a way that satisfies the partners (that is, if customization points are well defined and the granularity of future customizations is well-understood), then usual customization techniques can be used: for example dynamic binding, together with a flexible software product design that makes use of a subset of the various extensibility-related design patterns : factory methods, visitor pattern, adapter etc. Interface specifications can then be made more precise using for example design-by-contract (see Spec# [18], or JML [119], etc.). On the language side, the issue of co-variance versus contra-variance will then surface, and one will favor one option or the other, making a choice between openness and type safety [1, 36]. All this is now folklore in the software engineering community: it has been well-studied and well-documented. Of course more research in this field is certainly useful (see for example the work on ownership type systems [146]), but we believe that this does not address a problem which is *characteristic* of ERP systems. Once again, we believe that the main characteristic of ERP systems is the difficulty to anticipate the future needs for customization accurately.

What about Versioning Systems?

Versioning systems bear some resemblance with our work but have a more textual approach to code evolution. Versioning systems rely on a "diff" program to show the difference between two files [121]. One can consider diff as a more general approach

[3]Notice that, in this case, the semantics of the pre-condition is close to one of a guard.

to what we have described: using diff any text files can be compared, whether it is source-code or not. By targeting only .NET languages, we can implement some useful specific functionalities: for example, our use of delegates to denote typed expressions in our code queries, or simply ignoring irrelevant differences between code queries and the base code (such as lines indentation). An alternative approach to our work could have been to conceive a special version of an existing versioning system and specialize it to deal only with C# code. We decided for different approach which makes extensive use of the existing .NET infrastructure (re-use of the existing compilers, etc.).

Is there some connection with Software Product Lines?

A part of the software engineering community now focus on a line of research called *software product lines* (SPL). The goals behind SPLs bears some similarities with our work: both allow for the customization of software systems by third-parties. Nonetheless, there is one fundamental difference: SPLs have a close-world assumptions. That is, the SPL community usually assumes that the set of possible customizations is well-known in advance, by a central agent such as chief architect [182]. ERP systems cannot rely on this assumption, see section 4.1.3. From this perspective ERP systems are not SPLs.

What seems to be, at this point, the pros and cons?

First, ease-of-use is an important aspect of this work: the programming primitives are simple; they do not force domain experts to think at a meta-level; partners can think at the level of abstraction they are used to, using for a great part the same high-level language that they already know – They do not *say* what they want to customize but they *show* it. Second, adoption of our approach is completely incremental (no change is required to the existing code base). This is important since many software products such as Dynamics have a very large existing code base. Third, partners have control over the discussed precision-versus-recall trade-off by adding more or less context code to their query, and by abstracting more or less the code queries using delegates. Another positive aspect of our approach is that we are re-using extensively existing .NET technologies: for example, we rely on the standard .NET compiler for C# or VB.NET, which means that support for the next version of these languages is much simplified. One issue is that we have to support the evolution of the intermediate language (IL), but the recent history has shown that IL evolves at a slower pace than high-level languages. The main con seems to be that query is mainly done by matching code patterns, which might not be convenient to denote complex variability points.

4.8. Conclusions

We described the upgrade problem and we emphasized that anticipation, one of the pillars of modern software engineering technology, is not completely adequate for modern ERP systems. We presented the Eggther customization framework that mitigates the upgrade problem while at the same time allowing for un-anticipated and fine-grained customizations. We introduced precision and recall as two useful measures for exactness and completeness. We gave an outline of the implementation and presented the pros ans cons of our main design decisions. After a discussion we briefly introduced some future work, and finally we mentioned some related research.

Acknowledgments

We would like to thanks Antonio Cisternino with whom we developed many of the ideas during a stay at University of Pisa. This stay was made possible thanks to the kind invitation of Egon Börger. Thanks to Peter Sestoft, Yvonne Dittrich, Morten Rhiger, and Kasper Østerbye for their help and support. This work takes place under the umbrella of the Evolvable Software Project, and is sponsored by NABIIT under the Danish Strategic Research Council, Microsoft Development Center Copenhagen, DHI Water and Environment, and the IT University of Copenhagen.

5

Aspect-oriented programming made easy: An embedded pointcut language

Antonio Cisternino
University of Pisa
cisterni@di.unipi.it

Sebastien Vaucouleur
IT University of Copenhagen
vaucouleur@itu.dk

Submitted to: IEEE Asia-Pacific Software Engineering Conference, APSEC 2009 [40].

Abstract

An important challenge with respect to aspect-oriented programming is to make this technology *easier* to use, so that it becomes accessible to a larger number of developers. We address this challenge with a new pointcut language based on the concept of *code query by example*. Our framework can be used to denote *code patterns*, which are difficult to express using traditional join point languages. A further benefit of our approach is that it can be used to denote join points at almost arbitrary locations inside method bodies – without sacrificing *obliviousness*. Finally, a particularity of our pointcut language is that it is *embedded* in a general purpose language. We outline the benefits, and limitations of our framework, and we summarize the implementation of a prototype.

5.1. Introduction

5.1.1. Modular decomposition

Parnas pioneered the idea of *modular decomposition* in the 1970s. Implementation details are hidden behind an *interface,* that forms the boundary of a *module*. Clients do

not access the implementation of the module directly but must use this well-defined interface. Since the details of the implementation are unknown to clients, they can be changed at will (for example in order to use more performant data structures). Parts that are *likely* to vary together should reside in the same module [158]. That is, programmers have to foresee, somehow, how a software product will evolve or need to be customized in the future.

5.1.2. Limitations of modular decomposition

One can pinpoint some of the limitations of modular decomposition approach as described above. In many cases, the programmer cannot *anticipate* reliably the future needs for customization [200]. Similarly, the decomposition might be convenient for some of the programmers using the system, but not for all of them: some might have some specific or unusual needs. Programmers typically deal with these problems by implementing the customizations in-place (code patching). In-place modifications are simple and straightforward, but quickly become problematic when the code has to be upgraded: code patches have to be ported manually to newer versions with limited tool support (typically, versioning tools), a problem that was dubbed *the upgrade problem* [182, 200, 58]. Another problem with modular decomposition is the fact that some features might be difficult to textually modularize (the typical example being logging every method entry), and hence their implementation remains scattered over the source code.

5.1.3. Aspect oriented programming

Aspect oriented programming (AOP) tries to tackle the problem of crosscutting concerns. At the core of most AOP frameworks resides a language, called a pointcut language. Section 5.2 gives a concise reminder of the most important AOP terms. We refer the reader to Filman et al. [74] for a good introduction to AOP frameworks.

5.1.4. A teaser

Using the .Net framework, a call to the static method `System.GC.Collect()` forces garbage collection [135]. It is well-known that garbage collection has strong performance implications, as well as more subtle functional implications, for example with respect to *weak references* [66, 65]. Hence, a programmer wants to trace all method calls to that particular method; he writes a code example that defines a query (ie., a pointcut):

```
public class PointCut1 {
 Query["GC"]
 void Q1() {
   System.GC.Collect();
 }
}
```

This query definition is used by our framework to instrument the code source at locations where the method is invoked, and the interface `GC.Before` is generated (among others, more on this later). This interface consists of a single abstract method, called `Customization`. The programmer implements this interface accordingly:

```
public class Aspect1 : GC.Before {
 public void Customization() {
  Console.WriteLine("Garbage collector invoked");
 }
}
```

And, *voilà*! At runtime, the message will be printed on the console before all calls to `System.GC.Collect()` – we claim that this approach is *easy*. This was of course, only a small example, the next sections will explore in depth the proposed approach, and look at more complex examples.

5.1.5. Applications

Our approach, based on the the concept of *code query by example*, was previously developed and studied in two different contexts:

- To customize large enterprise systems [201].
- To perform lightweight static analysis [39].

From a more general perspective, our framework can be used when code quantification or code customization is required. Applicability depends of course on the kind of customizations that are required, as this will become clear in the next sections.

5.1.6. Contributions

This paper makes three main contributions:

- An innovative embedded pointcut language based on the concept of *code query by example*, and a prototype that implements our proposal.
- A concise comparison with existing AOP frameworks.
- A study of the advantages and limitations of our approach: the limitations of the pointcut language, and the limitations of the prototype.

5.1.7. Road-map

The next section gives a very short reminder of some of the most important AOP terms. Section 5.3, the core of the paper, describes the pointcut language. The implementation of a prototype is summarized in section 5.4. Section 5.5 looks at more advanced cases, involving multiple customizations: how customized software products can be further customized. Section 5.6 addresses limitations of our framework, and section 5.7 compares our framework with other AOP tools. Section 5.8 gives some further references to related work. Finally, the last section concludes.

5.2. Aspect-Oriented Programming: Terminology

For completeness, we give a very short introduction to AOP terminology, focusing on the definition of the most important terms. We follow the terminology defined by Filman et al. [74]:

- **Cross-cutting concerns:** A crosscutting concern is a feature whose implementation is scattered through the rest of the base code; ie. which is not textually modularized.
- **Join points:** A join point is a location in the program where additional behavior can be attached.
- **Pointcuts:** A pointcut describes a set of join points; pointcuts provide a quantification mechanism.
- **Aspects:** An aspect is a modular unit designed to implement a concern. An aspect definition may contain some advices and the construction on where and how to invoke it.
- **Advices:** An advice is behavior to execute at a join point.
- **Obliviousness:** A target program is oblivious to customization when there is no explicit notation at the join point that the advice should be executed there.

5.3. An embedded point cut language

5.3.1. An embedded domain-specific language

The approach that we propose is based on a new *embedded domain-specific pointcut language*. It is *domain specific* since it deals with a well defined task: quantification over an existing software product. It is *embedded*, in the sense that all valid definitions of pointcuts and of advices in our language are also valid programs in the host language. Of course, the *interpretation* of a particular program differs depending if one looks at it in the context of our embedded language or in the context of the host language.

5.3.2. Notation

As noted above, all join points and all advices must be valid programs in the host languages (here, we use C# [65]). In the rest of the paper, for brevity, we sometimes only show the relevant methods, and assume that they are defined in an enclosing class. The name of the enclosing class in non significant, and mainly matters for documentation purposes.

5.3.3. Defining pointcuts

Pointcut are defined using *code examples*. These code examples will denote *code patterns,* and will be used to quantify over existing software products to locate join points *statically*. We make use of the concept of an *Attribute* in the .Net framework [66, 65] to add meta-data to a procedure: our framework defines a special attribute called a *Query attribute*. A *query method* is a method annotated with a query attribute. All query attributes take as an actual parameter of their constructor a string. This string will be used to *name* the query. A pointcut has the following form:

```
public class PointCut1 {
  [Query("Name1")]
  public void Q1(T1 p1, T2 p2, ...) {
    // code pattern
  }
  [Query("Name1")]
  public void Q2(T1 p1, T2 p2, ...) {
    // code pattern
  }
}
```

As shown in the example above, there can be multiple query methods with the same query name. In this case, the semantics of the query *Name*1 is defined as the disjunction of two cases: code pattern of the form Q1, or code pattern of the form Q2. A query can have as many disjunct cases as required. There are two main constraints to the definition of query methods: (1) query methods must not return any value – they must be *procedures* and not *functions* (2) query methods that participate in the definition of the same query must have the same *signature*.

Dually, a query method can be annotated with several query attributes:

```
[Query("Name3")]
[Query("Name4")]
public void Q3(T1 p1, T2 p2, ...) {
 // code pattern
}
```

Hence, the query method *Q*3 defines two queries: a query called *Name*3, and a query called *Name*4.

5.3.4. Query variables

Formal parameters of query methods define *query variables*. Given a query variable p of static type T, we distinguish two cases (the *target code* is the code being quantified):

1. First case, if T is a *delegate type* (for brevity, a delegate is a type safe reference to a function [66]), instances of a method call to p in the method body of a query method will denote:

   a) a non-empty sequence of statements in the target code, if T is a procedure – ie., if T is of type `Action`;

b) an expression of type R in the target code, if T is a function with a return type R – ie., if T is of type `Func<R>`.

