View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

EXPLOITING SIMILARITY PATTERNS IN WEB APPLICATIONS FOR ENHANCED GENERICITY AND MAINTAINABILITY

DAMITH CHATURA RAJAPAKSE

(BSc.Eng (Hons), SL)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

Acknowledgments

My profound thanks are due to the following persons.

- My advisor A/P Stan Jarzabek, for the innumerable ways in which he made this thesis possible, and for guiding me with boundless patience, never shying away when help was needed.
- Members of my thesis committee A/P Dong Jin Song and A/P Khoo Siau Cheng for their valuable advice throughout this journey of four years, and for spending their valuable time in various administration tasks related to my candidature.
- Collaborators, advisors, and evaluators who gave feedback about my research: Dr. Bimlesh Wadhwa, Dr. Irene Woon, and Prof Kim Hee-Woong (NUS), Prof. Andrea De Lucia and Dr. Giuseppe Scanniello (Università di Salerno, Italy), Prof. Katsuro Inoue, Dr. Shinji Kusumoto, and Higo Yoshiki (Osaka Uni. Japan), Dr. Toshihiro Kamiya (PRESTO, Japan), Sidath Dissanayake (SriLogic Pvt Ltd, Sri Lanka), Ulf Pettersson (STE Eng Pte Ltd., Singapore), Yeo Ann Kian, Lai Zit Seng, and Chan Chee Heng (NUS), Prof. Athula Ginige (UWS, Sydney), Prof. San Murugesan (Southern Cross University, Australia).
- My colleagues at NUS, Hamid Abdul Basit, Upali Sathyajith Kohomban, Vu Tung Lam, Sun Jun, Yuan Fang, David Lo, and Sridhar KN in particular, for the comradeship during the last four years.
- Other friends at NUS, and back home in Sri Lanka (whom I shall not name for the fear of missing out one), for lightening my PhD years with your companionship.
- Various colleagues and students who took part in my experiments, Pavel Korshunov, Fok Yew Hoe, Li Meixuan, Anup Chan Poudyal and Tiana Ranaivojoelina in particular.

- Madam Loo Line Fong and others in the graduate office, and system admin Bernard Tay for taking care of various admin matters related to my candidacy.
- Anonymous examiners for their valuable comments, advice and very encouraging feedback on the thesis.
- My parents and sister for being there for me at good and bad times.
- Most of all, my wife Pradeepika who was a pillar of strength at every step of the way. Her boundless love, encouragement and assistance simply defy description.

Table of Contents

ACKNOWLEDGMENTS II

SUMMARY…… VI

LIST OF TABLES. 1

LIST OF FIGURES 2

CHAPTER 1. INTRODUCTION 6

1.1. The problem 6

1.2. Thesis objectives 7

1.3. Thesis scope 7

1.4. Research and contributions 8

1.5. Experimental methods 12

1.6. Thesis roadmap 12

1.7. Research outcomes 14

CHAPTER 2. BACKGROUND AND RELATED WORK 15

2.1. Clones 16

2.1.1. Simple clones 16

2.1.2. Structural clones 17

2.1.3. Reasons for clones 18

2.1.4. Effects of clones 21

2.1.5. Clone detection 23

2.1.6. Clone taxonomies 24

2.2. Clone management 24

2.2.1. Preventive clone management 24

2.2.2. Corrective clone management 27

2.2.3. Compensatory clone management 29

2.2.4. Practical challenges in clone management 30

2.3. An overview of web application domain ... 35

2.3.1. Web applications ... 35

2.3.2. Web technologies ... 37

2.4. Web engineering Vs software engineering ... 45

2.5. Cloning in the web application domain ... 48

2.6. Chapter conclusions ... 49

CHAPTER 3. AN INVESTIGATION OF CLONING IN WEB APPLICATIONS ... 51

3.1. Experimental method ... 52

3.2. Overall cloning level ... 56

3.3. Cloning level in WAs Vs cloning level in traditional applications ... 61

3.4. Factors that affect the cloning level ... 62

3.5. Identifying the source of clones ... 63

3.6. Chapter conclusions ... 65

CHAPTER 4. MORE EVIDENCE OF TENACIOUS CLONES ... 66

4.1. Case study 1: Java Buffer library ... 67

4.2. Case study 2: Standard Template Library ... 70

4.3. Examples of tenacious clones ... 71

4.4. Chapter conclusions ... 77

CHAPTER 5. MIXED-STRATEGY ... 78

5.1. Introduction to XVCL ... 79

5.2. Overview of mixed-strategy ... 83

5.3. Benefits and drawbacks of mixed-strategy ... 84

5.4. Mixed-strategy success stories ... 86

5.5. Mixed-strategy and tenacious clones ... 86

5.6. Why choose mixed-strategy? ... 87

5.7. Chapter conclusions ... 88

CHAPTER 6. UNIFICATION TRADE-OFFS........89

6.1. Case study: Project Collaboration Environment........90

6.1.1. Project Collaboration Environment (PCE)........91

6.1.2. Experimental method........93

6.1.3. PCEsimple........96

6.1.4. PCEpatterns........97

6.1.5. PCEunified........100

6.1.6. PCEms........101

6.1.7. Overall comparison........102

6.1.8. PCE on other platforms........105

6.2. Trade-off analysis........106

6.2.1. Performance........107

6.2.2. Rapid prototyping/evolution capabilities........108

6.2.3. Framework conformance........110

6.2.4. Tidiness in source distribution........111

6.2.5. Indexing by search engines........111

6.2.6. WYSIWYG editing........112

6.2.7. Difference in runtime structure........114

6.3. Discussion of results........115

6.4. Chapter conclusions........117

CHAPTER 7. STRUCTURAL CLONES........118

7.1. Some examples of structural clones........119

7.1.1. Example 1: a file-level structural clone........119

7.1.2. Example 2: a module-level structural clone........120

7.1.3. Example 3: multiple structural clones in the same file........122

7.1.4. Example 4: crosscutting structural clones........122

7.1.5. Example 5: heterogeneous entity structural clones........123

7.1.6. Example 6: structural clones based on inheritance hierarchy........124

7.1.7. Example 7: a structural clone spanning multiple layers........125

7.2. Structural clones and clone management........125

7.2.1. Fragmentation of structural clones........125

7.2.2. Clone fragmentation in web domain........127

7.2.3. Structural clones as 'configurations of lower level clones'........127

7.2.4. A Complete example: structural clones in Adventure Builder........128

7.3. Chapter conclusions........136

CHAPTER 8. SUM: STRUCTURAL CLONE MANAGEMENT USING MIXED-STRATEGY 137

8.1. Clone management using mixed-strategy 139

8.2. Pre-unification activities 143

8.2.1. Clone identification 143

8.2.2. Clone analysis 144

8.2.3. Choosing the unification technique 146

8.2.4. Clone harmonization 147

8.3. Unifying clones using SuM 148

8.3.1. Representing an SCC with the master 148

8.3.2. Unification activities 149

8.3.3. Bottom level – unifying simple clones 152

8.3.4. Building the hierarchy – unifying structural clones 153

8.3.5. Unification root 155

8.3.6. Aligning the solution along SC boundaries 156

8.3.7. Improving the quality of SC harvesting 157

8.4. Post-unification activities 157

8.4.1. Understanding mixed-strategy solutions 157

8.4.2. Maintenance of mixed-strategy solutions 158

8.4.3. Reuse within mixed-strategy applications 161

8.5. Applying SuM to Adventure Builder 161

8.6. Conquering the diversity of structural clones 164

8.6.1. Diversity in structural clones 164

8.6.2. Basic entity types 166

8.6.3. Basic structure types 167

8.7. Basic SuM unification schemes 171

8.7.1. Extra entity 172

8.7.2. Optional entity 173

8.7.3. Parametric entity 174

8.7.4. Alternative entity 175

8.7.5. Repetitive entity 176

8.7.6. Replaceable entity 177

8.7.7. Reordered entity 179

8.7.8. Using basic SuM schemes 180

8.7.9. Benefits of Basic SuM schemes 181

8.7.10. Basic SuM schemes in Adventure Builder 182

8.8. Chapter conclusions 184

CHAPTER 9. CONCLUSIONS AND FUTURE WORK 186

BIBLIOGRAPHY 190

APPENDIX A: ESSENTIAL XVCL SYNTAX 210

Summary

Similarities at analysis, design and implementation levels in software are great opportunities for reuse. When such similarities are not exploited, they can lead to repetitions in software (also called ‘clones’). Most clones negatively affect software maintenance, but clones may also have benefits. We believe that the lack of a holistic approach to unify and reuse clones without losing their benefits is behind the high levels of cloning in today’s software.

In this thesis we concentrate on the cloning problem in web application domain. Using an extensive study of existing web applications, we show that while cloning is common in both traditional and web applications, it is relatively more severe in web applications. This study also produced a framework of metrics for comparing the cloning characteristics of applications.

We use the term ‘clone management’ to describe a holistic approach to counter negative effects of clones (notably on maintainability), while preserving and leveraging their positive aspects (notably their reuse potential). In this thesis we attempt to overcome two challenges in clone management in general, and in the web application domain in particular.

1) Tenacious clones – i.e., some clones are difficult to unify, given the capabilities of the chosen implementation technology, and given the other design goals of the software:

    a. Sometimes unification is just not technically feasible. We call these ‘non-unifiable clones’.

    b. In other cases, unification is hindered due to trade-off caused by clone unification. We call these trade-offs ‘unification trade-offs’.

    c. Some clones are meant to remain in software, because they have been created to serve a purpose. We call these ‘intentional clones’.

2) Clone fragmentation – i.e., the fragmentation of clones results in scattered patterns of smaller clones that are harder to tackle.

This thesis describes two case studies in which we found many examples of tenacious clones in two public domain libraries. In those two case studies, and in other studies done by our research group, an approach called 'mixed-strategy' (i.e., mixing generative techniques and conventional implementation techniques) was able to achieve promising results in managing tenacious clones. Taking the success of mixed-strategy one step further, this thesis shows how mixed-strategy can be used to avoid most trade-offs incurred by conventional generics mechanisms. We use a comparative study of alternative designs of a web application to illustrate this point.

We use the term 'structural clones' to refer to higher-level clones, typically, cloned structures consisting of multiple program entities. Our thesis illustrates the concept of structural clones using various types of structural clones we found in software. Clone fragmentation may cause a clone to degenerate into a large number of small clone fragments. We show how such fragmentated clones can be viewed, and managed, as structural clones.

As the culmination of our research, we present SuM (Structural clone management using Mixed-strategy) as a holistic solution to the two challenges we set out to overcome. SuM is the application of mixed-strategy within the structural clone paradigm. SuM gives us a systematic approach to unify, and reuse, tenacious and fragmented clones, without sacrificing their benefits.