2. Second case, if T is not a delegate type, an occurrence of p in the method body of a query method denotes a variable in the target code.

Several instances of the same query variable p in the method body of a query method, denotes the same sequence of statements, the same expression, or the same variable respectively if p is of type `Action`, `Func<R>`, or an instance of a non-delegate type.

For example, the following query:

```
[Query("Name5")]
public void Q5(Action a) {
 a();
 a();
}
```

will match a code fragment in lines 2 and 3 below:

```
1 public void M() {
2  Console.WriteLine('x');
3  Console.WriteLine('x');
4  Console.WriteLine('y');
5 }
```

5.3.5. Running example

As a running example, imagine that we are trying to quantify over a software product that accesses a database using the well-known *active record* design pattern [80]: briefly, for each table a class is created; an instance of one of these classes represents a database tuple; and each of these classes contains methods to save records, to delete records, etc. Consider a table `Teacher` with two attributes, `Name` and `Office` (for simplicity, assume that teacher names are unique), and its corresponding class `Teacher`.

If a programmer wants to express all join points where the name of a teacher tuple is set, he can write the following pointcut:

```
[Query("SetName")]
public void Q4(Teacher t, string name) {
  t.Name = name;
}
```

Consider the target program in listing 5.1.

Matching the query `SetName` against this program creates 2 join points: just before and just after the statement on line 7. Join points can be located in between existing statements, before the first statement, or after the last statement of a method.

5.3.6. Interface generation

Following the example above, 4 interfaces will be generated and added to the instrumented target program.

LISTING 5.1: TARGET PROGRAM

```
1  public class Target {
2   public static void Main() {
3    [...]
4    string n = "Prof. Smith";
5    string o = "4D11";
6    var t = new Teacher();
7    t.Name = n;
8    t.Office = o;
9    t.Save();
10   [...]
11   }
12  }
13 }
```

- **interface** SetName.Before
- **interface** SetName.After
- **interface** SetName.BeforeByRef
- **interface** SetName.AfterByRef

Each interface contains a single abstract method, respectively:

- **void** Customization(string name);
- **void** Customization(string name);
- **void** Customization(ref string name);
- **void** Customization(ref string name);

Why four interfaces when the first two and the last two share the same single abstract method? Because interfaces play a dual role: first, through the signature of the abstract method Customization, they enforce how advices should be implemented (see below); second, they indicate whether the customization – the advice – should be executed before or after the matched code fragments.

5.3.7. Two advices

Imagine that we want to customize the target software product to introduce the constraint that a teacher name should not be null nor empty. (This particular functionality could also be implemented as field validation at the database level). Recall that after matching the query `SetName`, the framework created 4 interfaces, among which `SetName.Before`. Adding the desired customization is just matter of implementing this interface, and placing the assembly in the same directory of the target program.

```
public class ValidName : SetName.Before {
 int Count;
 public void Customization(string name) {
  Count++;
  if(String.IsNullOrEmpty(name))
   throw new ArgumentException(...);
 }
}
```

The first time the runtime reaches the join point, an instance of ValidName will be created, and the method `Customization` will be called, binding the name variable in the scope of the join point. More precisely, all classes that implement `SetName.Before` will be instantiated, creating *customization objects* (singletons), and their `Customization` method will be called in sequence. Subsequent execution of the same join point will only retrieve the existing customization objects and call their `Customization` method. Customizations can have state, for instance the advice above counts the number of times a teacher's name is set.

5.3.8. Instrumentation and triggering of advices at join points

Let's step back a little, and discuss more precisely what happens after a code fragment is matched. Recall that we use code examples, defined for example as a query Q. This query Q is used to match code fragments within the target software product. Matched code fragments denote two precise locations: the first one exactly before the match code fragment, the second one exactly after. At those two join points, method calls will be inserted on customizations that conform to one of the generated interface:

- Before the matched code fragment, all customizations that implement `Q.Before` and `Q.BeforeByRef` will be created and invoked;
- After the matched code fragment, all customizations that implement `Q.After` and `Q.AfterByRef` will be created and invoked.

Creation happens only the first time the code is executed at join points; successive executions at the same join point only retrieve and call the existing singleton objects. Method calls to advices are made using delegate calls not reflective calls (for performance reasons, see section 5.7). Also, an advice is loaded dynamically during program execution – which makes it easy to add behavior at runtime (see section 5.5).

5.3.9. Advice with a side effect at join point

So far our customizations had no direct side effect in the scope of the join points. We show an example of such side effect using the generated interface`Insert.BeforeByRef` : we make sure that teacher names are trimmed.

```
public class NormalizeName : Insert.BeforeByRef {
 void Customization(ref string name) {
```

```
  name = name.Trim();
 }
}
```

Note that this does not modify the teacher tuple directly, instead it modifies the variable `n` used to set the teacher's name.

5.3.10. Code pattern matching

```
[Query("ModifiedTeacher"]
public void Q5(Teacher t, string s) {
 t.Name = s;
 t.Save();
}
[Query("ModifiedTeacher"]
public void Q5(Teacher t, string s) {
 t.Office = s;
 t.Save();
}
```

The listing above shows an example of a complex pointcut that would be difficult to express using traditional pointcut languages: the query `ModifiedTeacher` matches all code fragments – inside method bodies – where the field `Name` *or* the field `Office` is set, followed immediately by a `Save()` operation. Two more points:

- First, the case where the field Name or the field Office is set with an expression, for instance: `t.Name = GetName();` can be expressed using a query variable of type `Func<string>`, see section 5.3.4.
- Second, the constraint that the setting of a field must be immediately followed by a save operation can be relaxed using a query variable of type `Action`, see also section 5.3.4. But, the risk then, is to capture code fragments where the variable `t` is re-bound – which is probably not what the programmer wants to express. For example:

  ```
  t.Name = "Prof. Smith";
  t = new Teacher();
  t.Save();
  ```

5.4. Implementation

Our prototype, Eggther, leverages existing .Net technologies, and exploits a symmetry inherent to our approach, that allows us to perform matching at the bytecode level. Bytecode analysis and instrumentation currently builds on top of Cecil, a well supported library, part of the Mono project [143]. There are various other options to perform bytecode analysis and instrumentation, including the performance-oriented CLIFileRW library [41].

5.4.1. Bytecode matching

Recall that both the target program, the pointcuts (the queries), and the advices, are valid .Net programs, since our language is embedded in a host language, such as C#. Compiling the target program and the point cuts turn those high level programs into a bytecode representation that can be exploited for matching, see figure 5.1. The main idea is that a match at the bytecode level implies a match at the C# level. The target program and the advices must be written in the same host language and compiled by the same compiler.

Performing matching at the bytecode level presents a number of advantages:

- Compilers enforce type checking of point cuts, remove unnecessary information such as comments or code indentation, and leave a sequence of byte-code that can be directly exploited for code matching. The subtle point is that even if compilation loses information, it does so symmetrically between the compilation of the target code and the compilation of the advices.

- Important information, such as type information, is retained at the bytecode level. As a simple example, the C# compiler transforms the static method call `Console.WriteLine('x');` into the following bytecode instructions:

  ```
  L_0000: ldc.i4.s 120 // load 'x'
  L_0002: call void System.Console::WriteLine(char)
  ```

  Obviously, the instruction labels will vary depending on the location of the statement, this will be further discussed in section 5.4.3.

- Standard compilers, such as *csc*, the Microsoft C# compiler, are well-tuned and actively maintained. It leverages a highly performant compiler, and allows us to circumvent the implementation of complex C# language features , such as conditional pre-processor directives [65], that would be required if our implementation worked at the C# abstract tree level.

5.4.2. Abstract stack interpretation

After various information about the existing query methods are gathered, one of the first challenge is to identify statements locations within the method bodies at the bytecode level. In our implementation, this is done by performing a simple form of abstract stack interpretation. In a few words:

- One can infer exactly what is the impact of each instruction on the stack height. For example, in the last paragraph, loading the character `'x'` on top of the stack, increases the stack height by one; the subsequent static method call to `Console.WriteLine(...)` decreases the stack height by one, etc.

- Any statement is exactly contained in-between two locations where the stack height is zero.

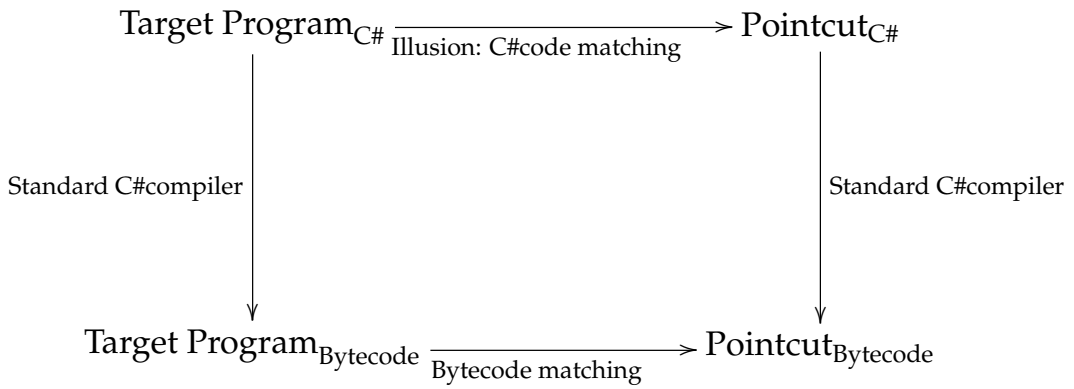


FIGURE 5.1.: MATCHING

5.4.3. Regex matching

Our implementation leverages the well-optimized regular expression library, part of the .Net framework. One should differentiate regular expression as studied in the field of theory of programming language from modern regular expression libraries, called *regex*. Regex libraries were extended with features that make them much more powerful that their formal counterparts. For example, our implementation makes use of *named captures* [135]. In few words, named captures means that matched substring can be referred to with a name. It is particularly important since, for example, variable names in the pointcut and variable names in the target program can differ. For instance, the code fragment: `int x = 0; x++;` and the code fragment: `int y = 0; y++;` use different variable names, but yet should match. Another example where name capture is needed, is for branch instructions: the instruction labels in the pointcuts are obviously different from the instruction labels in the target program. An advantage of using the .Net Regexp library is that patterns can be runtime compiled, yielding better performance.

5.4.4. Run-time generation of delegates

Advices are loaded at runtime and delegates are generated to efficiently invoke the customization methods. For this, we build on the *Managed Extensibility Framework* (MEF), a library recently made available by Microsoft, which will be part of the next release of the .Net framework [127].

5.5. Multiple customizations

This section studies how a customized program can be further customized. Multiple customizations have concrete applications in the form of software products, software

that have important needs for customization, for example enterprise systems [200, 182]. We distinguish three cases:

- The case when a new advice is added to an already instrumented assembly,
- The case when an instrumented assembly is further instrumented,
- The case when advices are composed together.

5.5.1. Adding an advice

In this section, we follow the *invoice example*, inspired by Johansen et al. [104]. Consider a class `Invoice` with a method `GrandTotal()` :

```
public class Invoice {
 public virtual double GrandTotal() {
   double total = ...; // computation
   return total;
 }
}
```

We construct the following code query example:

```
[Query("Total"))
public void Q1(Invoice i, double total) {
  total = i.GrandTotal();
}
```

Both the target program and the advice are compiled; the target program is instrumented as explained previously, and interfaces are generated. The first advice in the invoice example implements a discount if the grand total exceeds 10 000:

```
public class Discount : Total.AfterByRef {
 void Customize(ref Invoice i, ref double total) {
  if(total > 10000) total *= .95;
 }
}
```

When the customized program will be executed, the customizations will be triggered. If one would like to add further advices with the existing join points, it is simply a matter of: (a) implementing once again one of the generated interfaces (b) compiling the advice. The new advice can be loaded dynamically (at runtime). For example, imagine that we add a special tax of one euro, and keep track of how much tax was collected:

```
[After(typeof(Discount)]
public class Tax : Total.AfterByRef {
 protected int collected;
 void Customize(ref Invoice i, ref double total) {
  total++;
  collected++;
 }
}
```

There is a small twist. Obviously, the order in which the advices are called matters: $(1+x)*0.95 \neq 1+(x*0.95)$. A special feature of our framework allows us to specify a partial order between customizations using the attribute `After` and `Before`. The example above shows that we wish to perform the tax *after* the discount. When a given join point is reached, the framework will compute a linear order between customizations that implement the same interface. Internally, this is implemented using a topological sort. If no linear order compatible with the partial order can be found, an exception is raised (for instance, the tax advice cannot be executed both `After` *and* `Before` the discount advice).