List of Tables

Table 1. Further analysis of reasons for clones 31

Table 2. Summary of web technology trends 44

Table 3. Average cloning for WAs of different size 62

Table 4. Size and cloning level comparison 103

Table 5. Change propagation comparison 104

Table 6. Effort for adding 'strong composition' 109

Table 7. Three-way comparison between files in the three structural clones 133

Table 8. Summary of file similarity characteristics in AB 134

Table 9. Clone management actions using mixed-strategy 142

Table 10. Typical approach for modification in different scenarios 160

Table 11. Basic entity types 166

Table 12. Basic structure types 168

List of Figures

Figure 1. A pair of parameterized clones 17

Figure 2. A structural clone 17

Figure 3. Web application reference architecture 36

Figure 4. Clone analysis workflow 54

Figure 5. Sample FSCurves 56

Figure 6. Cloning level in each WA 57

Figure 7. CCFinder Vs WSFinder 58

Figure 8. Distribution of clone size 59

Figure 9. FSCurves for all WAs 60

Figure 10. Percentage of cloned files 60

Figure 11. WA-specific files Vs general files 62

Figure 12. Movement of cloning level over time 63

Figure 13. Contribution of different file types to system size 64

Figure 14. Contribution of different file types to cloning 65

Figure 15. Partial class hierarchy of Buffer library 68

Figure 16. Feature diagram for Buffer library 69

Figure 17. Feature diagram for associative containers 70

Figure 18. Declaration of class CharBuffer and DoubleBuffer 72

Figure 19. Keyword variation example 72

Figure 20. Method toString() of CharBuffer and its peers 73

Figure 21. Clones due to swapping 73

Figure 22. Generic form of method ix() 74

Figure 23. Access level variation example 74

Figure 24. Generic form of method order() in direct buffers 75

Figure 25. A clone that vary by operators 75

Figure 26. Generic form of a clone found in 'type_traits.h' 76

Figure 27. Method get(int) of DirectIntBufferS and DirectFloatBufferS 76

Figure 28. array() method for int – found in IntBuffer.java.. 81

Figure 29. array() method for double – found in DoubleBuffer.java..................................... 81

Figure 30. X-framework for unifying the array() clone.. 82

Figure 31. Generating two array() methods from the x-framework.. 82

Figure 32. Clone unification in a mixed-strategy application... 84

Figure 33. A screenshot from the Staff module 92

Figure 34. Domain model of PCE... 92

Figure 35. Feature diagram of a PCE module... 93

Figure 36. High level architecture of PCE 95

Figure 37. The four PCE implementations 95

Figure 38. Design of PCEsimple 96

Figure 39. Some clones in PCEsimple... 97

Figure 40. Meta-model of a module in PCEpatterns 99

Figure 41. Design of Staff module in PCEpatterns... 99

Figure 42. Design of PCEunified.. 101

Figure 43. X-framework for PCEms.. 102

Figure 44. Cloning level in three PCEs 106

Figure 45. Page generation time comparison.. 107

Figure 46. Parallel editing of dynamic pages.. 112

Figure 47. Effect of clone unification on WYSIWYG editing 113

Figure 48. WYSIWYG editing when using mixed-strategy 114

Figure 49. Similarity across three conventional PCEs.. 115

Figure 50. Using XVCL to unify all three PCEs 115

Figure 51. File-level structural clones... 120

Figure 52. Module-level structural clones 121

Figure 53. Multiple structural clones in one file... 122

Figure 54. Two crosscutting structural clones 123
Figure 55. Structural clone with heterogeneous entities 124
Figure 56. Structural clone based on inheritance 124
Figure 57. Structural clone spanning multiple layers 125
Figure 58. An SC hierarchy 128
Figure 59. Architecture of the Adventure Builder application 129
Figure 60. Cloning across three supplier system 131
Figure 61. First and second tier structural clones in AB 134
Figure 62. Third, fourth, and fifth tier structural clones in AB 135
Figure 63. Applying mixed-strategy for managing existing clones 140
Figure 64. Applying mixed-strategy for managing potential clones 141
Figure 65. Clone unification activities using mixed-strategy 143
Figure 66. Harmonization example 147
Figure 67. Choosing master based on clones, an example 149
Figure 68. Unifying clones using SuM 151
Figure 69. Unifying exact simple clones 152
Figure 70. Unifying parametric simple clones 153
Figure 71. Unifying a structural clone using SuM 154
Figure 72. Unifying a structural clone with mixed-strategy alone 156
Figure 73. Partial SC hierarchy for Adventure Builder 162
Figure 74. Unification of structural clone [S]ext 163
Figure 75. Partial x-framework for SUPPLIER 164
Figure 76. Two different structural clones 165
Figure 77. SC1 and SC2 simplified into two similar structural clones 166
Figure 78. Composition model for entity types 167
Figure 79. Fragment structures that crosscut files 169
Figure 80. Unifying fragment structures that crosscut files 170
Figure 81. SuM activities described in this chapter 171

Figure 82. An example of an extra entity ... 172
Figure 83. Solution for extra entity ... 173
Figure 84. An example of an optional entity ... 173
Figure 85. Solution for optional entity ... 174
Figure 86. An example of a parametric entity ... 175
Figure 87. Solution to the parametric entity ... 175
Figure 88. An example of an alternative entity ... 176
Figure 89. Solution to the alternative entity ... 176
Figure 90. An example of a repetitive entity ... 177
Figure 91. Solution for repetitive entity ... 177
Figure 92. An example of a replaceable entity ... 178
Figure 93. Solution for replaceable entity ... 178
Figure 94. Examples of a reordered entity ... 179
Figure 95. Solution for the reordered entity ... 179
Figure 96. Alternative entities or parametric entities? ... 181
Figure 97. Handling extra entities and parametric entities in AB ... 183
Figure 98. Optional entities and alternative entities in AB ... 183
Figure 99. Handling repetitive entities in AB ... 184

Chapter 1.

Introduction

'Cloning Considered Harmful' Considered Harmful

-Title of [KG06]

1.1. The problem

Similarities at analysis, design and implementation levels in software are great opportunities for reuse. When such similarities are not exploited, they can lead to duplication in software (also called ‘clones’). Therefore, clones signal unexploited reuse opportunities. Clones also complicate software maintenance by making the code base larger than necessary. They hinder program comprehension by injecting implicit dependencies among program parts. Tracing and updating all the clones is a tedious and error-prone process, often resulting in update anomalies (inconsistencies in updates). Therefore, clones signal opportunities for program simplification. Unifying clones with unique generic representations reduces the code size and conceptual complexity of software, explicates the dependencies, and reduces the risk of update anomalies.

Yet clones continue to plague today’s software. Case studies have found cloning levels as high as 68% [JL03]. With the enormous amount of code being maintained today (estimated 250 billion LOC in 2000 [Som00]) costing enormous resources (more than $70 billion in US alone in 1995 [Sut95]), there could be significant benefits in finding an effective solution to the clones problem.

1.2. Thesis objectives

While most clones have a negative effect on maintenance, some clones also have certain benefits. For example, in-lining function calls creates clones, but also improves the runtime performance by reducing function calls. We believe that the high level of cloning in today's software is due to the lack of a holistic approach to unify and reuse clones without losing their benefits. Therefore, we use the term 'clone management' to describe a holistic approach to counter negative effects of clones, while preserving and possibly leveraging their positive aspects. In support of finding an effective clone management approach, we define the objectives of this thesis as:

Objective 1. To identify, and analyze, drawbacks involved in applying conventional implementation techniques to manage clones

Objective 2. To define, apply, and evaluate a holistic solution to manage clones in which we counter negative aspects of clones, while preserving and leveraging their positive aspects.

1.3. Thesis scope

Cloning problem is applicable to any kind of software. However, this thesis specifically tackles the cloning problem in the web application domain. We use a sample of web applications to evaluate the intensity and nature of the cloning problem in web domain. We evaluate the current state of the art in clone management using both model web applications built based on industry best practices, and real web applications built under typical schedule pressure.

Product lines (a set of similar products) are examples of cloning at a massive scale. Our research mainly focuses on cloning issues within single applications, but where applicable, we

extend our focus to product line situations. For example, similar modules within a single application can be considered a mini product line, and the finding from such clones can be generalized to larger product lines. However, we do not address the full range of product line issues.

According to Rieger [Rie05], most cloning is done as a way of reusing one's own code, or code from inside sources (i.e., same team, same product line, same company). Therefore, we limit our focus to the cloning from *own code* or from *inside sources*. Cloning from outside sources (from online code examples, open source systems) has additional issues, and such cloning is not considered in this thesis.

1.4. Research and contributions

We started our research with a survey of literature in past clone research. Then, we conducted an extensive study of cloning in web applications, to evaluate the prevailing level of cloning in today's state of the practice. We also did a survey of the technologies used for building web applications, to understand the current state of the art in web application building.

Theses contributions resulting from these works are:

Contribution 1. It defines, and uses, a need-oriented framework for organizing web technologies. This framework helps us to overcome the difficulties of keeping track of the rapidly evolving web technology landscape.

Contribution 2. It provides concrete evidence of the cloning problem in the web domain, and compares the situation with traditional applications. It also identifies similarity metrics useful for evaluating the cloning level of software.

Based on this initial work, we decided to address two challenges in clone management: 'tenacious clones', and 'clone fragmentation'.

Work in the area of tenacious clones

'Tenacious clones' is the term we use to collectively refer to clones that tend to persist in software, mainly due to the following three reasons.

(a) For some clones unification is just not technically feasible. This may be due to limitations in the implementation technology, such as restrictions on type parameterization (e.g., Java does not allow type parameterization for primitive types). We coined the term 'non-unifiable clones' to refer to such clones.

(b) In other cases, it may be possible to unify clones using conventional techniques, but such unification requires us to trade-off other important qualities of the software. To give an example, unifying clones that have performance benefits may improve the maintainability of the code, yet the resultant executable would be slower than the clone-included code. We use the term 'unification trade-offs' to refer to such trade-offs.

(c) Some clones are meant to remain in software, because they have been created to serve a purpose. We call these 'intentional clones'. Examples include clones created to improve performance, reliability, or clones created when following standards/frameworks (such as .NET and JEE patterns).

In other words, clones may be tenacious because they are non-unifiable, intentional, or because their unification trade-offs are unacceptable. As further evidence of such tenacious clones, this thesis describes two case studies in which generics in Java and C++ failed to unify certain clones.

This thesis adds the following contribution in the area of tenacious clones.

Contribution 3. It shows more evidence of tenacious clones using two case studies (this is a joint contribution with Basit, H. A.)

In those two case studies, and in other studies done by our research group, promising results could be achieved when applying a strategy called the 'mixed-strategy' to unify such clones. Mixed-strategy is a meta-programming based reuse technique our research team has been developing for a number of years now. It uses conventional techniques to unify clones when possible, but resorts to the unrestrictive parameterization and composition capabilities of XVCL (XML-based variant configuration language [XVCL]) to unify non-unifiable clones. In the past case studies done by our research group, mixed-strategy have shown promise in dealing with non-unifiable clones and intentional clones. Taking this success of mixed-strategy one step further, this thesis shows how mixed-strategy can be used to avoid most unification trade-offs incurred by conventional clone unification techniques. We use an empirical study of alternative designs of a web application to illustrate how mixed-strategy avoided the trade-offs we observed when using conventional techniques such as design patterns.

This work produced the first main contribution of this thesis (in response to Objective 1):

Contribution 4. It illustrates and analyzes the trade-offs in applying conventional clone unification mechanisms to unify clones in the web application domain. It shows how mixed-strategy avoids most such unification trade-offs.

Work in the area of clone fragmentation

Clone fragmentation is the phenomenon of clones getting broken into smaller clones. Reasons for such fragmentation include software decomposition, requirements of the frameworks and design paradigms, and injection of variations. A concept related to clone fragmentation is 'structural clones': a term coined by our research group to refer to higher-level clones, typically cloned structures consisting of multiple program entities. This thesis illustrates the concept of structural clones using various types of structural clones we found in software. We show how fragmented clones can be viewed, and unified, as structural clones.

This work adds the following contribution to this thesis:

Contribution 5. It illustrates the concept of structural clones using examples from various software systems. It shows how fragmented clones can be treated as structural clones.

Note: Tenacious clones are a facet of the 'weak generics problem' put forward by Jarzabek [XVCL]. Weak generics problem states that generic design is difficult to achieve in the frame of conventional techniques.

The complete solution

As the culmination of our research, we present SuM (Structural clone management using Mixed-strategy) - a systematic and holistic approach to unify and reuse tenacious, and possibly fragmented, structural clones, without compromising other desirable qualities of the software. SuM is essentially a combination of the mixed-strategy and the structural clone concept which, taken together, overcomes the two challenges we set out to tackle. We first present the basic activities involved in applying the SuM to a legacy system or a system under development. We further support the SuM approach by presenting the basic SuM unification schemes, i.e., basic structural clone types and the mixed-strategy solutions for each basic structural clone type.

This work produced the second main contribution of the thesis (in response to Objective 2):

Contribution 6. It presents SuM, a combination of mixed-strategy and the structural clone concept to provide a systematic and holistic approach to unify and reuse tenacious, and possibly fragmented structural clones, without compromising their benefits.

1.5. Experimental methods

Our experiment method consisted of the following salient features.

- **Quantitative surveys** – To identify the intensity of the cloning problem, we did quantitative surveys of existing applications, using various clone detection/analysis tools
- **Critical analysis of existing applications** - To identify the nature of the cloning problem we examined a wide range of existing applications.
- **Empirical studies** – To observe how clones are created, and how they can be managed, we built various applications under a controlled lab environment.
- **Comparative studies** - To evaluate existing solutions and our proposed solution, we performed comparative studies, in reengineering or evolving existing applications, as well as in developing new applications.
- **Industry feedback** – We continually collaborated with our industry partners, to obtain feedback on our findings, and to obtain real life source code for our analysis.

1.6. Thesis roadmap

Chapter 2 (Background and Related Work) gives some background on the cloning problem, and summarizes previous research done in this area. It also gives some background on the web application development, and comments on why addressing the cloning problem in the web application domain is important.

Chapter 3 (An Investigation of Cloning in Web Applications) presents a study that evaluates the level of cloning prevalent in today's web applications.