5.5.2. Further instrumentation

We show an example where the instrumented assembly from the previous section is further instrumented. We also attempt a more ambitious customization: we would like to replace instances of `Invoice` with instances of a subclass, `Invoice2`. This subclass can of course introduce new fields, such as an `Address` field. We shall also override the `GrandTotal()` function. First, we write the new query, as a code example:

```
[Query("NewInvoice"))
public void Q2(Invoice i) {
 i = new Invoice();
}
```

As usual, the query is compiled, and is used to instrument the assembly generated in the last section. One makes an attempt to implement the desired functionality as:

```
public class Substitute : NewInvoice.After {
 public void Customize(ref Invoice i) {
  i = new Invoice2();
 }
}
public class Invoice2: Invoice {
 protected string Address {get; set;}
 public override double GrandTotal() { ... }
}
```

This works as expected: the redefined version of `GrandTotal()` will be executed when the target program is executed. The watchful reader will notice that the invoice is created twice: it is first created in the target program and then immediately re-assigned in the advice. This means a performance penalty, but more importantly, this is not exactly what we wished for: the constructor of the class `Invoice` could have subtle side effects. Two conclusions: Eggther support further weaving but does not support so-called *introductions* (we will come back to this in section 5.6).

5.5.3. Advice Composition

Advices can be composed by instrumenting an existing advice, following the same techniques described before. Nonetheless, we expect that in most cases this would not be very useful, since advices tend to be very small.

5.6. Limitations

Limitations of the join point language As noted in the previous section, our join point language cannot express introductions, and can only locate join point statically. Currently advices cannot be generics, although this limitation could be lifted in a future release. The receiver object is currently not accessible in the advice (the receiver object is the object enclosing the matched code fragments). This could be easily fixed by adding an extra formal parameter to the `Customization` methods, and binding this parameter to the current object in case of a non-static enclosing method – this functionality was available in a previous version of our framework, but is currently temporarily removed.

Limitations of the prototype Being based on bytecode matching, our prototype currently suffers from context sensitive optimizations (optimizations that are not necessarily applied symmetrically when compiling the target code, and compiling the pointcuts). Our framework does not deal with non-local transformations by compilers, such as the ones introduced by *enumerators* in C# [65]. Our implementation cannot remove an existing sequence of statements in a method body (a feature that can be considered, arguably, as too powerful), although we believe that this could be accomplished by performing more complex bytecode instrumentation.

5.7. Comparison with Other AOP Frameworks

We provide a concise comparison with other .Net AOP frameworks. There are, of course other very influential AOP frameworks outside of the .Net world, among others: AspectJ, CaesarJ, and Hyper/J, see [74] for an introduction to those frameworks.

5.7.1. Discovery of join points

Using our framework, join points are discovered statically: we do not support *cflow*-like constructs. Join points are also discovered exclusively through quantification; other AOP frameworks make use of explicit markers in the target code, such attributes on the method that need to be customized (PostSharp LAOS [164]). This makes the implementation easier, but sacrifices obliviousness.

5.7.2. Advices loaded statically and/or dynamically:

Most AOP frameworks load advices statically. Among the notable exceptions are Wicca Phx.Morph [208] and Loom.Net [120], that have support for dynamic updating. Eggther loads advices dynamically, which makes it easier to add customization at run-time. On the other hand, instrumentation only takes place statically, contrary, for example, to Wicca [208] that supports runtime weaving.

5.7.3. Join point locations:

Most AOP frameworks are limited to join points before, after, around methods or more generally, at locations where a class member is accessed. Our framework can match sequences of statements at arbitrary location within method bodies.

5.7.4. Control flow

Our framework does not have any `Proceed` construct. Contrary, for example, to Yiihaw [104], control flow goes from the enclosing method (at join point) toward the advices. A consequence is that existing behavior cannot be removed (when the flow goes from the advices toward the enclosing method, it is simply a matter of not calling the `Proceed` method). A work around is to throw an exception within an advice, as shown in section 5.3 (this works since exceptions are not checked in .Net [66]).

5.7.5. Introductions, modifications:

So-called *introductions* can be used to add a member (such as a method, an event, etc.) to an existing class; *modifications* can be used to change the base class of an existing class, to make it implement more interfaces, etc. These kinds of changes are allowed by AspectDNG [13] and Yiihaw [104], among others. Eggther does not support introductions, nor modifications. In our framework, the emphasis is to change the behavior of existing methods, not to change the existing class structures – which seems sufficient for a large range of customizations.

5.7.6. Runtime overhead

In a few words, with respect to runtime overhead, one can differentiate 3 notable cases: (a) when advices are in-lined (b) when advices are triggered using a delegate call, and (c) when advices are triggered using a reflection call. Method calls through reflection are notoriously slow: they show an overhead of a factor 5-50. Delegate calls have been extensively optimized on .Net 3.5. Yiihaw belongs to the first category and yield a zero-overhead performance penalty [104]. Our framework uses delegate calls, similarly to Rapier Loom [120]. AOP frameworks that do not focus on performance, such as AspectDNG [13] tend to use Reflection. Frameworks that do not focus on performance tend to provide better flexibility and expressiveness.

5.7.7. Visualization of Join points:

Eggther can match statements at arbitrary locations within method bodies, hence it is important to show the programmers where the join points are located. We provide a simple Visual Studio plug-in for this [67]. Other AOP frameworks also provide visualization capabilities, such as Aspect.Net [176] or EOS [69].

5.8. Related work

Code query by example Code query by example has been used extensively in the database community [184]. A few related projects have explored some variants of code query by example with respect to a general purpose programming language, for example the work by Cohen et al. on JTL [44]. One of the main difference with our framework is that they do not use an embedded language: their language is a variant of Java, and hence cannot be compiled with a standard compiler. Similarly, De Roover et al. use Java code patterns embedded in a logic query language [173]. Finally, Martin et al. use a dedicated query language, called *Program Query Language*, to embed code patterns [124]. Again, contrary to our approach their host language is a dedicated language, and not a general purpose language such as C#. In the Aspect Oriented Programming community, some variant of CQE have been proposed; the work by Noguerra et al. came recently to our attention; building on the Spoon framework [186], they propose to use source code templates to denote static pointcuts [149]; their proposal is, to the best of our knowledge, the closest to CQE, since their pointcut language is embedded in pure Java. Their implementation works at the abstract syntax tree level, whereas we perform matching and instrumentation at the bytecode level.

Diff and patch tools Diff tools, patch tools [121], and revision control systems in general (such as subversion [45]) are related to matching, but tend to work at a purely textual level. Meaning, for example, that the method calls: `f(a);` in file *A*, and `f(a);` in file *B* would match, even if the variable `a` in file *A* and the variable `a` in file *B* have a different static type. This is not the case with our approach: our matching procedure enforces type matching. There are, of course, more advanced diff and patch tools, that have language specific or data type specific functionalities, see for example the Coccinelle project [43]. There are also a wide variety of advanced diff tools in the industry; see for instance, Beyond Compare [27] that has specialized viewers for a variety of data types (such as HTML, or even images) and 3 ways merge, or WinMerge [210], that can for example compare Excel files using a dedicated plug-in.

Dependency injection and Inversion of Control Containers IOCCs are very active topics in the industry, see for example the Castle project and their Windsor container [37]. In the words of the Castle project: "[Inversion of control] is the opposite of using an API, where the developer's code makes the invocations to the API

code. Hence, frameworks invert the control: it is not the developer code that is in charge, instead the framework makes the calls based on some stimulus". Similarly to our work IIOCCs often rely on injection of dependencies. There is no clear-cut separation between what is AOP and what is dependency injection: one can be used to implement the other. Nonetheless, the emphasis with IIOCCs is more on complex configuration capabilities; on the other hand, using IIOCCs, obliviousness is typically not strictly respected: anticipation is often required in the form of a predefined interface and explicit calls to a container.

Pattern matching and unification Programming language support for *pattern matching* has been studied extensively by the functional language community, see for instance Sestoft [181] for a study of efficient implementation of pattern matching. Our matching requirements can be related to what the functional community refers to as non-linear patterns: patterns where the same variable can appear multiple times. Pattern matching itself can be seen as a special case of unification where there is no free variable in the target: unification is *two-way matching* [60].

Conclusions and future work

We have presented a new pointcut language, which is, to the best of our knowledge, a new approach to AOP. This framework is based on the concept of *code query by example*. One of the benefit of our approach is that join points can be located at arbitrary locations within method bodies – without sacrificing obliviousness. Another benefit of our approach based on an embedded language, is that we have full support for all the functionalities offered by existing development environments, such as design time and interactive typing, re-factoring, etc. We have shown that a customized software can be further customized, for example by adding a customization assembly at run-time. We shortly presented our prototype, which tries to innovate by doing matching within method bodies at the bytecode level, exploiting a symmetry between the bytecode representation of the advices and the bytecode representation of the target program.

We see many opportunities for further work: one could look at ways to regain some of the expressiveness that was lost, while trying to retain the concept of code query by example. For example, code matching could be done at the class level.

Acknowledgments

Thanks to Peter Sestoft and Estelle Barbot for their comments.

6

Describing default rules, prescribing custom rules

Antonio Cisternino
University of Pisa
cisterni@di.unipi.it

Rasmus Rasmussen
IT University of Copenhagen
lynet@itu.dk

Sebastien Vaucouleur
IT University of Copenhagen
vaucouleur@itu.dk

Technical Report [38], IT University of Copenhagen.

Abstract

In the recent years, rule checking software became a must for software practitioners: these tools, based on lightweight static analysis, provide warnings when software maintenance is required, and hence contribute to the quest for software quality. Rule checking tools are nonetheless the victims of their success: their user base is no longer limited to senior programmers comfortable with the ins and outs of bytecode analysis, but extends to project managers and junior programmers. These new kinds of users require tools that are easier to understand, compare, and use. In order to face this challenge, we suggest two measures. First, we propose to help comprehension of rule checking software by giving a unified categorization of default rules – the rules provided by the tools. Second, we propose the innovative concept of code query by example as a new way to express custom rules – the rules written by the users.

6.1. Introduction

Rule checking tools currently enjoy a momentum in the industry. The number of downloads alone confirm a real interest from the IT professionals – currently on the order of 10,000 downloads per month[1] for one of its prime representative, FindBugs

[1] Sourceforge download statistics, beginning of 2009

[76]. Related tools enjoy a similar interest as one can witness a rising activity on specialized forums.

Lightweight static analysis Rule checking tools are based on the attractive and intuitive concept of lightweight static analysis: programmers can easily detect a number of potential rule violations by performing fast and automatic program analysis. Rules can be checked on a regular basis and provide warnings, so-called *positives*, when software maintenance is *likely* to be required.

Challenges Two challenges lie ahead. First, as the related tools are flourishing, the number of default rules is exploding. Since each tool makes its own rule categorization, comparison between the tools is difficult. Second, even though the tools are easy to use with *default* rules, writing *custom* rules is relatively difficult. It typically requires understanding of bytecode, the ability to reason at a meta-level, and in some cases, (viewed from the perspective of the user) the willingness to learn an arcane domain-specific language.

Contributions This paper makes two contributions. The first one relates to default rules, the second one to custom rules. Both contribute to the goal of making rule checking tools easier to use and comprehend.

- To improve tool comprehension, we examined five of the most prominent tools and inferred a categorization based on their default rules – 933 rules altogether. Our categorization is thematic and complete.
- To address the difficulty of writing custom rules, we propose a new language based on the concept of *code query by example*. We believe that this language provides a simple and intuitive way to express custom rules.