Chapter 4 (More Evidence of Tenacious Clones) describes two case studies in which we found many tenacious clones in two popular public domain libraries: Java Buffer library, and the C++ Standard Template Library.

Chapter 5 (Mixed-Strategy) introduces the mixed-strategy, and the XVCL meta-programming language which is at the core of the mixed-strategy.

Chapter 6 (Unification Trade-offs) uses an empirical study of alternative designs of the same web application, to illustrate how the mixed-strategy overcomes most of the unification trade-offs incurred by other clone unification techniques.

Chapter 7 (Structural Clones) illustrates the concept of structural clones using examples from various software systems. Then it goes on to show how structural clones can help in managing fragmented clones, using Java Adventure Builder model application as an example.

Chapter 8 (SuM: Structural Clone Management Using Mixed-Strategy) presents SuM as a unified approach to overcome the challenges of tenacious clones, and clone fragmentation. It systematically describes the basic activities and techniques of applying SuM, including basic SuM unification schemes.

Chapter 9 (Conclusions and Future Work) sums up the thesis and points to possible future directions.

Appendix A provides a summary of essential XVCL syntax, for the convenience of the reader.

1.7. Research outcomes

Presented at Refereed International Conferences

- Basit, H. A., Rajapakse, D. C., and Jarzabek, S., “An Empirical Study on Limits of Clone Unification Using Generics,” *17th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, Taipei, Taiwan, 2005, pp. 109-114
- Rajapakse, D. C., and Jarzabek, S., “An Investigation of Cloning in Web Applications,” *5th Intl Conference on Web Engineering (ICWE'05)*, Sydney, Australia, 2005 (acceptance rate **19%**), pp. 252-262
- Rajapakse, D. C., and Jarzabek, S., “A Need-Oriented Assessment of Technological Trends in Web Engineering,” *5th Intl Conference on Web Engineering (ICWE'05)*, Sydney, Australia, 2005, pp. 30-35
- Basit, H. A., Rajapakse, D. C., and Jarzabek, S., “Beyond Templates: a Study of Clones in the STL and Some General Implications,” *28th Intl. Conf. on Software Engineering (ICSE'05)*, St. Louis, Missouri, USA, 2005 (acceptance rate **14%**), pp. 451-459
- Rajapakse, D. C., and Jarzabek, S., “An Investigation of Cloning in Web Applications,” poster presentation at *14th Intl World Wide Web Conference (WWW'05)*, Japan, 2005
- Basit, H. A., Rajapakse, D. C., and Jarzabek, S., “Extending Generics for optimal Reuse,” poster presentation at *8th Intl. Conf. on Software Reuse (ICSR'04)*, Madrid, Spain, 2004

Tutorials at International Conferences

- Jarzabek, S. and Rajapakse, D. C., “Pragmatic Reuse: Building Web Application Product Lines,” *5th Intl Conference on Web Engineering (ICWE'05)*, Sydney, Australia,2005

Chapter 2.

Background and Related Work

What a tangled web we weave

-Title of [Pre00]

This chapter gives some background on the cloning problem, and summarizes previous research done in the area of cloning. It also gives some background on the area of web engineering, and comments on why addressing the cloning problem in web domain is important.

The organization of this chapter is as follows:

Section 2.1 defines commonly used clone nomenclature and introduces various aspects of clones, such as causes, effects, detection and taxonomies.

Section 2.2 presents various types of clone management approaches, and discusses practical challenges in effective clone management.

Section 2.3 gives a brief introduction to web applications, presents an overview of today's web technologies using a need-oriented framework we defined for web technologies, and discusses special characteristics of web application development as compared to traditional software development.

Section 2.5 describes various research efforts specific to cloning in web applications, and comments on why web domain might be suitable our research.

Section 2.4 summarizes why engineering web applications may be somewhat different from engineering traditional applications.

The contribution contained in this chapter is:

Contribution 1. It defines, and uses, a need-oriented framework for organizing web technologies. This framework helps us to overcome the difficulties of keeping track of the rapidly evolving web technology landscape

2.1. Clones

2.1.1. Simple clones

Simple clones, generally referred to as just 'clones' in literature, are code fragments that are similar to each other. More formally, a 'clone relation' is said to exist between two code fragments if there is a *significant similarity* between them. The threshold of significant similarity is open to interpretation. For example, one may define significant similarity between two code fragments as 'more than 90% of the contents to be exact matches'. A 'clone relation' is an equivalence relation (i.e., reflexive, transitive, and symmetric relation) [UHK+02]. For a given clone relation, a pair of code fragments is called a 'clone pair' if a clone relation holds between them. Such fragments are called clones of each other. An equivalence class of a clone relation is called a 'clone class'. That is, a clone class is a maximal set of code fragments in which a clone relation holds between any pair of code fragments.

Two[1] commonly found types of code clones are:

- Exact code clones – code fragments that are identical to each other.

[1] Some use the term "gapped clones" to refer to another type of clones that have non-parametric variations. We consider such clones under the category of structural clones (section 2.1.2)

- Parameterized code clones – clones that show only parametric differences (e.g., Figure 1)

Figure 1. A pair of parameterized clones

2.1.2. Structural clones

'Structural clones' are higher level clones that represent repeated program structures. Causes for structural clones include similarities in analysis (e.g., due to repeating analysis patterns [Fow96]) and design (e.g., due to repeating design patterns [GHJ97]), requirements of the programming language (e.g., due to a repeating coding idiom), and mental templates repeatedly used by programmers. Figure 2 shows a pair of structural clones our industry partner (STE Eng Pte Ltd., Singapore) found in a real system. Each clone is made up of a structure of 4 modules, and spans multiple layers, from GUI layer to database (DB) layer.

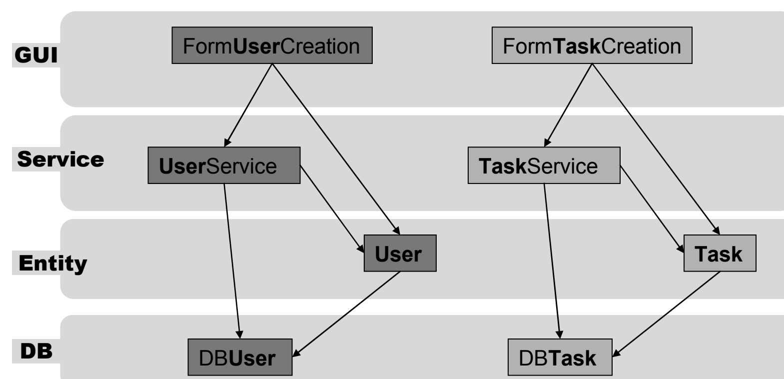


Figure 2. A structural clone

'Structural clone pair' and 'Structural clone class' can be defined similar to simple clone pair and simple clone class.

2.1.3. Reasons for clones

In an ethnographic study of IBM programmers, Kim et al [KBLN04] observed that a programmer produced four non-trivial clones per hour on average. Her other work [KN05][KSNM05] found that most clones (up to 68% of the clones found) are not locally refactorable.

> A study revealed that a programmer produces four non-trivial clones per hour on average.

Summarizing the cause and the effect of clones, Baxter et al [BYM+98] states "The act of copying indicates the programmer's intent to reuse the implementation of some abstraction. The act of pasting is breaking the software engineering principle of encapsulation". Literature frequently, if not extensively, discusses reasons for clones. Given next is a list of those reasons, compiled from such sources. We believe this is the most comprehensive compilation of clone causes yet, while the list given by Rieger [Rie05] is a close second.

(a) **Cloning is simpler.** Cloning gives us short-term productivity gains, because copying a piece of code and modifying it is much simpler and faster than writing it from scratch. In addition, the fragment may already be tested so the introduction of a bug seems less likely [DRD99]. Time pressure typical to industrial software development is a common excuse for cloning to save time and effort.

(b) **Cloning is less risky.** A conservative and protective approach to modification and enhancement of a legacy system too would introduce clones [KKI02]. A programmer who does not fully understand the original code (or does not have time to invest in understanding it) would opt to work with a copy rather than altering the original, to avoid possible ripple effects [FR99]. Cordy [Cor03], drawing from his experience in studying 4.5 GLOC of COBOL code in the financial industry, observed that clones are sometime not removed due to the risks attached to code modifications. In critical industrial software

such as financial systems, cost of quality control is high and the cost of failure is immense [Cor03]. This encourages cloning to avoid cost and risk of changing existing software.

(c) **Some clones reduce explicit coupling.** Cloning is sometimes used to reduce unwanted coupling [Cor03]. The rationale behind this is that a cloned code is effectively protected against latent changes to the original code. A common example is when developers want to share an unstable piece of code while working in parallel. Though this strategy gives protection against latent changes to the original code, it also deprives the clone of latent fixes/enhancements to the original code.

(d) **Some clones improve space/time efficiency.** Efficiency considerations may render the cost of a procedure call seems too high a price [DRD99]. Systems with tight time constraints are often hand-optimized by replicating frequent computations, especially when a compiler does not offer automatic optimizations [BYM+98]. All too often, programmers are not aware of such compiler help, even if it is available. Additional generalizations in the reusable code often make reusable code larger than a custom-fitted version of it. When runtime memory is scarce, developers can opt for leaner custom-fitted clones rather than use a bloated reusable version.

(e) **Some clones improve understandability.** Ironically, some cloning is done with the intention of increasing understandability and the maintainability of the code. For instance, sometimes methods are in-lined to reduce levels of indirection. Such clones help in improving locality and linearity [SCD03], two properties important for readability and understandability [Wei71]. Some clones are made solely to reduce coupling and increase understandability, so as to ease future maintenance.

(f) **To follow a style/pattern.** Sometimes a "style" for coding a regularly needed code fragment will arise, such as error reporting or user interface displays. The fragment will purposely be copied and modified to maintain the style [BYM+98].

(g) **Frozen legacy code.** When the reuse candidate is part of a legacy system that is frozen, the only option is cloning.

(h) **Uncooperative code owners** Developers who are unwilling to change shared code leave others with no choice but to clone when they want a slightly different version.

(i) **Ignorance of reusable code.** Sometimes due to the ignorance of reusable code, or due to lack of mechanisms to find a reusable code (e.g., due to lack of proper documentation), developers "re-invent the wheel" [Kru92] by implementing similar code repeatedly. This usually results in semantically equivalent clones.

(j) **Not invented here syndrome**. An unwillingness to use others' code too leads to re-inventing the wheel [Rie05].

(k) **To inflate productivity measures.** Evaluating the performance of a programmer by the amount of code produced gives a natural incentive for cloning [DRD99].

(l) **Mental macros.** Mental macros (code segments frequently coded by a programmer in a regular style, such as payroll tax, queue insertion, data structure access etc.) are simple to the point of being definitional. As a consequence, even when copying is not used, the resultant code might have clone-like properties [BYM+98]. Usually clones created in this manner are small, though can be frequent.

(m) **Just bad coding.** Some clones are in fact complete duplicates of functions intended for use on another data structure of the same type [BYM+98]. This happens when inept programmers do not realize they can simply use existing code. Such lack of knowledge of proper reuse techniques too can lead to clones.

(n) **Due to limitations of implementation technologies used.** Limitations of programming languages sometimes necessitate clone-like code [Joh94]. For example, strongly typed languages require clone-like code for handling different types of data while weakly typed languages can use the same code.

(o) **Requirements of platforms/ frameworks.** Complying with protocols (e.g., CORBA) or frameworks (e.g., Enterprise Java Beans) sometimes requires certain code/files to be duplicated in various physical locations.

(p) **Clones induced by editors and other tools.** Many Integrated Development Environments (IDEs) and other tools like visual GUI builders and UML-to-code generators are not specifically built to minimize cloning. On the contrary, many generate clone-like code because the sophisticated logic needed to reduce cloning is beyond those tools.

(q) **Accidental clones.** There are occasional code fragments that are just accidentally identical [BYM+98]. Such accidental clones are small and rare. Therefore we ignore accidental clones from this point onwards.

2.1.4. Effects of clones

Following are the main negative effects of clones.

(a) **Large clones are laborious to create.** The process of cloning itself is laborious and error prone in certain cases. For example, some clones require the same modification to be repeated many times during the 'modify' part of the copy-paste-modify cycle (e.g., change a variable name in each place it is used). Errors in this process can lead to unintended aliasing and latent bugs [Joh94] (e.g., if a variable in the copied code has the same name as a variable in the reuse context).

(b) **Clones multiply maintenance effort.** Maintenance work has to be repeated for all instances of the clones [Rie05].