Road-map We introduce the main concepts behind rule checking and the tools discussed, respectively in sections 6.2 and 6.3. The categorization is presented in section 6.4. Section 6.5 introduces a new way to express custom rules and compare this approach with existing techniques. We compare our contributions to existing work in section 6.6. Finally, section 6.7 concludes.

6.2. Rule checking software

Software maintenance Unlike physical products, software products do not wear out from repeated usage – but yet require maintenance. This fact is often linked with evolution. Extensive empirical studies [128] have showed that, in many cases, software evolution is mandatory. Unfortunately, every-time the software text is touched, defects or imperfections can be introduced inadvertently. A naive solution is to perform extensive code reviews on a regular basis to make sure that software

quality is still up to standard. Another solution is to tackle this tedious and repetitive task by using dedicated software – software that analyze software [97, 52, 203].

Lightweight static analysis To the best of our knowledge, there is no clear and well accepted definition of the concept of lightweight static analysis. In this paper, we will simply define it as analysis that can be completely automated, which does not perform inter-procedural context sensitive analysis, that always terminates, and which does so in a reasonable amount of time on a modern personal computer.

Rule checking tools Several aspects differentiate modern rules checking tools from traditional warning systems (such as Lint). One of them is the use of dedicated graphical environment (or plug-ins) to report warnings. Another one is the extensive use of false positives. More importantly, it is the capacity offered to the users to express custom rules that distinguish them from their predecessors.

Default rules versus custom rules Rules come in two flavors. They can be either pre-defined (that is, be supplied with the tools), we will call them *default rules*, or can be defined by the user, we will call them *custom rules*. Default rules are usually very much appreciated by users, since they can be used immediately. Custom rules are also important, since many companies use special conventions, or have particular quality control procedures.

Definitions *Mistakes* are made by IT professionals during the specification, or the implementation of a system, and result in *defects* in the implementation of a system; defects typically result in system crashes (*faults*), or in results that does not respect the software product specification; *imperfections* are considered to be code artifacts that are not optimal, as opposed to defects (typically, imperfections do not result in system crash, nor do they impact the result of a computation); finally, *flaws* are either defects or imperfections.

Positives and negatives Rule checking tools report a list of warnings that indicate code areas that are *likely* to need further investigation. We call warnings *positives*. If those warnings are correct, meaning that there is indeed a need to modify the code, then they are called *true positives*. If they turn out to be nuisance alarms, we call them *false positives*. Finally, issues in the code which are not reported by the tool are called *false negatives*. Naturally, tools try to maximize the number of true positives. A small number of false positives is generally acceptable (as long as the contract with the users is made clear), since programmers can manually go through the list of warnings and validate if each warning is indeed a true positive. On the other hand, a large number of false positives is typically a deterrent. False negatives are more problematic, since they can bring a false sense of confidence with respect to the quality of the code base.

Table 6.1.: RULE CHECKING TOOLS SUMMARY

Tool selected	*Version examined*	*Platform*
FindBugs	1.3.6 (free standalone version)	Java
FxCop	1.36 (free standalone version)	.Net
NDepend	2.11.2 (professional edition)	.Net
Semmle	0.5 (free edition plug-in, Eclipse)	Java
StyleCop	4.3 (free plug-in, VS Studio)	.Net

6.3. Tools examined

Selection criteria We selected the tools using the following set of criteria: both .Net and Java tools must represented; both commercial and academic tools must be present; the tools must have a substantial user base; and finally, the tools must offer the functionality to create custom rules. Time limits forced us to narrow down the selection to five tools, see table 6.1.

FindBugs Findbugs is open source and is part of an ongoing research project [76]. The tool performs static analysis at the bytecode level. Like the other tools, FindBugs comes with a set of default rules, but also provides a full framework to write and report custom rules.

FxCop FxCop is developed by Microsoft [82]. It can be used for free, but the source code is not available. This tool targets the Common Intermediate Language (CIL). It mainly enforces the guidelines recommended by Microsoft, but can also be used in more domain specific contexts.

NDepend NDepend is a commercial product [147]. This tool principally targets .Net CIL, but has also limited support for C# (when debugging files are available).

Semmle Semmle is commercially licensed but a free version is available for non-commercial usage [180]. This product analyzes both Java source code and bytecode. Semmle aggregates data into a database, and queries are made against this database using a dedicated language.

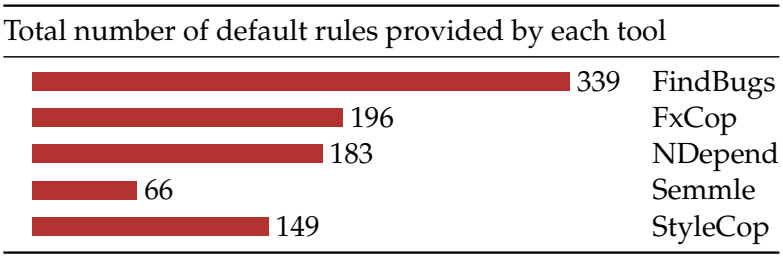
StyleCop StyleCop is developed by Microsoft [191]. It is free to use but it is close-sourced. Contrary to FxCop, this tool works at the source code level. Much like the other tools, it can also be used alone (for example as part of a build script), or through a development environment (Visual Studio).

6.4. Describing default rules

Rule checking tools are flourishing, and each new version of those tools regularly increase the number of default rules available to users. On one hand, the users can happily use many rules made out of the box, but on the other hand, they quickly become overwhelmed by a large quantity of rules. A hierarchical structure can help in this respect. In this section, we present a unified thematic categorization of rules, inferred from the default rules of tools presented in section 6.3.

Regarding categorizations Note that categorizations are not unique: as it is often the case, they are several ways to abstract over an existing domain. Our categorization was made with the user in mind, how to best help him select the appropriate rules or tools. Note also that categorization are not fixed: just like most software systems, they have to be maintained, or they will eventually stop being relevant.

Approach and overall structure The categorization was made by inspecting the existing default rules of the tools, 933 rules altogether, and inferring candidate categories. Our categorization is thematic, meaning that it tries to reflect the major themes found in modern rule checking tools (we contrast this approach with other options in section 6.4.7). Rules are either related to *flaws* or *code changes*. The flaw category consists both of defects and imperfections and relate only to one version of a software product. The code changes category considers multiple versions of the same software product. (The full categorization is given in the companion document [39].)



6.4.1. Flaw rules

With respect to flaws, we distinguish style rules; documentation rules; design rules and implementation rules. *Style rules* are not defects, they are merely imperfections. Nonetheless, strict observance of style rules is important to facilitate source code comprehension, especially in large software products maintained by a large number of developers. For example, some .Net conventions encourage developers to use only one namespace per file (this is usually not enforced by compilers). Similarly to style rules, *documentation* is an important aspect of software quality. We will further describe these categories in section 6.4.3. *Design rules* and *implementation rules* make for the largest part of default rules in most tools, these will be described respectively in sections 6.4.4 and 6.4.5.

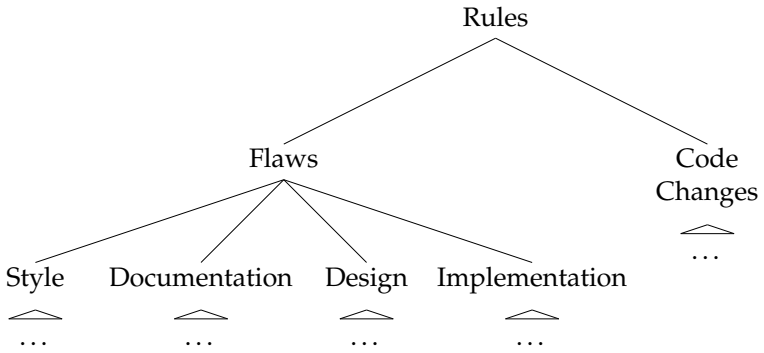


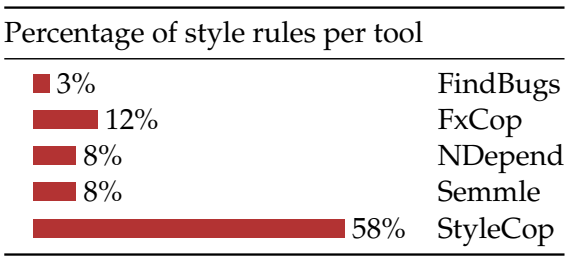
FIGURE 6.1.: FLAWS

The categorization of existing default rules shows that tools are very oriented towards flaws, with the exception of NDepend that gives explicit support for code changes, this will be detailed in section 6.4.6. Flaw rules represent 75% of NDepend default rules and 90% of FindBugs default rules. (In this paper, percentages are always given with respect to the total number of rules for a given tool.) The default rules of the other tools are all flaw rules.

6.4.2. Style rules

Figure 6.2 summarizes the sub-categories related to style rules. The naming category concerns all conventions around member names, class names, and file names (for example, in .Net, method names should start with an uppercase letter). *Layout* refers to the arrangement of code in a file (for example, code indentation). Finally, *ordering rules* concern the relative textual position of two elements: for example, static members should appear before instance members (StyleCop). Tools that can work at the source code level have a clear advantage here, since this kind of information might be unavailable in bytecode.

StyleCop has the most extensive support for style rules through its default rules. Default rules of other tools related to style rules mainly concern naming.



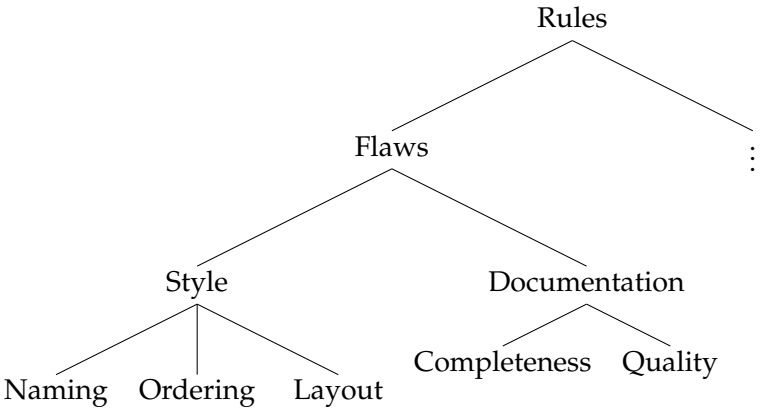


FIGURE 6.2.: CATEGORIZATION: STYLE AND DOCUMENTATION RULES

6.4.3. Documentation rules

With respect to documentation rules, we distinguish two sub-categories: the *completeness* of the documentation and its *quality* (figure 6.2). The former category contains rules that check whether a particular documentation aspect is absent, the latter checks for a property of some existing documentation elements. Most tools give little support for documentation through default rules, excepts StyleCop. For example, this tool checks whether some documentation element has a summary (which we refer to as documentation completeness), and whether comments contain valid XML (a documentation quality rule). StyleCop has 32% of its default rules related to documentation, and NDepend 1%. The other tools do not have any default rules in this category.

6.4.4. Design rules

We now turn our attention to design rules, one of the main focus of most of the tools that we examined. For typographical reasons we prefer to show the sub-categories of design rules using a table (see table 6.2), together with the distributions of rules per tool.

Percentage of design rules per tool	
23%	FindBugs
58%	FxCop
34%	NDepend
73%	Semmle
3%	StyleCop

We summarize the support for design issues through default rules below. (Using absolute numbers, the tools that allocate the more rules to design issues are in

Table 6.2.: DESIGN RULES

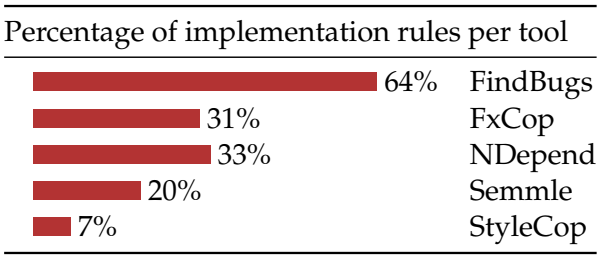
Design rules	*FindBug*	*FxCop*	*NDepend*	*Semmle*	*StyleCop*
Dependencies	0%	1%	8%	2%	0%
Inheritance	1%	1%	2%	0%	0%
MemberDesign	14%	26%	8%	62%	0%
Metadata	0%	6%	3%	0%	1%
Modifiers	6%	11%	14%	9%	2%
Security	3%	11%	0%	0%	0%
Version	0%	1%	0%	0%	0%

decreasing order FxCop, FindBugs, NDepend, Semmle and StyleCop.)

6.4.5. Implementation rules

Together with design rules, implementation rules constitute the core of most rule checking software. The distinction between *design* and *implementation* is not always sharply defined; nonetheless, we find the distinction between design and implementation useful and rather intuitive in most cases. We distinguish 12 subcategories to implementation rules, which are listed in table 6.3.

We summarize the support for implementation themes through default rules below. Historically, NDepend first focused on dependency management but now support rule checking in general, and especially implementation rules, as it is clear from the summary below. Finally, FindBugs shows a clear focus on implementation issues.



6.4.6. Code changes

We now consider the second top-level category: code changes, see figure 6.3. We distinguish rules that refer to the evolution of a given software product, called *own code*, with rules that deal with the evolution of *tier code* (code for which a given software product as a dependency). NDepend is the only tool that gives explicit support for code changes through its default rules (25% of its rules).

Table 6.3.: IMPLEMENTATION RULES

Impl. Rules	*FindBug*	*FxCop*	*NDepend*	*Semmle*	*StyleCop*
Assignment	1%	0%	0%	0%	0%
CodeComplexity	0%	0%	4%	0%	0%
Comparison	8%	0%	0%	0%	0%
Concurrency	13%	0%	4%	5%	0%
ControlFlaw	3%	0%	0%	0%	0%
Exceptions	2%	6%	1%	6%	0%
Invocations	12%	12%	3%	6%	4%
NullReferences	7%	0%	0%	0%	0%
ResourceMgt	6%	6%	1%	0%	0%
SuperfluousCode	5%	4%	3%	3%	2%
Testing	2%	0%	16%	0%	0%
Typing	5%	6%	1%	0%	1%

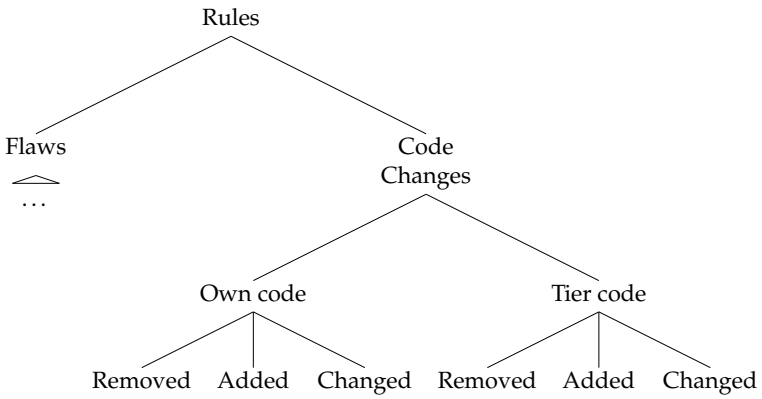


FIGURE 6.3.: CODE CHANGES

6.4.7. Reflections on the categorization

In this section, we discuss design choices for our categorization, and we contrast them with possible alternatives.

Summary of our choices Our categorization is thematic, meaning that we tried to infer the major themes behind existing default rules. Our categorization forms a tree, which makes it easy to comprehend, but also calls for difficult choices. In particular, some rules are candidates to appear in multiple categories, and therefore a choice had to be made on which category some the rules more naturally belong to. We first considered having defects and imperfections as sub-categories, but it turned out that this form of categorization forced us to make many arguable choices with respect to what is a defect, and what is simply an imperfection. We deliberately avoided certain categories that might appear evident at first sight, such as metrics. We address this choice below.

Concerning "Metrics" *Metrics* is a candidate category. NDepend and Semmle give explicit support for metrics. Metrics refers to rules that map software product to a number, for example:

$NbParameters : Method \rightarrow Integer$

Note that metric rules can easily be turned into predicates, typically by fixing a threshold. For example, if methods should not have more than 8 parameters:

$TooManyParameters : Method \rightarrow Bool$

$TooManyParameters(x) = NbParameters(x) > 8$

Similarly, virtually all rules can be turned into a metric by counting the number of elements that have a certain property. Hence, we find that the distinction between metric and non-metric rules does not reflect the essence (the theme) of a given rule.

Member specific categories We tried to avoid categories that would refer to a specific kind of member (such as classes, methods, assemblies, etc.). The rationale behind this choice is that member specific categories would force very closely related rules to belong to different categories. Also, this choice allows us to keep the number of categories down to a reasonable number.

Framework specific categories Finally, we tried to avoid framework and language specific categories (such as Enterprise Java Beans). Our categorization does make a number of assumptions with respect to the language of the target code base, but we tried to keep those assumptions to a strict minimum to make our categorization applicable to a greater number of platforms.

6.5. Prescribing custom rules

PRESCRIBE: (1) To lay down a rule.
Merriam-Webster Dictionary

In this section we approach the second challenge behind rule checking software, as stated in section 6.1: how to express custom rules easily. When existing default rules are not enough for a user, he will want to write his own rules to satisfy his needs. All the tools we discussed so far allow for this. Unfortunately, those tools typically choose one of the following options:

- Either they require the user to have a good knowledge of bytecode (or if they work at source code level, of the structure of an abstract syntax tree). In which case they will instruct him to write source code that inspects existing bytecode or abstract nodes. This is the choice made by FindBugs, FxCop and StyleCop.
- Or alternatively, they require the user to use a declarative, domain specific language that is very convenient to query code – but which is distant from his working language. This is the choice made by NDepend and Semmle.

We argue that many users neither have a good working knowledge of bytecode, nor are willing to spend a lot of time learning a powerful but complex domain specific language. We propose a new approach, based on *code query by example* (CQE), that does not require the user to know about bytecode or about abstract syntax trees.

6.5.1. Code query by example

Query by example is a well-known approach [213] to querying that requires lesser knowledge about formal methods than its formal counterparts, such as Datalog [196]. Using query by example, the user provides a template of the class of documents he is interested in. We follow this approach by creating a query language for matching program fragments with a structure defined by example.

Overview of CQE *Code query by example* (CQE) can be seen as an embedded domain-specific language. The target domain is code matching: querying over an existing code base. The language contains a very small number of primitives. A *program fragment* is a sequence of statements. The result of a code query is a set of program fragments. Queries are written using code examples. Programmers express their queries using their favorite high level programming language (the host programming language must support meta-data annotations, like attributes in the .Net framework); we use C# in the discussion thereafter [65, 66].

Typing and Matching A *query method* is a method annotated with the *Query* attribute; queries contain code patterns that will be used to match code in the target

code base. Instances of query attributes have a *query name*, specified as the first actual argument of the attribute constructor. The name of the query method is not significant. All query methods are valid C# programs, hence any C# compiler can be used to ensure that queries are well-typed. Formal parameters of query methods define *query variables*. In .Net, `Action` and `Func<R>` denote higher-order types, respectively procedures and functions – they are given a special semantics in CQE, as explained below.

Given a query method Q with a formal parameter `x` of static type `T`:

- If `T` is a subtype of `Action`, all occurrences of `x()` in Q must match the same sequence of statements.
- If `T` is a subtype of `Func<R>`, all occurrences `x()` in Q must match the same top-level expression of type `R`.
- Else, occurrences of `x` in Q must match variables of type `T`.

A query Q matches a fragment of code C if all occurrences of query variables in Q can be matched in C and if every other statements match exactly.

Consider the following query Q:

```
[Query("Q")]
void AssignTwice(int g, Func<int> f)
{
  g = f() * 2;
}
```

This query can be read as follows: "Find all the code fragments that assign to a variable of type **`int`** the value obtained as the multiplication of an expression of type **`int`** by the number 2".

6.5.2. First example

FindBugs describes the following defect pattern:

> "QUESTIONABLE_BOOLEAN_ASSIGNMENT: This method assigns a literal Boolean value (true or false) to a Boolean variable inside an if or while expression. Most probably this was supposed to be a Boolean comparison using ==, not an assignment using =." [76]

Using FindBugs FindBugs uses a visitor pattern [83] to implement a rule that can detect this defect. The class in listing 6.1 is taken from the FindBugs distribution, and contains two important methods: `visitCode` and `sawOpcode`, both override a base method. The former is called when a new method is entered by the analysis framework. The latter is called for each instruction contained in a method. The programmer has access to both the opcode and the operands of a given instruction [97]. With that information in hand, the programmer can for example implement

LISTING 6.1: QUESTIONABLEBOOLASSIGNMENT (FINDBUGS) [76]

```
public class QuestionableBooleanAssignment
 extends BytecodeScanningDetector
 implements StatelessDetector {
 [...]
 @Override
 public void visitCode(Code obj) {
  state = SEEN_NOTHING;
  super.visitCode(obj); [...]
 }
 @Override
 public void sawOpcode(int seen) {
    [...]
    case SEEN_NOTHING:
     if ((seen == ICONST_1) ||
         (seen == ICONST_0))
      state = SEEN_ICONST_0_OR_1;
     break;
    case SEEN_ICONST_0_OR_1: [...]
    case SEEN_DUP: [...]
    case SEEN_ISTORE: [...]
      bug = new BugInstance([...]); [...]
    case SEEN_IF:
     state = SEEN_NOTHING; [...]
     String cName = getClassConstantOperand();
     [...]
     bugReporter.reportBug(bug);
     break;
    case SEEN_GOTO: [...]
  }
 }
}
```

a finite state machine using a switch statement. On one hand, one can appreciate that using a visitor pattern together with good library support results in a scheme that is flexible and powerful. On the other hand, one can also appreciate that writing a custom rule using this approach is not trivial. Programmers must have a good command of bytecode analysis – which we argue is not the case of many users.

Using CQE We show the same rule implemented in CQE (listing 6.2). The rule is defined as the disjunction of the following two cases: we look for suspicious statements, that are either *if* statements or *while* statements – therefore we have two query methods. The rule will match all *if* and *while* statements with the code pattern `someBool=true` as the boolean comparison, irrespectively of the inner statements (as mentioned previously, $a()$ will match any sequence of statements). One can very easily extend this rule for the code pattern `someBool=false` using two more query methods, one for *if* statements, the other one for *while* statements.

LISTING 6.2: QUESTIONABLEBOOLEANASSIGNMENT (CQE)

```
class FirstExample {
  [Query(QuestionableBooleanAssignment)]
  void CaseIf(bool b, Action a){
    if(b = true) a();
  }
  [Query(QuestionableBooleanAssignment)]
  void CaseWhile(bool b, Action a){
    while(b = true) a();
  }
}
```

6.5.3. Second example

For our second example, we chose an imperfection. We are concerned with empty branches of conditionals, and empty loop bodies: `while(b){ }`, `if(b){ }`

Using Semmle The public documentation of Semmle [180] gives an implementation of a rule that can detect this flaw, which we slightly modified to make it oblivious to comments, see listing 6.3. Note the constraints in the constructor of the class `EmptyBlock`, that refine the definition of `Block`. Note also the SQL inspired *from-select* clause.

Using CQE We proceed similarly as for the first example: we have two disjunct cases, `EmptyConditional` and `EmptyWhile`, see listing 6.4.

The query `EmptyConditionalOrEmptyLoop` looks for code patterns that have either the form described by `EmptyConditional`, or the form described by `EmptyWhile` .