(c) **Clones increase the risk of update anomalies.** During maintenance, any changes to a cloned code have to be replicated possibly in all copies of the code [DRD99]. Clones make the source files very hard to modify consistently since it is difficult to find all

instances of the clone. Modifying a set of clones blindly (using search and replace techniques) can introduce bugs since each clone might be different to the other in subtle ways. For a large and complex system, there are many engineers who take care of each subsystem and then such modifications become very difficult [KKI02].

(d) **Clones increase cognitive load.** When attempting to maintain a software system, the maintainer must first gain some understanding of the system. Clones increase the size of the code one has to understand in order to understand the system fully [BM97]. Clones make it difficult to see what is similar from what is different. Cloning to avoid unwanted side effects (described earlier) can lead to dead code. Such dead code acts as red herrings to mislead maintenance engineers at the cost of wasted time and effort [Joh94].

(e) **Impact on compile/load/runtime efficiency.** Code duplication within a system increases the size of the code, extending compile time and expanding the size of the executable. Note that in certain situations clones increase the compile/load/runtime efficiency (*cf* 2.1.3(d))

Except for (e), the other four negative effects of clones are directly related to software maintenance. Hence, clones are generally thought to have a negative impact on maintenance.

> Clones are generally thought to have a negative impact on maintenance.

Clones' effect on maintenance was studied by Burd, and Munro [BM97] who discuss the maintenance problems created by clones in legacy systems and emphasize the need for greater tool support to tackle clones. Kapser and Godfrey [KG05] also point out the value of improved tool support for clone investigation. Monden et al [MNK+02] did a quantitative study that investigated the relationship between code clones in legacy software have with software reliability and maintainability. It reported that clone-included modules are 1.7 times

as reliable as non-clone modules on average. However same study reported that modules with large clones were less reliable than non-clone modules on average. Also they found that clone-included modules are less maintainable than non-clone modules on average and modules having larger code clones are less maintainable than modules having smaller code clones. The experiments claims to have quantitatively pointed out that there is a relation between code clones and the software reliability and maintainability, though the relation itself is not clarified. In another study Lague et al [LPM+97] investigated, and confirmed, the potential benefits of introducing a function clone detection technology in an industrial software development process.

2.1.5. Clone detection

In clone research, clone detection appears to be the most popular (e.g., [BYM+98][MLM96][Joh93][Joh94][DRD99][Bak95][KKI02][KH01a][KH01b][PMP02][CDS04]). Clone detection and analysis are very important support activities when tackling clones. This is particularly important in large legacy systems where locations of the clones are not known, and the maintainers are not necessarily the original developers. While most clone detection so far has concentrated on detecting cloned code fragments, there has been some effort on moving beyond this level, into detecting higher level clones. For example, Marcus and Maletic [MM01] has attempted to detect what they call "high level concept clones". Ueda et al [UKKI02a] reports a method to detect gapped clones (clones with non-parametric variations). Our own research group is working on detecting structural clones (e.g., [BJ05]). Burd and Bailey [BB02] provide a good evaluation of clone detection tools. Among the interesting uses of clone detection are plagiarism detection (e.g., [PMP02]), detection of refactoring opportunities (e.g., [HKK+04]), and the detection of crosscutting aspects (e.g., [BVVT04][BVVT05])

2.1.6. Clone taxonomies

Research in the area of clone taxonomies includes work by Kapser and Godfrey [KG03a] [KG03b] and Balazinska et al [BMDL99] (a classification based on reengineering opportunities). Research specifically in the area of clone visualization includes work by Johnson [Joh96], and Ueda et al [UKKI02b].

2.2. Clone management

Rieger [Rie05] defines clone management as “activities to keep their (clones’) detrimental effects in check” and describe three types of clone management measures: preventive, corrective and compensatory.

- Preventive measures: A strict definition is, ‘the avoidance of creating clones in code’. A more practical definition would be, ‘the avoidance of clones in released code’.
- Corrective measures: Removing clones from existing software.
- Compensatory measures: Compensating negative effects of clones, without actually removing them from the code

2.2.1. Preventive clone management

The essence of preventive clone management measures is to apply a reuse technique instead of cloning in the first place. A pure preventive approach calls for proactively recognizing potential clones before they are created, and using a reuse technique to avoid the cloning. Some conventional reuse techniques used for clone avoidance are given below.

- Language features – Language features such as closures, higher order functions, generics, inclusion, reflection, and inheritance can be used to avoid clones.

- Design patterns [GHJ97] – There are design patterns that help to avoid duplication (some of these are mentioned in section 6.1.4).
- Server pages – Server pages (e.g., ASP, JSP, PHP) is the most common technique for implementing web application user interfaces. The essence of this technique is to combine HTML with programming language features to generate web pages at runtime (dynamic page generation). A Server page can represent many similar web pages in a generic but adaptable form.
- Meta-programming – Template meta-programming (e.g. using C++ templates) [CE00], and macros are examples of meta-programming approaches that helps to avoid clones.
- Platforms/Frameworks – Frameworks helps us to reuse common code (e.g., when using frameworks such as Struts, Ruby on rails, Spring), and more low level services (e.g., when using such as J2EE, .NET).
- Reuse at higher level of abstraction – Model Driven Development, Domain Specific Languages and generators are examples of reusing at a higher level than code.
- Separation (and reuse) of concerns – Aspect Oriented Programming, Hyperspaces [TOHS99] claim to be able to separate concerns, which should also help us to reuse them.

However, a purely preventive approach requires much upfront analysis, and high expertise on the part of the programmer. A more pragmatic approach is to prevent clones being released into production code, for example, by using clone detection at defined points (e.g., at check-in to the source control). This is in fact a corrective measure applied very early in the life of a clone, when it is easiest to correct and its negative effect is minimal. For example, the experience report of Lague et al [LPM+97] describes a control technique used for preventing clones being created. In this mechanism, clones are detected automatically at the point of

submitting new source code to the source control, and unjustifiable clones are removed (manually).

Next, we give various research efforts related to preventive clone management, or more specifically, related to reuse techniques that help to prevent clones.

Schwabe et al's OOHDM (Object Oriented Hypermedia Design Method) [SERL01] [SR98a][SR98b][RSL00][SREL01][SRB96][RSG97] "uses abstraction and composition mechanisms in an object oriented framework to, on one hand, allow a concise description of complex information items, and on the other hand, allow the specification of complex navigation patterns and interface transformations". OOHDM promotes separation of concerns at design level. It involves three design steps: conceptual design, navigational design, and abstract interface design. Such separation is expected to help in reuse at design level, thus preventing clones.

Gaedke et al's WebComposition [GGS+99][GG00] is a component based approach to web application engineering which tries to find a better composition model for web applications than the traditional coarse-grained resource-based model. WebComposition was initially enabled by WCML (WebComposition Markup Language) [GSG00][GT99]. WCML allows us to define arbitrarily sized components and combine them in a fairly unrestricted manner using aggregation and prototype-based inheritance. Thus, WCML views a web application as a composition of arbitrary sized components. With WCML's arbitrary sized components it was expected to, among other things, achieve better reuse. WCML project is no longer active. The last released version supported generating HTML artifacts. Currently WebComposition concept is continued in WSLS (WebComposition Service Linking System) [GNT03][GNM04a][GNM04b], which allows us to configure and combine existing services in prescribed ways. Thus, WSLS views a web application as a composition of services.

Ginige et al have developed a component based web application development framework called CBEADS [GDG05] that is based on end user development paradigm. In CBEADS, end users themselves can create variants of application functionality using a GUI provided. This reduces the need for common business logic variations to be maintained by developers at code level, thus reducing the possibility of clones related to such common variations.

WebML[CFB00][CFM02] follows the model driven development (MDD) paradigm. It is a visual language for expressing the hyper-textual front-end of a data-intensive web applications. WebML is backed by a CASE tool called WebRatio [ABB+04]. WebRatio uses WebML for the functional specification, and Entity-Relationship (ER) model for the data requirement specification. The code is generated semi-automatically.

An approach to generate web application based on templates, such as used in Freemarker and Velocity, is proposed by Zdun [Zdu02].

2.2.2. Corrective clone management

The essence of corrective clone management is to remove existing clones by using alternative reuse mechanisms. There are two approaches to clone removal:

- Refactor [Opd92][Fow99] – Incremental changes to replace clones with reuse techniques described in the previous section, while keeping the external behavior unchanged. Refactoring involves small scale, localized changes to the implementation, typically to improve the implementation. The reuse techniques used in refactoring are drawn from the ones described under preventive clone management (previous section).
- Rebuild – Redesign the system from scratch. This involves drastic changes to the system, possibly including a changeover to a different implementation technology

with better reuse support. Again, the reuse techniques used are drawn from the previous section.

Next, we describe some research on corrective measures.

There are number of research work on clone removal in software systems. Balazinska et al [BMD+00][BMD+99] describes a method for computer assisted clone refactoring for OO systems using template and strategy design patterns. Fanta, and Rajlich [FR99] describes a tool assisted clone unification technique that is capable of removing certain function and class clones in OO software. In [HUK+02] Higo et al describe how the clone visualization tool Gemini [UHK+02] was extended to support refactoring of clones. Di Penta et al [DNAM05] discuss language independent software renovation in general, including factoring out clones, although they do not describe a specific technique.

De Lucia et al's work on detecting cloned patterns in web applications [DFST04] also includes removing those cloned patterns. However, the emphasis is on the novel approach they used to detect cloned navigational patterns, rather than the specific technique they use for removing those patterns.

Work described in [SCD03] attempts to select unification method that minimizes the disruption to the structure of the original web site, so that the resulting code is still familiar to its maintainers and maintainable by hand.

Boldyreff and Kewish [BK01] propose to store unified clones in a relational database, and to retrieve the clone at runtime using scripts. This approach has the potential to remove the highest proportion of clones, according to [SCD03]. However, they also argue that this approach can disrupt the website structure. A somewhat similar approach used by Ricca and Tonella [RT03] where clustering is used to recognize candidate template, to be used in the dynamic generation of the pages. A comparison between original pages and template identifies the records to be inserted into the database. Then, a script generates the migrated

pages dynamically, from the template and the database. Manual intervention is limited to the refinement of the constructed template and database.

Work by Ping and Kontogiannis [PK04] proposes an approach to automatically refactor web sites that removes some "potential duplication".

Tonella et al tackles a special type of clones – clones created by language-specific variations in multi-lingual web sites – by introducing a language called MLHTML [TRPG02] to unify such clones.

2.2.3. Compensatory clone management

The essence of compensatory clone management is to combat negative effects of clones without removing clones. The most straight forward compensatory technique is documentation. This may be in the form of comments in the source code, or in the form of a separate list of clones.

Software configuration management (SCM) helps in managing different versions of a product, and since different versions of a product are in fact clones, SCM too can be considered as having a compensatory effect on clones.

Another approach is to automatically extract the clone information at real time, using clone detection/analysis tools. The work of Kim et al [KN05][KSNM05] advocate the use of a clone genealogy extractor to support clone management. Such a tool can provide real-time data about clones in the system, thus compensating some of the negative effects of clones. She also advocates the use of simultaneous text editing tools, such as [MM01], which may help in reducing update anomalies.

An experience report by Lague et al [LPM+97] describes a compensatory measure called "problem mining" used for managing clones in an industrial system. In problem mining,

changes submitted to the code repository are compared with all existing code and any clones found are presented to the developer, thus mitigating the risk of an update anomaly.

Automatic generation of clones (e.g., using IDEs or frameworks) address one negative effect of clones: the laboriousness of creating clones (*cf* section 2.1.4 (a)). Hence its compensatory effect is partial at best. Once generated, these clone are maintained either at code level, or at a higher level (e.g., via a GUI). The T-Web system [TST03] by Taguchi et al is another example of a generative framework. In T-Web web applications are generated from based on web transition diagrams. CBEADS framework [GDG05] mentioned in section 2.2.1 is also a generative framework because it generates the code as directed by the end users via the GUI. MODFM is another generative framework proposed by Zang and Buy [ZB03]. Another generative approach is suggested by Loh and Robey [LR04], in which they propose to generate web applications from use cases.

2.2.4. Practical challenges in clone management

Further analysis of reasons for clones, shown in Table 1, gives us some clues as to why cloning is pervasive in today's software. For each reason for clones, the table speculates (based on author's opinion) the benefit (if any) given by those clones, whether the benefit is *transient* (for a short period only) or *permanent* (throughout the life of the application), whether creation of the clone could be prevented, and whether the clone could be removed (corrected) without negating the reason behind its creation, using conventional reuse techniques[2]. In the last column we categorize the reasons into three types: benefit (i.e., clone gives some benefit), non-unifiable clones, and organizational (i.e., clone is caused by organizational problems such as deficiencies in its reuse culture).