LISTING 6.3: EMPTYBLOCKS RULE (SEMMLE) [180]

```
class EmptyBlock extends Block {
 EmptyBlock() { this.getNumStmt() = 0 }
}
class BlockParent extends Stmt {
 BlockParent() {
  this instanceof IfStmt or
  this instanceof LoopStmt
 }
}
from BlockParent s
select (EmptyBlock)s.getAChild(), "Empty block"
```

LISTING 6.4: EMPTY CONDITIONALS, EMPTY LOOPS (CQE)

```
class SecondExample {
  [Query(EmptyConditionalOrEmptyLoop)]
  void EmptyConditional(Func<bool> b) {
    if(b()) { }
  }
  [Query(EmptyConditionalOrEmptyLoop)]
  void EmptyWhile(Func<bool> b) {
    while(b()) { }
  }
}
```

6.5.4. Third example

Our third example is performance related. Using the collection library of the .Net framework, the cost of checking if a list contains an object is proportional to the size of the list, $O(N)$. For large lists and frequent calls to `Contains()`, programmers should consider using the class `HashSet<T>` , for which calls to `Contains()` are expected to take constant time.

Using NDepend Custom rules in NDepend are written using a dedicated language, also inspired by SQL. Nonetheless, there are a number of core differences with SQL: for example, there are no variables nor nested queries. In short, the tool trades expressiveness of the language for ease of use. The strength of the framework comes from convenient pre-programmed primitives such as `IsDirectlyUsing`. The query part returns a set of types, namespace or methods. The rule, taken from the documentation [147] is shown in listing 6.5. (Note that the rule does not return a set of statements but a set of methods.)

Using CQE The same rule, expressed in CQE is slightly more verbose (see listing 6.6). In the NDepend listing, the method is mentioned as a string, so type checking must supported explicitly by the tool. In the case of CQE, it is a real method call: any compiler will refuse to compile the rule if the programmer, for example, mis-spelled `Contains` for `Contain`. Note that the method call will never be executed, it is just there as an example – a code query example.

LISTING 6.5: PERFORMANCE RULE (NDEPEND) [147]

```
SELECT METHODS WHERE
IsDirectlyUsing "List<T>.Contains(T)" OR
IsDirectlyUsing "IList<T>.Contains(T)" OR
IsDirectlyUsing "ArrayList.Contains(Object)"
```

LISTING 6.6: PERFORMANCE RULE (CQE)

```
class ThirdExample {
  [Query(PotentiallySlow)]
  void ContainsIList<T>(IList<T> a, T e) {
    a.Contains(e);
  }
  [Query(PotentiallySlow)]
  void ContainsArray<T>(ArrayList a, T e) {
    a.Contains(e);
  }
}
```

6.5.5. Discussion regarding custom rules

Tools that have a procedural approach Concerning the rule checking tools that have a procedural approach, we only showed an example from FindBugs. Nonetheless, all of them share basically the same approach: the use of a visitor pattern, together with good library support that provides the user with a lot of pre-programmed functionalities. The approach is very flexible but can also quickly turn into a very technical exercise.

Tools that have a declarative approach NDepend, Semmle, and CQE that we introduced, have a declarative approach. NDepend and Semmle draw their simplicity from the of use a dedicated language inspired by SQL (which is well-known by most developers), whereas our approach draws it simplicity from the close resemblance between the code query and the target code. (Semmle does not stop at this SQL-like language, but also let users define new data types.) Semmle and NDepend have extensive support for metrics, whereas we do not address this aspect at all.

Limitations of CQE On one hand it is relatively easy to express custom rules using CQE; on the other hand, the language is also currently very limited. For example, it is not clear how to express style rules, dependency rules, or code complexity rules using CQE. One can easily find a large number of existing rules that cannot currently be expressed with this language. In this respect, it currently does not have the required industrial strength to be used in practice, neither does it pretend in its actual state to compete with the flexibility of the other tools.

More than just code Even if we mostly focused our discussion on pure coding aspect of custom rules, the added-value of these tools do not stop there. Among other things, flexible configuration capabilities, and convenient visualization of positives are very important. Integrations with development environments and build systems are also very much appreciated by the users.

6.6. Related work

Rule categorization There does not seem to be a general agreement on how to classify rules, an observation also made by Ayewah [16]. Tools typically provide their own categorization of rules. Unfortunately these categorizations can differ a lot from one tool to another, making comparison difficult. This is further made difficult by the fact that tools tend to use platform specific categories – we tried to make our categorization as platform neutral as possible. Existing categorizations [175] also tend to mix thematic categories with categories based on the effect of flaws ("critical defect", etc.) or the likelihood to have a true positive ("Most likely a defect", etc.), whereas we enforced a purely thematic categorization.Also, contrary to many existing categorizations, we tried to make our categorization complete, in the sense that no "miscellaneous" category is present.

Custom rules Originally, the first and the last authors proposed the concept of code query by example for parallelization on modern hardware [56], and for the customization of large enterprise systems [201]. To the best of our knowledge, no language closely related to CQE has been proposed previously for rule checking. Nonetheless, from a more general perspective, code quantification has been approached many times before: for example in the context of aspect-oriented languages to denote pointcuts [110]. Versioning systems and text based search tools such as diff and grep [121] can be seen as some related, but distant work, since they work on a purely textual level and we propose an approach that is both typed and that ignores code details such as code comments or indentation.

6.7. Conclusions

We made the dichotomy between default rules and custom rules, two important elements of modern lightweight static analysis tools. We addressed a problem with each of them, and discussed the current limitations of our proposals.

Regarding default rules, we proposed to tackle the overflow of default rules with a thematic categorization. This categorization was inferred from the default rules of five tools. In short, we addressed the problem of *describing* default rules.

Regarding custom rules, we outlined that expressing custom rules is challenging using the current tools. We proposed a new language based on the concept of code query by example to ease this process. In short, we addressed the problem of *prescribing* custom rules.

Acknowledgments

Many of the initial ideas presented here were developed during a stay at University of Pisa by the last author. This stay was made possible thanks to a grant from the

FIRST research school. Thanks to Peter Sestoft and Estelle Barbot for their help and support.

Bibliography

[1] ABADI, M., AND CARDELLI, L. *A Theory of Objects*. Springer-Verlag, 1996. 19, 20, 21, 187

[2] ABADI, M., AND LAMPORT, L. Composing specification. *ACM Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 73–132. 26

[3] ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985. 19, 79, 104

[4] ABRIAL, J.-R. Formal methods in industry: achievements, problems, future. 761–768. 7, 78

[5] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. 53, 72, 73, 74

[6] ALLEN, E. Object-oriented programming in Fortress. Invited talk at FOOL/-WOOD 2007, January 2007. At `http://www.cs.hmc.edu/~stone/FOOL/FOOLWOOD07/Allen-slides.pdf`. 112

[7] ALLEN, E., ET AL. The Fortress language specification. Tech. rep., Sun Microsystems, March 2008. At `http://research.sun.com/projects/plrg`. 112

[8] ALLOY. Alloy project web page. `http://alloy.mit.edu`. 86

[9] AMRRESEARCH. Amr research web page. `http://www.amrresearch.com`. 6, 97

[10] ANT. Ant project. `http://ant.apache.org`. 95

[11] APPEL, A. W. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1997. 72, 73, 74

[12] APT. APT package management. `http://wiki.debian.org/Apt`. 95

[13] ASPECTDNG. AspectDNG web site. `http://aspectdng.tigris.org`. 204

[14] ASPECTJ. AspectJ web site. `http://www.eclipse.org/aspectj`. 181

[15] AXELROD, R. *The evolution of co-operation*. Penguin science politics. 94

[16] AYEWAH, N., PUGH, W., MORGENTHALER, J. D., PENIX, J., AND ZHOU, Y. Evaluating static analysis defect warnings on production software. In *PASTE* (2007), pp. 1–8. 223

[17] BAEZA-YATES, R., AND RIBEIRO-NETO, B. *Modern Information Retrieval*. Addison-Wesley, 1999. 181

[18] BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)* (New York, NY, 2005), G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 49–69. 20, 187

[19] BATORY, D. Feature oriented programming for product-lines. Slide set for tutorial, OOPSLA'04, Vancouver, Canada, October 2004. 139

[20] BATORY, D. Multilevel models in model-driven engineering, product lines, and metaprogramming. *IBM Systems Journal 45*, 3 (July 2006), 527–539. 104, 138, 139, 143

[21] BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. JTS, tools for implementing domain specific languages. In *Fifth International Conference on Software Reuse* (1998), pp. 143–153. 138

[22] BATORY, D., AND O'MALLEY, S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology 1*, 4 (1992), 355–398. 138

[23] BATORY, D., SINGHAL, V., SIRKIN, M., AND THOMAS, J. Scalable software libraries. In *SIGSOFT* (1993), pp. 191–199. 138

[24] BEATTY, R. C., AND WILLIAMS, C. D. ERP II: best practices for successfully implementing an ERP upgrade. *Commun. ACM 49*, 3 (2006), 105–109. 6, 11

[25] BERGEL, A., DUCASSE, S., AND NIERSTRASZ, O. Analyzing module diversity. *J.UCS: Journal of Universal Computer Science 11*, 10 (2005), 1613–1644. 27

[26] BERGEL, A., DUCASSE, S., NIERSTRASZ, O., AND WUYTS, R. Stateful traits and their formalization. *Computer Languages, Systems & Structures 34*, 2-3 (2008), 83–108. 134

[27] BEYONDCOMPARE. Beyond compare web page. `http://www.scootersoftware.com`. 84, 205

[28] BILLE, P. A survey on tree edit distance and related problems. *Theoretical Computer Science 337*, 1-3 (2005), 217–239. 185

[29] BOHANNON, A., PIERCE, B. C., AND VAUGHAN, J. A. Relational lenses: a language for updatable views. In *PODS* (2006), pp. 338–347. 102

[30] BOOST. Boost project web page. `http://www.boost.org`. 28

[31] BREHM, L., HEINZL, A., AND MARKUS, M. L. Tailoring ERP systems: A spectrum of choices and their implications. In *34th Hawaii International Conference on System Sciences* (2001), pp. 1–9. 146

[32] BRUCE, K. B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002. 19, 20, 21, 80, 81

[33] BUSCHER, M., GILL, S., MOGENSEN, P., AND SHAPIRO, D. Landscapes of practice: Bricolage as a method for situated design. *In: JCSCW 10* (2001), 1–28. 165

[34] C2. Premature generalization. C2.com wiki. `http://c2.com/cgi/wiki?PrematureGeneralization`. 94

[35] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17*, 4 (Dec. 1985), 471–522. 20

[36] CASTAGNA, G. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems 17*, 3 (May 1995), 431–447. 80, 81, 104, 187

[37] CASTLEPROJECT. Castle project web site. `http://www.castleproject.org`. 100, 205

[38] CISTERNINO, A., RASMUSSEN, R., AND VAUCOULEUR, S. Describing default rules, prescribing custom rules (companion document). Tech. rep., IT University of Copenhagen, 2009. xviii, 207

[39] CISTERNINO, A., RASMUSSEN, R., AND VAUCOULEUR, S. Describing default rules, prescribing custom rules (companion document). Tech. rep., IT University of Copenhagen, 2009. `http://www.itu.dk/people/vaucouleur/rules.pdf`. 192, 211

[40] CISTERNINO, A., AND VAUCOULEUR, S. Aspect oriented programming made easy: An embedded pointcut language (submitted for publication). Tech. rep., IT University of Copenhagen, 2009. xviii, 190

[41] CLIFILERW. CLIFileRW Project. http://www.codeplex.com/clifilerw. 198

[42] CLIFTON, C. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Tech. rep., Iowa State University, 2001. 101

[43] COCCINELLE. The coccinelle project web page. `http://www.emn.fr/x-info/coccinelle`. 205

[44] COHEN, T., GIL, J., AND MAMAN, I. JTL: the java tools language. In *OOPSLA* (2006), P. L. Tarr and W. R. Cook, Eds., ACM, pp. 89–108. 99, 205

[45] COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. *Version Control with Subversion*. O'Reilly & Associates, Inc., 2004. For version 1.0. 68, 82, 83, 104, 205

[46] CONRADI, R., AND WESTFECHTEL, B. Version models for software configuration management. *ACM Computing Surveys 30*, 2 (1998), 232–282. 68, 78, 82, 83, 102