---

[2] By 'conventional reuse techniques' we mean those that are in common use among software developers. Therefore, we exclude techniques that are at experimental level such as those proposed by researchers.

Table 1. Further analysis of reasons for clones

Reason	Benefit	Transient (T)/ Permanent (P)?	Can prevent?	Can correct?	Root cause
1. Cloning is simpler	reduces development effort	T	**No**	Yes	benefit
2. Cloning is less risky	needs less testing	T	**No**	Yes	benefit
3. Mental macros	reduces development effort	T	**No**	Yes	benefit
4. Clones induced by editors and other tools	IDEs increase productivity	T/P[3]	**No**	Some	benefit
5. Some clones reduce explicit coupling	easy for clones to diverge	T/P[4]	**No**	Some	benefit
6. Requirements of platforms/ frameworks	can continue to use platforms/ frameworks	P	**No**	**No**	benefit
7. Frozen legacy code	legacy code does not change	P	**No**	**No**	benefit
8. Some clones improve space/time efficiency	better performance	P	**No**	**No**	benefit
9. Some clones improve understandability	easier to maintain	P	**No**	**No**	benefit
10. To follow a style/pattern	standardization	P	**No**	**No**	benefit

[3] IDEs help to rapidly develop applications (transient benefit), but they also help in long-term maintenance (permanent benefit) , removing these clones may affect the ability to use the editor/tool to maintain the code

[4] This benefit is permanent for systems that continually evolve, otherwise it is relevant only during the initial volatile period

11. Due to limitations of implementation technologies used	none	-	**No**	**No**	non-unifiable[5]
12. Just bad coding	none	-	Yes	Yes	organizational
13. Ignorance of reusable code	none	-	Yes	Yes	organizational
14. To inflate productivity measures	none	-	Yes	Yes	organizational
15. Not invented here syndrome	none	-	Yes	Yes	organizational
16. Uncooperative code owners	none	-	Yes	Yes	organizational

As Table 1 shows, clones created under the first 10 reasons provides some real benefit. In recognition of the fact that clones also have such benefits, we extend Rieger's definition [Rie05] of the term "clone management" to describe a holistic approach to counter negative aspects of clones, *while* preserving and leveraging their positive aspects.

> *Clone management* is a holistic approach to counter negative aspects of clones, while preserving and leveraging their positive aspects.

Looking at the last column of Table 1 we see that some clones are created because removing them forces us to trade-off the benefits given by the clones. Some such clones are in fact *intentional* (e.g., item 8), and not meant to be removed at all. Even when the clone is unintentional, if the benefit of the clone is permanent, anticipated trade-offs prevent us from removing the clone from code. Clones with only transient benefits can be (theoretically) corrected once their benefit ceases to exist. However, corrective measures carry the risk of breaking existing code [Cor03], and hence require extensive regression testing. Therefore,

[5] These, clones are 'non-unifiable' within the confines of the implementation technologies already used in the system. i.e., short of introducing new technologies just for the sake of unifying those clones.

even clones with only transient benefits may not be amenable to latent correction, due to a trade-off in system reliability. We call such trade-offs that help clones to persist 'unification trade-offs'.

> Even the clones that no longer have benefits may not be removable due to the risk of breaking existing code.

Clones created due to limitations of implementation technologies (item 11 in Table 1) can neither be prevented nor corrected. We identify such difficult to unify clones as another practical challenge in the prevention and correction of clones, and call such clones 'non-unifiable clones'. We use the umbrella term 'tenacious clones' to refer to clones that are intentional, non-unifiable, or those that have unification trade-offs. Note that some tenacious clones may belong to more than one of these three categories.

Clones blamed on the last five reasons in Table 1 are caused by deficiencies in the organization's reuse culture. While we accept this as another fundamental obstacle to effective clone management in particular (and effective reuse in general), we do not address this problem in this thesis. We believe this is an organizational problem that can be addressed independent of the technical challenges we address here.

We have identified another practical challenge in effective clone management, not indicated by Table 1. Our first clue to this challenge was the huge numbers of clones typically reported by clone detection tools, for example, when we detected clones in web applications using CCFinder tool (described in next chapter)[6]. A manual examination of some of those clones suggested that clones reported are bits and pieces belonging to larger clones. Re-examination of previous case studies done by our research group (e.g., [JL03], [KSNM05]), and the

[6] In another study, CCFinder found 8047 clone pairs in Java development kit (JDK) and 25621 clone pairs between FreeBSD and NetBSD codes [KKI02].

parallel research done on clone detection by Basit H. A. (e.g., [BJ05]) further confirmed this state of affairs. That is, coarse-grained clones have been fragmented into large numbers of smaller clone fragments, due to various forces. We identify such 'clone fragmentation' as another practical challenge in effective clone management.

Our analysis of Table 1 also hints at the inadequacies of preventive and corrective clone management measures, and therefore the importance of complementing them with compensatory measures. Preventive and corrective measures only help in the "reducing negative effective of clones" aspect of clone management. In contrast, compensatory measures address both negative and positive aspects.

However, existing compensatory techniques presented in section 2.2.3 are not very effective:

- Documentation, the simplest of compensatory measures, is notorious as the first to be neglected under time pressure. Out-of-sync documentation can sometimes cause more harm than good.
- SCM, while capable of keeping track of a range of product configurations in a compensatory manner, is not suitable to manage smaller clones with finer-grained variations.
- Clone detection/analysis tools such as Gemini [UHK+02] and genealogy extractor [KN05], can play a limited compensatory role when used for just-in-time clone detection. But accuracy issues and high resource requirements limits the effectiveness of those as a clone compensation measure.
- Simultaneous editing may work in simplest of cases, but we are yet to see a good realization of this technique in practice.

Based on this analysis, we can sum up the aim of our research as an attempt to find a holistic solution to manage tenacious clones, possibly highly fragmented into patterns of similarity, so

that their genericity and maintainability is enhanced without losing their benefits. As previously mentioned (section 1.3), our research focuses particularly on the manifestation of these challenges in the web application domain.

> We need to find a holistic solution to effectively manage tenacious and possibly fragmented clones.

Note: 'Genericity' is a term frequently used in computer science literature in connection with the ability for type parameterization (e.g., [GBGM89]). We use it in a somewhat broader sense, to refer to the degree of flexibility in software that makes them amenable to extension, reuse and combination, including, but not limited to, the flexibility given by type parameterization

2.3. An overview of web application domain

This section contains a general overview of how a web application works, followed by a rather extensive review of web technologies. This information is intended as general background, and not directly pertinent to the thesis topic, a reader who is already familiar with the web domain may safely skip (or skim through) this section.

2.3.1. Web applications

A static web site is driven by a collection of HTML files. The output delivered to the browser by a static site usually does not change over time, unless HTML pages are modified intentionally. Current tools allow people with little or no formal knowledge of Software engineering to create web sites in record time. Such web sites, usually small and static, may not need to be maintained well. However, maintaining large web sites involves editing a large number of files: a laborious and error-prone task.

Today's web sites are rapidly changing from mere collections of static hypertext documents to full blown software applications, implemented by combining a variety of languages and technologies. These applications, called web applications (WA), differ from normal web sites due to the use of business logic (Business logic is a rule, or process, that affects the business state of the system. Business logic is not concerned with presentation however [Con00]). Web applications output dynamic content, though they may have a static component.

As shown in Figure 3, WAs typically follow a multi-tier client-server architecture. The client-side of a WA consists of users accessing the WA using a 'user agent' (E.g., web browser) running on a 'user device' (e.g., PC). The most common user agent configuration is a web browser running on a PC. The server-side of the WA may be organized into multiple tiers and run on a web server, possibly augmented by Application servers, Transaction monitors or Message servers.

Figure 3. Web application reference architecture

As compared to static web sites, WAs are bigger, more complex, more business critical, and closer to traditional software applications. They require bigger initial investments and hence require to live longer to payoff these investments. This makes maintainability of WAs a critical success factor. To worsen matters, maintenance of WAs poses some additional challenges as compared to traditional applications. For example, keeping the hyperlinks live all the time is an additional type of maintenance challenge that is not present in traditional

software maintenance [WBM99b]. Forced cloning of files or data, the peculiarities of file structuring in web sites, and the HTML format of combined code and data storage compounds the maintenance situation [BK01].

2.3.2. Web technologies

Web technologies change and multiply fast. For the practitioner and the researcher alike, a relatively short summary of the state of the art in web technologies could be invaluable in quickly grasping the current state of the art. To be useful, such a summary needs to be concrete enough to give sufficient details about the technologies, yet abstract enough to withstand rapid changes to concrete details. In this section we attempt to present such a summary, organized around 'technology needs'. Technology needs are both important elements in technology assessment/selection and drivers of technology proliferation and evolution. Hence, we believe that such an organization provides a perspective that is more user-oriented, fundamental and stable than the technologies themselves. We identify important technology needs of the tiers and workflows of a typical WA (*cf* WA reference architecture in section 2.3.1), and then organize the technologies into different trends that has emerged to serve these needs.

> An organization of Web technologies around technology needs provides a perspective that is more stable than the technologies themselves.

Ours is not the first attempt to ease the difficulty of comprehending web Engineering Resources (WER). For example, Christodoulou et al proposed a reference model [CP04] for organizing knowledge about WERs, with a framework [CTP03] for comparative evaluation of WERs. While the goal of Christodoulou's work and ours is the same, the methods are different, and the results - complementary to each other. Christodoulou's framework is more abstract; it does not concentrate on needs or specific technologies. Our framework is specific

about concrete details of technologies, and their relation to needs and trends. It does not require the reader to discover and assemble concrete details on their own, as is the case with Christodoulou's framework [CTP03]. Therefore, we believe our approach could be of immediate benefit to those seeking a quick overview of the web technology landscape.

Next, we present the most important WA-specific needs of the tiers and workflows of a WA, and trends in web technologies that address those needs. For each trend, we briefly mention the implementation related technologies (languages, standards, protocols, tools and techniques) that typify each trend.

The Need for Better Front-End Languages

Client-side of a WAs is primarily driven by HTML, a non-proprietary language standardized by World Wide Web Consortium (W3C)[7]. However, HTML syntax lacks the strictness of a programming language. The resulting difficulties in validating and processing HTML documents have led to a trend towards XML syntax. Extensible HTML (XHTML), the successor of HTML, is a family of document types and modules that reproduce, subset, and extend HTML, reformulated in XML[7]. Reduced authoring costs, an improved match to database and workflow applications, and clean integration with other XML applications are some of the cited benefits of XHTML[7]. Furthermore, HTML's lack of support for specialized contents has led to a number of specialized markup languages (e.g., MathML[7] - for mathematical content).

The Need to Separate Content, Structure, and Presentation

A typical HTML document is a mixture of content, structure, and presentational information. Keeping these three aspects as separate as possible is beneficial for development,

[7] World Wide Web Consortium Web site (containing home pages for CSS, HTML, XHTML, MathML, Styles, WebServices, XForms, XSL), http://www.w3.org

maintenance (as different experts could develop/maintain each separately), and reuse (as each could be reused separately). Styles[7] were added to HTML as a way to separate out presentational information. Styles describe how documents are presented on a User agent. Cascading Style Sheets (CSS)[7] is one such style mechanism that is gaining wide use. Another related technology is XSL (Extensible Style Sheets)[7], a family of recommendations for defining XML document transformation and presentation. Included in XSL is XSL Transformations (XSLT). An XSL style sheet can change the presentational as well as structural information of a document. It can be used on any XML document. XSL and CSS can be used together in a complementary manner.

The Need for a Better UI

Pure HTML UIs are static, and limited in functionality. The need to make WA UIs as sophisticated as traditional GUI applications has resulted in several trends. The first trend is to embed client-side scripts in HTML pages. JavaScript and VBScript are two languages commonly used for client-side scripting. Jscript[8] (succeeded by Jscript.NET[8]) is the Microsoft variant of JavaScript. ECMAScript[9] is a public domain specification that attempts to standardize client-side scripting. An interesting new development in this area is the AJAX (Asynchronous JavaScript and XML), a technique for creating interactive web applications using a combination of HTML (or XHTML) and Cascading Style Sheets for presenting information, Document Object Model, and JavaScript to dynamically display and interact with the information presented. AJAX is commonly used to communicate with the web server in the background without reloading the whole web page, thus bringing the user experience of WAs closer to traditional desktop applications.

---

[8] ASP, ASP.NET, COM, JScript, VBScript and .NET at, http://www.microsoft.com/

[9] ECMA home page, http://www.ecma-international.org

The second trend is embedding lightweight applications/components in HTML pages. Java applets and ActiveX controls are two technologies used for this purpose. A Java applet is a Java program that can be downloaded and executed by a browser. ActiveX controls can be run by a COM (Component Object Model) [8] aware browser and can be written in a variety of languages.

The third trend is the use of plug-ins to enable using different objects inside the browser (e.g., Adobe Acrobat plug-in allows viewing PDF documents from within browsers).

The Need for Client-Side Processing

Although WAs follow 'thin client' paradigm (minimal functionality client, more processing on server), performing some processing on the client-side (e.g., input validation on forms) can significantly reduce network traffic and improve response time. The trends for client-side processing are similar to that of the previous section, i.e., embedded client-side scripts (JavaScript, VBScript, etc.), embedded small applications (Applets, ActiveX), and plug-ins.

The Need to Use Mainstream Languages for Business Logic Processing

The bulk of the business logic processing of a typical WA happens on the server-side. Common Gateway Interface (CGI) is one standard for using mainstream programming languages to implement business logic. CGI defines how data is passed from a server to a CGI-compliant program. Two popular CGI programming languages are Perl and Python. Java is another popular language used for developing WAs. For example, Java Servlets[10] are modules of Java code that run in a server application and respond to client requests by interpreting the request, doing business logic processing, and generating dynamic content. Component technologies such as Enterprise Java Beans (EJB[10]) can further simplify server-

[10] Java Servlets, JSP, JSF, J2EE, RMI home pages at http://java.sun.com/

side programming. They facilitate reuse of common services, allowing a developer to focus on the business logic of a WA, rather than on the “plumbing” code.

The Need to Separate Response from Response Generation Code

Generating the WA response involves generating text of one language using another language (e.g., generating HTML using Perl or Java). The simplest solution is to write the server response directly to the output stream (e.g. using print() function). Java Servlets follow this method. However, this approach requires encoding each piece of the server response as a string literal, obviously a cumbersome task. Embedding scripts to represent dynamic content in otherwise static text files, commonly called ‘Server pages’, tries to separate server response from the code generating that response (scripts). The web server processes the server page and sends the generated text output to the client-side. In Server-side Includes (SSI) technique - a limited form of server pages - scripting commands embedded within a web page are parsed by the web server to generate dynamic content. SSI functionality is limited to adding small pieces of dynamic information (e.g., common footer). PHP (Hypertext Preprocessor), ASP (Active Server Pages - succeeded by ASP.NET), and JSP (Java Server Pages) are Server page technologies that are more capable than SSI. Several extensions with similar capabilities exist for Perl (e.g. Mason[11]) and Python (e.g., Spyce[12]). A further improvement is to separate the server response and scripts into separate files. Java Beans (in conjunction with JSP) and ASP.NET's Code-behind feature are some technologies that push in this direction. A successful separation of server response from code gives us Templates - representative documents one can create and edit using ordinary web authoring tools while preserving the hooks to scripts. Freemarker[13] and Velocity[14] for Java, HTML::Template[15] for Perl, Smarty[16]

---

[11] HTML::Mason home page, http://www.masonhq.com

[12] Spyce home page, http://spyce.sourceforge.net/

[13] Freemarker home page, http://freemarker.sourceforge.net/

for PHP, DTML[17] for Python, are examples of templating mechanisms. Macromedia's CFML (Cold Fusion Markup Language)[18] is another proprietary templating language.

The Need for Rapid UI Building

Unlike a traditional application where UI and the event handling code form one cohesive unit, UI of a WA needs to run on a diverse set of thin clients while communicating with the server-based event handling logic via the stateless HTTP protocol. Server-side UI component technologies are an effort to hide this complexities from the developer. They include a set of APIs for representing UI components against which it is easy to write code for managing their state, handling events, input validation etc. ASP.NET Web Forms[8] and JSF (Java Server Faces)[10] are two such server-side UI component technologies.

The Need for Integration

There are three types of integration that we can think of: intra-WA integration, inter-WA integration, and integration between WA and other external systems. The trend in intra-WA integration (integration of the remotely located parts of a WA) is to use general purpose distributed application technologies (e.g., CORBA[19], DCOM[8], .NET remoting technology[8], and Java RMI[10]). In inter-WA integration we can also use WA-specific technologies. For example, JSR-168[20] Portlet specification defines a common API for Portlets in web portals. Even more sophisticated integration could be achieved using web services[7] - programmatic

[14] Velocity home page, http://jakarta.apache.org/velocity/

[15] http://html-template.sourceforge.net/

[16] Smarty home page, http://smarty.php.net

[17] http://www.zope.org

[18] CFML home page, http://www.macromedia.com/devnet/mx/coldfusion/cfml.html

[19] CORBA home page, http://www.corba.org/

[20] JSR documentation at http://www.jcp.org

interfaces made available by a WA for communication with other WAs. Web services could be combined to create WAs, regardless of where they reside or how they were implemented. When WAs need to integrate with external non-WAs (e.g. Mail servers) the integration method depends on the mutual availability of an integration technology and a communication protocol.

The Need for End-to-End Solutions

The need for end-to-end technology solutions is based on two desires: the desire to start with a set of compatible technologies, to avoid interoperability issues, and the desire to have much of the common infrastructure ready-made and well integrated, to minimize the development effort. Platforms (underlying technological environments or architectures) and frameworks (collections of software containing specialized APIs, services, and tools) serve this need. The J2EE (Java 2 Platform, Enterprise Edition, now called JEE)[10] defines the standard for developing multi-tier enterprise applications (not limited to WAs) using Java. It provides containers for client applications, web components based on Servlets and JSP technologies, and EJB components. The J2EE Connector Architecture defines a standard architecture for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EIS). From the Microsoft camp, the .NET[8] umbrella includes a similar set of WA building technologies. It is integrated with Windows platform and has a heavy emphasis on web services. A major part of .NET is the .NET framework, which consists of the Common Language Runtime (CLR) and the .NET Framework class library. CLR provides common services for .NET Framework applications written in a variety of languages, including C, C++, C#, and Visual Basic. The .NET Framework class library includes ASP.NET, ADO.NET, and support for web services. Microsoft Host Integration Server and Microsoft BizTalk Server aid in integration of .NET WAs and other EIS. In addition, numerous other less sophisticated

frameworks exist (e.g., Seagull[21] for PHP, Mason[11] for Perl, Albatross[22] for Python, Jakarta Struts[23] Java).

Table 2. Summary of web technology trends

Need	Trends → Technologies
Better front-end languages	Incorporate XML → XHTML
	Markup for specialized contents → MathML, SVG, etc.
Separate content, structure, presentation	Styles → CSS, XSL
	Transformations → XSLT (part of XSL)
Better UI, Client-side processing	Embed client-side scripts → JavaScript, Jscript, VBScript, AJAX
	Embed light weight applications → Java Applets, ActiveX
	User agent plug-ins → e.g., Adobe plug-in for PDF
Use mainstream languages	Standards (e.g., CGI with Perl, Python, etc.)
	Components → E.g., Java Servlets, EJB, COM+
Separate response from response generation code	Write to output stream → Java Servlets
	Server pages → SSI, ASP/ASP.NET, JSP, PHP, Mason, Spyce
	Server pages (with hooks) → JSP+Java Beans, ASP.NET Code behind
	Templates → Freemarker, Velocity, Smarty, HTML::Template, DTML, CFML
Rapid UI building	Server-side UI components → ASP.NET Web forms, JSF
Integration	Regular → CORBA, RMI, DCOM, .NET Remoting
	Web specific → Portlets, Web services

[21] Seagull home page, http://seagull.phpkitchen.com/

[22] Albatross home page, http://www.object-craft.com.au/projects/albatross/

[23] Jakarta Struts project home page, http://struts.apache.org/

Need	Trends → Technologies
End-to-end solutions	Platforms/ Frameworks → J2EE, .NET Struts, Turbine, Seagull, Mason, Albatross

It should be noted that the list of technology needs given here is not an exhaustive one. It can be extended by incorporating more technology needs, such as the need for device independence, the need to make WAs secure, the need to 'internationalize', need for 'accessibility', and the need for server-side/client-side data persistence.

2.4. Web engineering Vs software engineering

There are two schools of thought on the issue on how web engineering – the process by which web applications are created – differs from traditional software engineering [Pre98][Pre00]. One feels that the speed of WA development and rapidly emerging web technologies calls for a unique approach for web engineering. The other believes the same Software engineering principles apply despite the "differences" in the web applications, or at least many of such principles can be applied though "with a different spin".

Next, we describe some of the characteristics of WAs put forward by the web engineering camp as specific (or more relevant) to WA development.

(a) **Fuzziness of requirements** [DMG+02] – Some WAs have organizational impact. One example is e-business WAs that introduce drastic changes to the business model of the organization. Requirements of such WAs are speculative rather than definitive at the initial stages.

(b) **Constant evolution/maintenance** [DMG+02] – High competitive pressure (internet has very low barriers to entry), user feedback (user base of a WAs is large and diverse), and changes to the business process of the organization continue to affect the requirements of the WAs, thus requiring a high rate of change.

(c) **Rapid technology changes** [DMG+02] – Web technologies continue to emerge at a rapid pace; WAs are continuously under pressure to incorporate the latest technologies or face the risk of being left behind.

(d) **Multi-disciplinary teams** [DMG+02] – Any non-trivial software system involve expertise from multiple disciplines, but WAs particularly require a wider range of expertise such as content authoring, multimedia and hypermedia, graphic design, security, legal, network management, information retrieval to name a few.

(e) **Lack of accepted testing process** [DMG+02] – Testing of WAs is still in an immature state, often requiring laborious manual techniques.

(f) **Dramatically shorter development life-cycles** – Some WAs represent the business tool of an organization in the internet market space where first-in-the-market advantage is significant. Such WAs typically have shorter development schedules than traditional software development [MW01][DMG+02].

(g) **Importance of aesthetics** [DMG+02] – Some WAs represent the public face of an organization in the internet community. Therefore it is a competitive advantage for such a WA to look aesthetically appealing to its target audience.

(h) **High information content** [DMG+02] – Most WAs, despite the business logic element, still has a major role to play as a source of information to the users.

(i) **Multilingual source code -** Web applications are much more multilingual than normal software, in that a single file can contain highly intertwined code written in several languages [SCD03].

(j) **Criticality of performance, scarcity of runtime resources** [DMG+02] – In the web domain, ‘slow’ is synonymous with ‘bad service’. Therefore WAs are constantly under pressure to make the best of available resources (e.g., quota of processor cycles, bandwidth, number of open connections, etc.).

(k) **Market pressure** [DMG+02] – With relatively free flow of information and low barriers to entry, existing competitors and new entrants in the web space exert tremendous pressure on organizations, and in turn their WAs.

(l) **Mixing of the best of the breed technologies** – It is common for web applications to be built using a selection of best of the breed technologies. Currently there is no single technology that can handle all aspects of a WA.

(m) **High availability requirements, 24/7 uptime** [DMG+02] – Because WAs are exposed to the all time zones of the world, truly global WAs do not have the luxury of off-peak or off-line periods.

(n) **Small teams** [DMG+02] – Due to agility requirements, web projects tend to favor small teams.

(o) **Syntactic errors in code** – Forgiving nature of browsers allow syntactically incorrect code. This allows syntax errors to persist in code, making analysis of such code difficult.

(p) **Use of special purpose programming languages** – At least some part of a WA uses special purpose languages (such as scripting, templating, or presentation oriented languages) that lack the features of a fully fledged programming language.

(q) **Some parts are deployed in source form** – Certain deployment platforms require some parts of the WA to be deployed in source form (e.g. client-side scripts).

(r) **Some parts of source code have public access** – Parts that are deployed as source code sometimes can be accessed by browsers and hence has public access.

(s) **Discoverability** – Success of some WAs (e.g., e-commerce WAs) depends on the ability of search engines to find them.

(t) **Accessibility** – Most WAs need (sometimes legally obliged) to be accessible by less-abled users.

(u) **Multimodal access** – WAs are open to be accessed with any client software/device with internet access. Usually this include a diverse and rapidly growing base of access modes including mobile phones, text messagers, PDAs, TVs, and a wide variety of web browsers.

Web Engineering is discussed as a distinct discipline in [DMG+02], [GM01a], [GM01b], and [MDHG99]. The issue of whether web engineering represents a new discipline is discussed in [Pre00], [Pre98], and [KMP+04].

2.5. Cloning in the web application domain

We believe that the web domain is a good candidate for clone research due to following reasons.