[47] CWALINA, K., AND ABRAMS, B. *Framework design guidelines: conventions, idioms, and patterns for reusable .Net libraries*. Addison-Wesley Professional, 2008. 21, 78, 89, 90, 93, 96

[48] CZARNECKI, K., AND EISENECKER, U. W. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 101, 102

[49] D'AMBROS, M., GALL, H., LANZA, M., AND PINZGER, M. Analysing software repositories to understand software evolution. In *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 37–67. 98

[50] DARCS. Darcs web page. `http://darcs.net`. 85

[51] DARCS. Understanding darcs, patch theory. Open-content textbooks collection. `http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory`. 85

[52] DE MOOR, O., SERENI, D., VERBAERE, M., HAJIYEV, E., AVGUSTINOV, P., EKMAN, T., ONGKINGCO, N., AND TIBBLE, J. .QL: Object-oriented queries made easy. 78–133. 209

[53] DIJKSTRA, E. W. Notes on structured programming. Tech. Rep. 70-WSK-03, Technological University Eindhoven, Apr. 1970. 95, 96

[54] DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976. 95

[55] DINGSØYR, T., AND CONRADI, R. A survey of case studies of the use of knowledge management in software engineering. *International Journal of Software Engineering and Knowledge Engineering 12*, 4 (2002), 391–414. 165

[56] DITTAMO, C., CISTERNINO, A., AND DANELUTTO, M. Parallelization of C# programs through annotations. In *International Conference on Computational Science* (2007), pp. 585–592. 223

[57] DITTRICH, Y., AND VAUCOULEUR, S. Customizing and upgrading ERP systems: a reality check. Tech. Rep. TR2008-105, IT University of Copenhagen, 2008. xviii, 145, 172

[58] DITTRICH, Y., AND VAUCOULEUR, S. Practices around customization of standard systems. In *Cooperative and Human Aspects of Software Engineering, ICSE* (2008). xviii, 114, 145, 172, 191

[59] DITTRICH, Y., VAUCOULEUR, S., AND GIFF, S. ERP customization as software engineering. knowledge sharing and cooperation when adapting a large existing code base. *IEEE Software* (2009). xviii, 145

[60] DOWEK, G. Higher-order unification and matching. 1009–1062. 101, 206

[61] DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. P. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems 28*, 2 (2006), 331–388. 104, 132, 133, 143

[62] DYNAMICSUSER. Dynamics user group: post the dynamics community. `http://dynamicsuser.net/forums/t/1184.aspx`. 97

[63] EADDY, M., AND AHO, A. Statement annotations for fine-grained advising. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2006), Nantes, France, July 2006* (2006). 134

[64] ECLIPSE. Eclipse web page. `http://www.eclipse.org`. 24, 78, 96

[65] ECMA. *Ecma-334: C# Language Specification*, 4th ed. Ecma International, June 2006. 18, 19, 20, 21, 31, 40, 58, 68, 69, 70, 73, 80, 87, 104, 129, 143, 173, 175, 178, 179, 180, 184, 191, 193, 194, 199, 203, 217

[66] ECMA. *Ecma-335: Common Language Infrastructure (CLI)*, 4th ed. Ecma International, June 2006. 18, 21, 22, 23, 31, 56, 175, 178, 179, 184, 191, 194, 204, 217

[67] EGGTHER. Eggther prototype. `http://www.itu.dk/people/vaucouleur/eggther.zip`. 52, 205

[68] EISENBACH, S., JURISIC, V., AND SADLER, C. Managing the evolution of.NET programs. In *FMOODS* (2003), E. Najm, U. Nestmann, and P. Stevens, Eds., vol. 2884 of *Lecture Notes in Computer Science*, Springer, pp. 185–198. 86

[69] EOS. Eos web page. `http://ect.jate.hu`. 205

[70] ERICKSON, J., AND SCOTT, J. E. The relationship of between outsourcing and knowledge transfer for ERP upgrade project. In *PRE-ICIS workshop on enterprise systems research in MIS* (2007). 99

[71] ESP. Evolvable software products project web page. `http://www.itu.dk/research/esp`. 97

[72] FERNÁNDEZ-RAMIL, J., LOZANO, A., WERMELINGER, M., AND CAPILUPPI, A. Empirical studies of open source evolution. In *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 263–288. 98

[73] FILMAN, R. E. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns, ECOOP* (2001). 25

[74] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M., Eds. *Aspect-Oriented Software Development*. Addison-Wesley, 2005. 25, 30, 102, 181, 191, 193, 203

[75] FILMAN, R. E., AND FRIEDMAN, D. P. *Aspect-Oriented Programming Is Quantification and Obliviousness*. Addison-Wesley, Boston, 2005, pp. 21–35. 25, 29, 181

[76] FINDBUGS. `http://findbugs.sourceforge.net`. xvi, 208, 210, 218, 219

[77] FOWLER. Domain specific languages reference web page. `http://martinfowler.com/dslwip`. 29, 95

[78] FOWLER, M. Continuous integration. Web article. `http://martinfowler.com/articles/continuousIntegration.html`. 95

[79] FOWLER, M. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`. 90, 104

[80] FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison Wesley, Reading, Massachusetts, Nov. 2002. 195

[81] FRIEDL, J. *Mastering regular expressions*. O'Reilly, 2008. 56, 57, 60

[82] FXCOP. `http://code.msdn.microsoft.com/codeanalysis`. 210

[83] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. xv, 86, 87, 88, 89, 91, 92, 93, 124, 125, 126, 184, 218

[84] GARTNER. Gartner home page. `http://www.gartner.com`. 6, 97

[85] GILBERT, D. *Stumbling on happiness*. Vintage, 2006. 14

[86] GIT. Git web page. `http://git-scm.com`. 85

[87] GOUGH, J. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, 2002. 21, 22, 23, 53, 56, 59

[88] GREEF, A., ET AL. *Inside Microsoft Dynamics AX 4.0*. Microsoft Press, 2006. 11, 13, 18, 66, 78, 104, 116, 117, 143, 147, 168

[89] GROUP, J.-. E. Jsr-277: Java module system. Tech. rep., Sun Microsystems, October 2006. At `http://jcp.org/en/jsr/detail?id=277`. 112

[90] HANSEN, P. B. Structured multiprogramming. *Commun. ACM 15*, 7 (1972), 574–578. 186

[91] HANSON, E. Improving performance with sql server 2005 indexed views. Microsoft technet documentation, 2005. `http://technet.microsoft.com/en-us/library/cc917715.aspx`. 95

[92] HATCLIFF, J., LEAVENS, G. T., LEINO, K. R. M., MÜLLER, P., AND PARKINSON, M. Behavioral interface specification languages. Tech. Rep. CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, Mar. 2009. 26, 27, 93, 94, 102

[93] HERLIHY, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys. 15*, 1 (Mar. 1990), 96. 82

[94] HEVNER, A. R., MARCH, S. T., PARK, J., AND RAM, S. Design science in information systems research. *MIS Quarterly 28*, 1 (2004). 8, 9

[95] HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580. 26, 186

[96] HOARE, T. The science of computing and the engineering of software, 2009. `http://www.infoq.com/presentations/tony-hoare-computing-engineering`. 7, 104

[97] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Notices 39*, 12 (2004), 92–106. 209, 218

[98] HRUBY, P., AND SCHELLER, C. V. Understanding accounting from the rea perspective. In *3rd REA Technology Workshop* (2008). 10, 99

[99] HUDAK, P. Building domain-specific embedded languages. *ACM Comput. Surv. 28*, 4es (1996), 196. 29

[100] HYPERJ. Home page. At `http://www.alphaworks.ibm.com/tech/hyperj`. 140

[101] JEFFRIES, R. Cost of anticipatory design. Web article. http://www.xprogramming.com/xpmag/cost_of_antici.htm. 93

[102] JIANG, WANG, AND ZHANG. Alignment of trees – an alternative to tree edit. *TCS: Theoretical Computer Science 143* (1995). 98

[103] JIRA. Jira project. http://www.atlassian.com/software/jira. 95

[104] JOHANSEN, R., SESTOFT, P., AND SPANGENBERG, S. Zero-overhead composable aspects for .NET. In *Advances in Software Technology* (2008), E. Börger and A. Cisternino, Eds., vol. 5316 of *LNCS*. 104, 118, 134, 137, 143, 144, 201, 204

[105] JOHANSEN, R., AND SPANGENBERG, S. Yiihaw. an aspect weaver for .NET. Master's thesis, IT University of Copenhagen, Denmark, February 2007. At http://www.itu.dk/people/sestoft/itu/JohansenSpangenberg-Aspects-2007.pdf. 104, 137, 143

[106] KELLENS, A., MENS, K., BRICHAU, J., AND GYBELS, K. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP* (2006), pp. 501–525. 100

[107] KENNEDY, A., AND RUSSO, C. Generalized algebraic data types and object-oriented programming. In *OOPSLA, October 2005, San Diego, California*, pp. 21–40. 128

[108] KENNEDY, R., CHAN, S., MING LIU, S., LO, R., TU, P., AND CHOW, F. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems 21* (1999), 627–676. 73

[109] KHANNA, S., KUNAL, K., AND PIERCE, B. C. A formal investigation of diff3. 84

[110] KICZALES, G., ET AL. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP), Finland, Lecture Notes in Computer Science 1241* (1997), Springer-Verlag, pp. 220–242. 134, 223

[111] KICZALES, G., ET AL. An overview of AspectJ. In *15th ECOOP, LNCS 2072* (2001), J. L. Knudsen, Ed., pp. 327–353. 134

[112] KIRK, D., ROPER, M., AND WOOD, M. Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering 12*, 3 (2007), 243–274. 163

[113] KOCH, C. Enterprise software upgrades: Less pain, more gain. Web article, November 2002. http://www.cio.com/article/31499/Enterprise_Software_Upgrades_Less_Pain_More_Gain. 6, 11

[114] KOCH, C., AND WAILGUM, T. ERP definition and solutions, 2008. http://www.cio.com/article/40323/ERP_Definition_and_Solutions. 6

[115] KROAH-HARTMAN, G. Linux internal API document. Linux Internal documentation. `http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`. 94

[116] KRUEGER, C. W. Software reuse. *ACM Computing Surveys 24*, 2 (June 1992), 131–183. 24, 102

[117] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (Mar. 1977), 125–143. 26

[118] LAMPORT, L. A simple approach to specifying concurrent systems. 32–45. 26

[119] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J., CHALIN, P., AND ZIMMERMAN, D. M. *JML Reference Manual*. May 2008. 187

[120] LOOM. Loom web page. `http://www.dcl.hpi.uni-potsdam.de/research/loom`. 204

[121] MACKENZIE, D., EGGERT, P., AND STALLMAN, R. *Comparing and Merging Files With Gnu Diff and Patch*. 2002. 78, 82, 83, 187, 205, 223

[122] MAF. Managed extensibility framework (System.AddIn). `http://www.codeplex.com/clraddins`. 100

[123] MARCH, S. T., AND SMITH, G. F. Design and natural science research on information technology. *Decis. Support Syst. 15*, 4 (1995), 251–266. 8, 9

[124] MARTIN, M. C., LIVSHITS, V. B., AND LAM, M. S. Finding application errors and security flaws using pql: a program query language. In *OOPSLA* (2005), pp. 365–383. 99, 205

[125] MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003. xv, 86, 88, 89, 90, 91, 93, 94, 95, 104

[126] MCILROY, M. D. Mass produced software components. In *Software Engineering, Garmisch, Germany, 7-11 October 1968* (1969), P. Naur and B. Randell, Eds., NATO Science Committee, pp. 138–155. 118

[127] MEF. Managed extensibility framework web page. `http://www.codeplex.com/MEF`. 64, 97, 100, 200

[128] MEIR M. LEHMAN. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE 68*, 9 (September 1980), 1060–1076. 110, 208

[129] MEIR M. LEHMAN. Rules and tools for software evolution planning and management. *Annals of Software Engineering 11*, 1 (2001), 15–44. 110

[130] MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering 28*, 5 (May 2002), 449–462. 68, 82, 83