1) Web projects have dramatically shorter development life-cycles (typically shorter than three months) when compared to traditional software development [MW01]. One of the benefits of cloning is reducing the initial development time. This makes web applications ideal breeding grounds for clones. In fact, researchers have showed that cloning can be used to develop web sites in shorter time with less effort [ACDG01].
2) The lack of suitable reuse and delegation mechanisms in web technologies makes WAs a good candidates for clone proliferation [DDFG01]. For example, HTML's lack of code reuse features like 'include' directives, libraries, encapsulation, parameterization, subroutines, contribute towards cloning [SCD03][BBH98]. In WA development, additional pages are usually obtained by cloning existing pages or page components, but without explicitly documenting the cloning activity.
3) WAs have more involvement from professional software developers as compared to static web sites, and therefore a better chance of applying clone management techniques.

4) Others have reported cloning percentages of 30% or higher as common in web sites [SCD03]. Our own studies [RJ05b] found up to 63% of some WAs being contained in clones. This study also found evidence supporting our hypothesis that cloning in WAs is higher when compared to cloning in traditional software.

Different aspects of cloning have been studied, with special emphasis on the web domain. For example, some researches have defined clones in the web domain in slightly different ways. For example, Di Lucca,., Di Penta, and Fasolino [DDF02] define web pages as clones if they have the same, or a very similar, structure, i.e., the code implementing the final rendering of the page in a browser, the business rules processing, the event management, etc., is the same in the pages, while they may differ just for the information included (i.e., the information to be read/displayed from/to a user).

Some clone detection methods are specific to the web domain (e.g., [DDFG01][DDF02] [LM03][SCD03][CLM04][DST04]). Technique by Di Lucca et al [DDFG01] detects only static pages but in later [DDF02] it was extended to include ASP pages. A (semi) automatic process aimed at identifying static web pages that can be transformed into dynamic ones, using clustering to recognize a common structure of web pages, is described by Riccaand Tonella [RT03]. The work by De Lucia et al [DFST04][DFST05] detects higher level web clones made up of web page structures linked by hyperlinks. Kienle, Müller, and Weber [KMW03] observed that some clones in web are generated by tools, and hence should not be the focus of clone detection.

2.6. Chapter conclusions

Cloning problem

While most obvious clones are the simple clones (cloned code fragments), cloning can happen at higher levels, leading to cloned structures we call structural clones.

There are many reasons for creating clones.

Majority of the negative affects of clones directly impact the maintainability. Therefore, clones are generally bad for the maintenance.

In preventive clone management we use conventional reuse techniques to avoid clones altogether, or in released code. Corrective clone management can be incremental (i.e., refactor), or more drastic (i.e., rebuild), but both involves removing clones using reuse techniques. Compensatory clone management seems to be the least developed area among the three types, both in terms of conventional techniques and research.

We have identified two practical challenges in effective clone management: tenacious clones, and clone fragmentation. It is important to complement preventive and corrective measures with good compensatory measures when overcoming these challenges. However, prevailing compensatory techniques are not up to this task.

Web Engineering

Engineering web applications, while having many similarities to engineering traditional software applications, also involves a number of distinguishing considerations.

A need-oriented organization provides us a stable framework to organize and evaluate web technologies, in the face of rapidly evolving web technology landscape.

Chapter 3.

An Investigation of Cloning in Web Applications

Before software can be reusable it first has to be usable.

-Ralph Johnson

This chapter presents a study that evaluates the level of cloning prevailing in today's web applications (WAs).

As per our knowledge, no published study of cloning in the web domain is available at this point of time. In research on cloning in the web domain (e.g., [DST04][DDF02][DDFG01][LM03][RT03][SCD03]), we did not find any concrete evidence of the extent of cloning in the web domain. Particularly, it is not known how the cloning problem in the web domain compares to that in the traditional software domain. This observation encouraged us to conduct a study of cloning in the WA domain described in this section. In this study we examined 17 existing web applications drawn from diverse application domains, implemented using different technologies, having different sizes, and in different life cycle stages. We adopted a general-purpose clone detector CCFinder [KKI02] for analysis of the many types of source files that form WAs. It revealed cloning levels of 17-63%, indicating that preventive measures have failed to reduce cloning and the corrective measures have not been applied. Further analysis suggested that the cloning level in WAs is higher than that of traditional applications. These findings support our decision to focus on the web domain.

We also used, and validated, our clone evaluation framework in the study and we believe it will provide useful guidelines for future similar studies done by others, not only in the web domain, but in the other domains as well.

The organization of this chapter is as follows:

Section 3.1 describes the experiment method, and the tools, metrics, and web applications used in the study.

Section 3.2 discusses the overall cloning level in web applications as reported by the study.

Section 3.3 describes the comparison of cloning level between web applications and traditional software applications.

Sections 3.4 and 3.5 are explorations of how factors such as systems size, system age, and file type influence the cloning level.

Our contribution contained in this chapter is:

Contribution 2. It provides concrete evidence of the cloning problem in the web domain, and compares the situation with traditional applications. It also identifies similarity metrics useful for evaluating the cloning level of software

3.1. Experimental method

In this experiment, we analyzed 17 WAs[24] covering the following.

- Languages/technologies – HTML, Java, JSP, ASP, ASP.net, C#, PHP, Python, Perl, web services, proprietary template mechanisms

[24] We are unable to give a list of WAs used, due to non-disclosure agreements with some vendors who provided source code for this survey

- Application domains - collaboration portals, e-commerce applications, web-based DB administration tools, conference management, corporate intranets, bulletin boards, etc.
- System sizes - 33 ~1719 files
- License types - free, commercial, internal use,
- Development models - open source, closed source
- Life cycle stage - pre/first/post release, dead
- Usage types - off-the-shelf, one-time-use, custom-built, model applications
- Team structures - single author, centralized teams, distributed teams
- Organizations - software development companies including Microsoft, Sun Microsystems, and Apache Software Foundation, free lance software developers, in-house development teams of non-software companies

In our choice of WAs, we have tried to represent the diversity of WA domain in an unbiased manner. Due to practical limitations, the number of WAs we could include in the study was limited to 17. Although it was possible to increase the sample size by including many readily available open source WAs, we refrained from doing that, in order to keep a balance between open source WAs and (less readily available) closed source WAs. The scope of analysis was clones in any text file that is likely to be maintained by hand, including files not normally considered 'source code', such as build scripts and readme files. More than 11000 files were analyzed in total.

We used CCFinder [KKI02] as our clone detector. CCFinder can detect exact clones and parameterized clones. However, CCFinder can detect parameterized clones in only a limited number of programming languages. To our knowledge, no single clone detector that was available at the time could detect parameterized clones in *all* programming languages. Our

experiment needed to detect clones in files written in many languages, not necessarily languages supported by CCFinder. Therefore, we instructed CCFinder to assume all input files as 'plain text'. In this mode, only exact clones were detected. We also instructed CCFinder to ignore trivially short clones (i.e. clones shorter than 20 tokens) and clones occurring within the same file, in order to keep the volume of reported clones within manageable limits. We developed a Java program called 'Clone Analyzer' to control the clone detection process and to analyze the clones detected by CCFinder. Figure 4 shows the steps of clone analysis process. Next, we describe the metrics and visualizations used in the experiment.

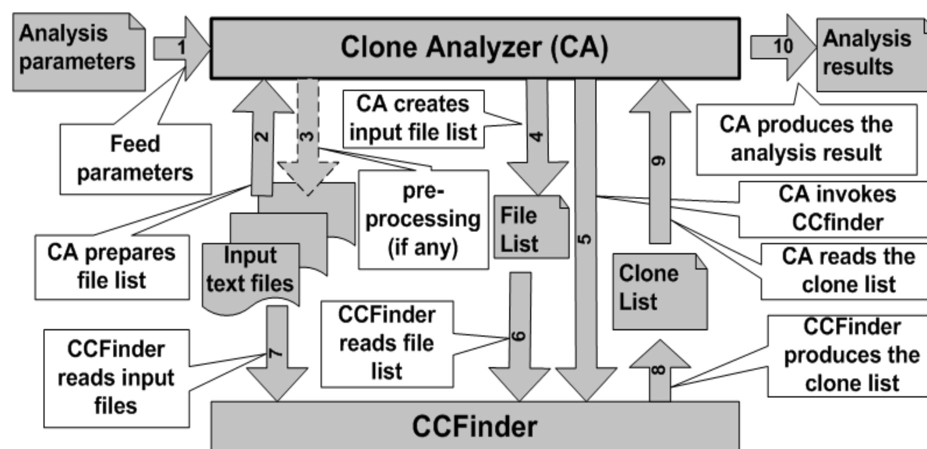


Figure 4. Clone analysis workflow

Total Cloned Tokens (TCT)

We defined TCT of a system as the sum of clone related tokens, i.e., tokens that form a part of any of the clones in that system. *TCTp* is TCT expressed as a percentage of total number of tokens in the system. When *TCTp* is high, update anomaly risk (i.e., the risk of inadvertently creating an update anomaly while modifying the system) is also high. If the *TCTp* is greater than 50%, system has more clones than non-clones; every update to the system has a higher chance of involving a clone than not; and hence runs a high risk of creating an update anomaly. We can call such systems high update anomaly risks.

File Similarity (FSA)

While *TCTp* is a good indication of the overall cloning level of a system, it can be further complemented by a measure of file similarity. For example, consider two systems *X* and *Y* of similar size, both having the same *TCTp*. In *X*, clones are scattered across the system in such a way that no two files are substantially similar. But In *Y*, clones are well concentrated into a certain set of files. From a clone treatment perspective, system *Y* is more interesting than *X* because the clones in *Y* are more easily treatable than that of *X*. To identify the similarity of a file f to other files, we calculated the metric *FSA(f)*. We defined *FSA(f)* as follows (This is analogous to *RSA(f)* defined in [UHK+02]).

$$FSA(f) = \frac{1}{Tn(f)} \sum_{c \in CF(f)} Tn(c)$$

Here, *Tn(f)* is the number of tokens in file *f*, *CF(f)* is a set of code fragments which are included in file *f* and have a clone relation with some code fragments in other files, and *c* is an element of *CF(f)*. In this summation, overlapped code portions are counted only once. *FSA(f)* is a direct measure of the similarity (resulting from cloning) of file *f* to other files in the system. For example, *FSA*=0.6 for a given file f means 60% of f has been cloned from other files of the system. For convenience, we defined the metric *FSAp* as *FSA* given as a percentage (i.e., *FSAp(f)* = *FSA(f)* * 100%)

Qualifying File Count (QFC)

We define Qualifying File Count for *FSAp* value *v*, *QFC(v)*, as the number of files for which *FSAp* is not less than *v*. For example, *QFC(30%)* gives the number of files in the system having an *FSAp* value not less than 30%. *QFCp* is *QFC* expressed as a percentage of the total number of files in the system. For example, *QFCp(60%)* = 43% means, in 43% of files in the system, 60% or more have been cloned.

File Similarity Curve (FSCurve)

To observe the overall file similarity characteristics across an entire system, we used 'File Similarity Curve' (FSCurve). An FSCurve is created by plotting *QFCp* against *FSAp*. In the example FSCurve shown in Figure 5, we have marked points A, B and C to illustrate how to interpret FSCurves. Point *A* corresponds to the fact that in 100% of files at least 0% has been cloned. At the other extreme, point *C* indicates that 40% of the files in System *X* have been completely (100%) cloned. Similarly, point *B* denotes that for System *X*, *QFC(50%)* ≈ 80%. i.e. in about 80% of the files in System *X*, at least 50% of the contents have come from other files. From FSCurves we can also get an idea about relative file similarity characteristics of different systems. For example, from the three FSCurves in Figure 5 we can clearly see that file similarity in system *Y* is generally less than that of *X* but more than that of *Z*. i.e. Higher the position of the curve, higher the file similarity.