[131] MENS, T., BUCKLEY, J., ZENGER, M., AND RASHID, A. Towards a taxonomy of software evolution. In *International Workshop on Unanticipated Software Evolution, Warsaw, Poland* (April 2003). 102, 108, 110

[132] MENS, T., AND DEMEYER, S., Eds. *Software Evolution*. Springer, 2008. 102

[133] MERCURIAL. Mercurial web page. `http://www.selenic.com/mercurial/wiki`. 84

[134] MEYER, B. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. 19, 20, 27, 80, 93, 127

[135] MICROSOFT. Microsoft developer network web site. `http://msdn.microsoft.com`. 18, 19, 20, 21, 24, 53, 55, 56, 57, 58, 60, 61, 64, 65, 68, 69, 72, 74, 75, 85, 96, 100, 104, 191, 200

[136] MICROSOFT. Microsoft Dynamics. Homepage. At http://www.microsoft.com/dynamics. 21, 24, 97

[137] MICROSOFT. Microsoft Dynamics AX. Homepage. At `http://www.microsoft.com/dynamics/ax`. 10, 112

[138] MICROSOFT. Microsoft Dynamics NAV. Homepage. At `http://www.microsoft.com/dynamics/nav`. 10, 112

[139] MICROSOFT. *Application Designer's Guide*. Microsoft Business Solutions, 2006. 147

[140] MICROSOFTDYNAMICSFORUMS. Microsoft dynamics forums. `http://www.microsoftdynamicsforums.com`. 97

[141] MILI, H., MILI, F., AND MILI, A. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering 21*, 6 (June 1995), 528–562. 24, 26

[142] MONO. Homepage. At `http://www.mono-project.com/Mono:Runtime`. 22, 73

[143] MONO. Homepage. At `http://www.mono-project.com/Cecil`. 56, 62, 198

[144] MORTENSEN, F. Software development with Navision. Talk, ERP Crash Course, University of Copenhagen, 2007. At `http://www.3gerp.org/Documents/ERPV.ppt`. 10, 112, 115

[145] MOZILLA. Firefox addon webpage. `https://addons.mozilla.org`. 24

[146] MÜLLER, P., POETZSCH-HEFFTER, A., AND LEAVENS, G. T. Modular invariants for layered object structures, 2006. 187

[147] NDEPEND. http://ndepend.com. xvi, 210, 221

[148] NIERSTRASZ, O., BERGEL, A., DENKER, M., DUCASSE, S., GÄLLI, M., AND WUYTS, R. On the revival of dynamic languages. In *Software Composition* (2005), T. Gschwind, U. Aßmann, and O. Nierstrasz, Eds., vol. 3628 of *Lecture Notes in Computer Science*, Springer, pp. 1–13. 105

[149] NOGUERA, C., AND PAWLAK, R. Open Static Pointcuts Through Source Code Templates. In *International AOSD Workshop on Open and Dynamic Aspect Languages* (2006). 100, 205

[150] NREFACTORY. NREFactory web page. http://www.icsharpcode.net. 54

[151] NUNIT. Nunit project. http://www.nunit.org. 95, 96

[152] ODERSKY, M. The Scala language specification, version 2.0. Tech. rep., Ecole Polytechnique Federale de Lausanne, Switzerland, January 2007. At http://www.scala-lang.org. 104, 133, 143

[153] OFBIZ. Ofbiz web page. http://ofbiz.apache.org. 97

[154] OLSON, D. L. Evaluation of ERP outsourcing. *Computers & OR 34*, 12 (2007), 3715–3724. 99

[155] ORACLE. Oracle e-business suite. http://www.oracle.com/applications/e-business-suite.html. 97

[156] OSSHER, H., AND TARR, P. Multi-dimensional separation of concerns and the hyperspace approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development* (2000). 27, 28, 170

[157] OSSHER, H., AND TARR, P. Hyper/J: multi-dimensional separation of concerns for Java. In *ICSE '01: 23rd International Conference on Software Engineering, Toronto, Canada* (2001), IEEE Computer Society, pp. 821–822. 28, 104, 140, 141, 143

[158] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery 15*, 12 (Dec. 1972), 1053–1058. 26, 27, 93, 94, 127, 191

[159] PARNAS, D. L. On the design and development of program families. *IEEE Transactions on Software Engineering SE-2*, 1 (March 1976). 26, 27, 93, 94, 104, 126, 143

[160] PERLIS, A. J. Epigrams on programming. *SIGPLAN Notices 17*, 9 (1982), 7–13. 128

[161] PIERCE, B. C. *Basic category theory for computer scientists*. The MIT Press, Cambridge, US, 1991. 4

[162] PIETRASZAK, M., AND RUBINSTEIN, B. Microsoft visual sourcesafe roadmap. Msdn documentation, 2004. http://msdn.microsoft.com/en-us/library/aa302175.aspx. 95

[163] PONTOPPIDAN, M. F. Smart customizations. Screen cast, 2006. At http://channel9.msdn.com/Showforum.aspx?forumid=38&tagid=94. 114, 118

[164] POSTSHARP. PostSharp web site. http://www.postsharp.org. 181, 203

[165] PRASANNA, D. R. *Dependency Injection*. Manning, 2009. 90, 100

[166] PREHOFER, C. Feature-oriented programming: A fresh look at objects. In *ECOOP* (1997), pp. 419–443. 138

[167] PRISM. The prism project web page. http://www.codeplex.com/CompositeWPF. 100

[168] QUICKGRAPH. Quickgraph project web page. http://www.codeplex.com/quickgraph. 28

[169] RHIGER, M. Analyzing differences between w1 and gdls using tree alignment, 2009. http://www.itu.dk/research/esp-net/documents/Morten.ppt. 98

[170] ROBERTSON, T. Embodied actions in time and place: the cooperative design of a multimedia, educational computer game. *Comput. Supported Coop. Work 5*, 4 (1996), 341–367. 165

[171] ROBSON, C. *Real World Research*. Blackwell Publishers Ltd, Oxford, UK, 2002. 8, 14, 148, 150

[172] ROGERSON, D. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997. 127

[173] ROOVER, C. D., D'HONDT, T., BRICHAU, J., NOGUERA, C., AND DUCHIEN, L. Behavioral similarity matching using concrete source code templates in logic queries. In *PEPM* (2007), pp. 92–101. 99, 205

[174] RUBY. Ruby doc web page. http://www.ruby-doc.org. 101

[175] RUTAR, N., ALMAZAN, C. B., AND FOSTER, J. S. A comparison of bug finding tools for java. In *ISSRE* (2004), pp. 245–256. 223

[176] SAFONOV, V., AND GRIGORYEV, D. Aspect.net – aspect-oriented programming for .net in practice. *.Net Developers Journal* (2005). http://www.aspectdotnet.org/articles/Aspect_NET_Pilsen_2006.pdf. 205

[177] SAP. Sap home page. http://www.sap.com. 21, 97

[178] SCHMIDT, G., AND STRÖHLEIN, T. *Relations and Graphs - Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer, 1993. 28

[179] SCHNEIDER, F. B. Decomposing properties into safety and liveness. Technical Report TR87-874, Cornell University, Computer Science Department, Oct. 1987. 26

[180] SEMMLE. `http://semmle.com`. xvi, 210, 220

[181] SESTOFT, P. ML pattern match compilation and partial evaluation. *Lecture Notes in Computer Science 1110* (1996), 446–?? 101, 206

[182] SESTOFT, P., AND VAUCOULEUR, S. Technologies for evolvable software products: The conflict between customizations and evolution. In *Advances in Software Technology* (2008), E. Börger and A. Cisternino, Eds., vol. 5316 of *LNCS*, Springer. xviii, 107, 165, 168, 169, 170, 188, 191, 201

[183] SHANKS, G., SEDDON, P. B., AND WILLCOCKS, L. P. *Second-wave Enterprise Resource Planning Systems*. Cambrige Press, 2003. 10, 11, 18, 168

[184] SILBERSCHATZ. *Database systems*. 1997. 99, 205

[185] SOFTWARE ENGINEERING INSTITUTE. Software product lines. Web site. At `http://www.sei.cmu.edu/productlines`. 108

[186] SPOON. Spoon web page. `http://spoon.gforge.inria.fr`. 99, 205

[187] STÖRZER, M., AND GRAF, J. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM* (2005), pp. 653–656. 100

[188] STROUSTRUP, B. *The C++ programming language*. Addison-Wesley, 2000. 128

[189] STUCKENHOLZ, A. Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes 30*, 1 (2005), 7. 86, 93

[190] STUDEBAKER, D. *Programming Microsoft Dynamics NAV*. Packt Publishing, 2007. 11, 13, 168

[191] STYLECOP. `http://code.msdn.microsoft.com/sourceanalysis`. 210

[192] SUCHMAN, L. Working relations of technology production and use. *Computer Supported Cooperative Work (CSCW) 2* (1994). 166

[193] SYME, D. Leveraging.NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML* (2006), A. Kennedy and F. Pottier, Eds., ACM, pp. 43–54. 105

[194] TORGERSEN, M. New features in c# 4.0. Tech. rep., 2009. 81

[195] TOURWE, T., BRICHAU, J., AND GYBELS, K. On the existence of the AOSD-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA* (2003). 137

[196] ULLMAN, J. D. *Database and Knowledge-Base Systems, Volumes I and II*. Computer Science Press, 1989. 217

[197] UNGAR, D., AND SMITH, R. B. SELF: The power of simplicity. *Lisp and Symbolic Computation 4*, 3 (1991), 187–205. 18

[198] UNITY. The unity project web page. `http://www.codeplex.com/unity`. 100

[199] UNPHON, H., AND DITTRICH, Y. Organisation matters: How the organisation of software development influences the development of product line architecture. In *IASTED Intl. Conf. on Software Engineering, Innsbruck, Austria* (2008). 99

[200] VAUCOULEUR, S. Beyond the crystal ball assumption: Towards upgradable ERP systems. In *3GERP workshop* (2008). xviii, 191, 201

[201] VAUCOULEUR, S. Customizable and upgradable enterprise systems without the crystal ball assumption. *IEEE International EDOC Conference* (2009). xviii, 167, 192, 223

[202] VAUCOULEUR, S., AND EUGSTER, P. Atomic features. In *OOPSLA Workshop on Synchronization and concurrency in object-oriented languages (SCOOL)* (2005). 186

[203] VERBAERE, M., GODFREY, M. W., AND GÎRBA, T. Query technologies and applications for program comprehension. In *ICPC* (2008), pp. 285–288. 209

[204] VISSER, J. Matching objects without language extension. *Journal of Object Technology 5*, 8 (2006). xiii, 33, 37, 53, 54

[205] WEI, H.-L., WANG, E. T. G., AND JU, P.-H. Understanding misalignment and cascading change of erp implementation: a stage view of process analysis. *Eur. J. Inf. Syst. 14*, 4 (2005), 324–334. 168

[206] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (Aug. 1984), 352–357. 82

[207] WENGER, E. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press, 1998. 164

[208] WICCA. Wicca web page. `http://www1.cs.columbia.edu/~eaddy/wicca`. 204

[209] WING, J. M. Five deep questions in computing. *Commun. ACM 51*, 1 (2008), 58–60. `http://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/Wing08.pdf`. 3

[210] WINMERGE. Winmerge web page. `http://winmerge.org`. 84, 205

[211] WĄSOWSKI, A. *Code Generation and Model Driven Development for Constrained Embedded Software*. PhD thesis, IT University of Copenhagen, Denmark, 2005. At `http://www.itu.dk/~wasowski/papers/wasowski-dissertation-20050909.pdf`. 78, 101

[212] WOLFINGER, R., AND PRÄHOFER, H. Integration models in a .Net plug-in framework. In *Software Engineering* (2007), W.-G. Bleek, J. Raasch, and H. Züllighoven, Eds., vol. 105 of *LNI*, GI, pp. 217–230. 65

[213] ZLOOF, M. M. Query by example. In *AFIPS '75: Proceedings of the May 19-22, 1975, national computer conference and exposition* (New York, NY, USA, 1975), ACM, pp. 431–438. 217