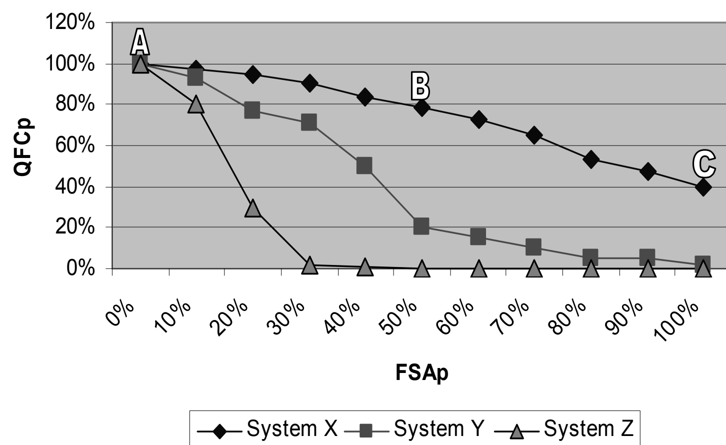


Figure 5. Sample FSCurves

3.2. Overall cloning level

The initial phase of our investigation was focused on the overall cloning level in WAs. Given in Figure 6 is the *TCTp* of each WA we studied. Only one WA has a *TCTp* below 20%. The average *TCTp* is 41% (with a standard deviation of 15%). Five WAs are high update anomaly risks (*TCTp*>50%) while three more are close behind.

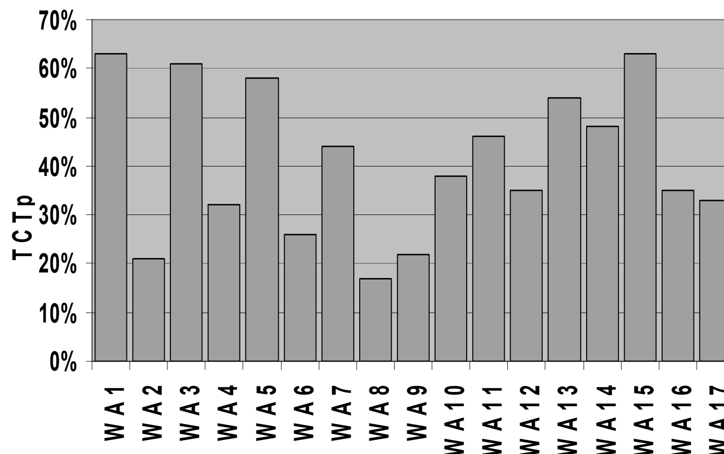
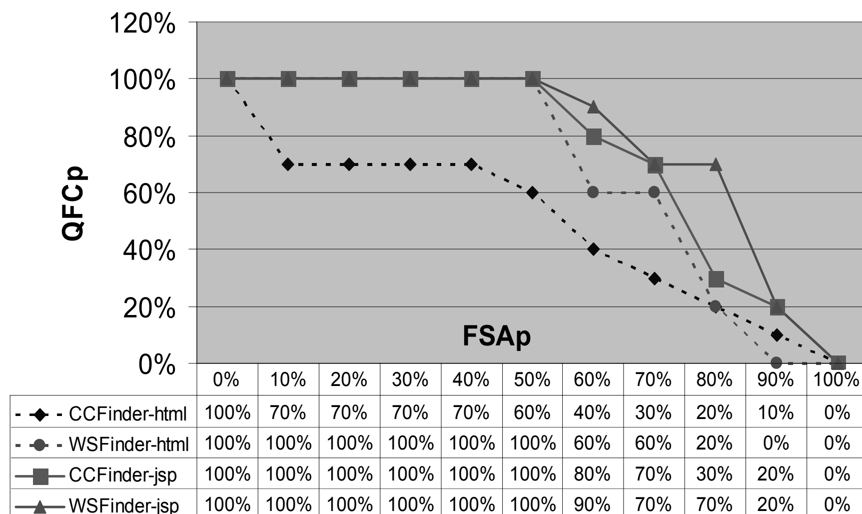


Figure 6. Cloning level in each WA

From these data alone, the level of cloning in WAs seems substantial. Still, these data do not include clones with parametric variations (parameterized clones) and non-parametric variations (gapped clones). As a result, the actual cloning level in WAs could be even higher than the levels indicated by these data. We tested this hypothesis by comparing cloning level reported by CCFinder and a web-specific clone detector described in [DST04]. This clone detector (for convenience, we refer to it as WSFinder) detects the similarity among web-specific files. We did not use it as our main clone detector because it currently supports HTML and JSP files only. WSFinder reports three different values of file similarity based on 1. HTML tags, 2. Text included inside HTML tags, and 3. Scripts included in the file (only applicable to JSP pages). For a small set of HTML and JSP pages, we applied both CCFinder and WSFinder to compare results. To make the comparison least biased towards the hypothesis, we compared the minimum of the three values reported by WSFinder against CCFinder results. As shown in Figure 7, CCFinder almost always reported a cloning level less than or equal to that reported by WSFinder. This supports our hypothesis that actual cloning level in WAs could be even higher than what is reported here.



	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
CCFinder-html	100%	70%	70%	70%	70%	60%	40%	30%	20%	10%	0%
WSFinder-html	100%	100%	100%	100%	100%	100%	60%	60%	20%	0%	0%
CCFinder-jsp	100%	100%	100%	100%	100%	100%	80%	70%	30%	20%	0%
WSFinder-jsp	100%	100%	100%	100%	100%	100%	90%	70%	70%	20%	0%

Figure 7. CCFinder Vs WSFinder

A high level of cloning does not necessarily mean a high reuse potential. The clones detected could be too small, too dispersed, or false positives. Since our minimum clone length was 20 tokens, these results could include clones as short as 20 tokens. (We did not use a higher minimum clone length, in the hope of capturing some of the parameterized clones or gapped clones; a parameterized/gapped clone contains a number of smaller exact clones). This could prompt one to argue that clones detected are trivially short ones, not worthy of elimination. To address this concern, we used the breakdown of the clones by length, in each system (as shown in Figure 8). Clone size increases from 20 to 100+ as we go from top to bottom of each bar. Increasingly larger clones are shown in increasingly darker colors. As an average LOC is accounted by 6-8 tokens, a 100 token clone is roughly 15 LOC long. Therefore, this graph shows that most clones we detected are longer than 15 LOC.

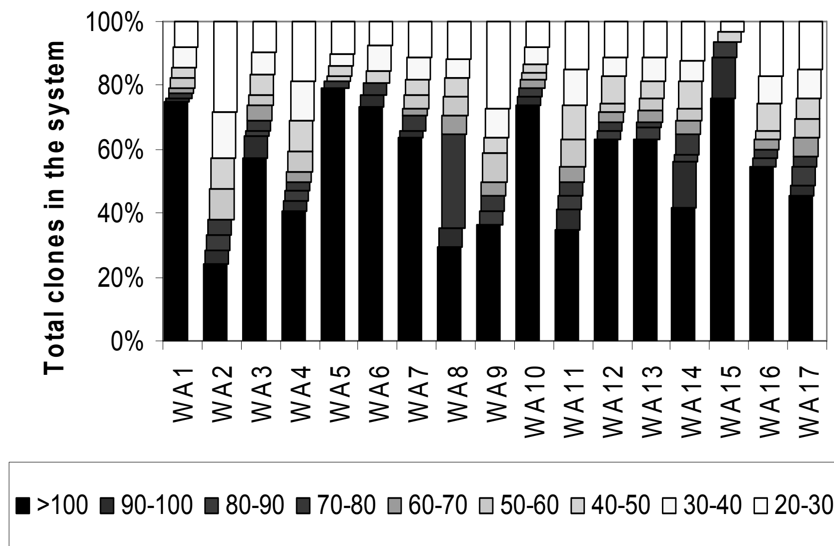


Figure 8. Distribution of clone size

To address the issue of clones dispersed across the system too thinly, we generated FSCurves for each system. To save space, we show all the FSCurves together in Figure 9, with the average, the minimum, and the maximum curves marked with dashed lines. According to the average curve, close to 50% of the files have at least 50% of their content cloned. Figure 10 represents two cross sections of Figure 9, namely, at *FSAp*=50% and *FSAp*=90%. We use this graph to give a bit more detailed view of the clone concentration in each WA. It shows the percentage of files in each system that we can consider 'cloned' (*FSAp*≈50%) and 'highly cloned' (FSAp≈90%). In eleven of the WAs, we find more than 10% of the files have been highly cloned. In five, we find more than 20% of the files have been highly cloned. Aggregating all the WAs, the percentages of cloned and highly cloned files are 48% and 17% respectively. These data suggest that there is good clone concentration in files.

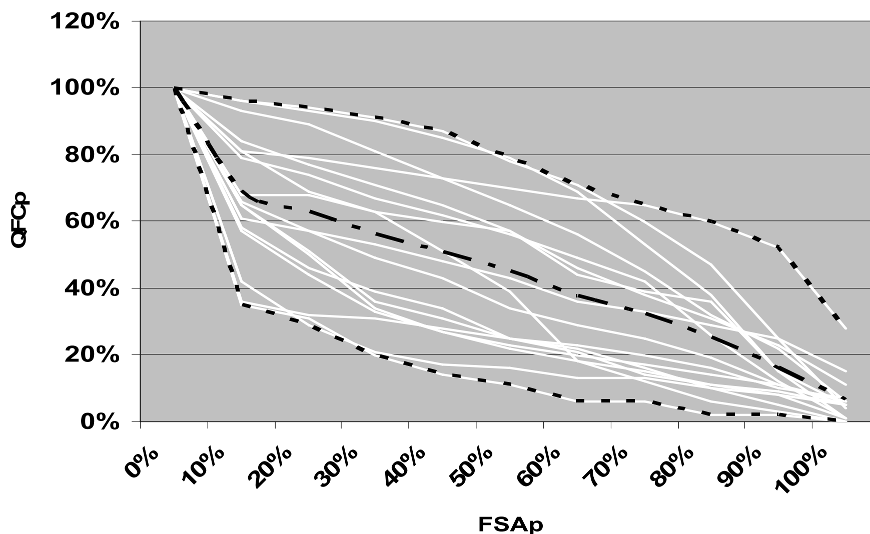


Figure 9. FSCurves for all WAs

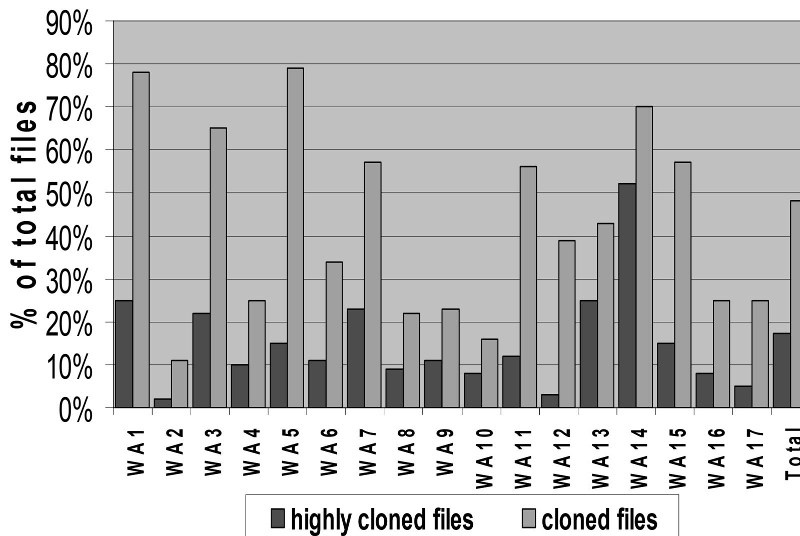


Figure 10. Percentage of cloned files

With regards to the issue of false positives, it is not practical to manually weed out the false positives in a study of this scale. However, since we detected only exact clones, we believe the false positives are at a minimum.

3.3. Cloning level in WAs Vs cloning level in traditional applications

Since cloning in Traditional Applications (TAs) has been widely accepted as a problem, we wanted to compare cloning levels of WAs to that of TAs. We started by separating the files in WAs into two categories:

- WA-specific files - files that use WA-specific technologies, e.g., style sheets, HTML files, ASP/JSP/PHP files
- General files - files equally likely to occur in WAs and TAs. e.g., program files written in Java/C/C#, build scripts

We found 13 of the WAs had both type of files, while some smaller WAs had only web-specific files. For WAs with both type of files, we calculated *TCTp*_W (*TCTp* for WA-specific files) and *TCTp*_G (*TCTp* for general files) as given in Figure 11. The last two columns show that overall *TCTp*_W was 43% and overall *TCTp*_G was 35%. The *TCTp* comparison of individual WAs shows that in 6 WAs *TCTp*_W is significantly higher (*TCTp*_W > *TCTp*_G by more than 10%), in 3 WAs levels are similar (|*TCTp*_W-*TCTp*_G| ≤10%), and only in 4 WAs *TCTp*_G was significantly higher (*TCTp*_G > *TCTp*_W by more than 10%). These figures suggest that WA-specific files have more cloning than general files. But we can reasonably assume that cloning in full fledged TAs is not worse than cloning in these general files. This infers that cloning in WAs is worse than cloning in TAs.

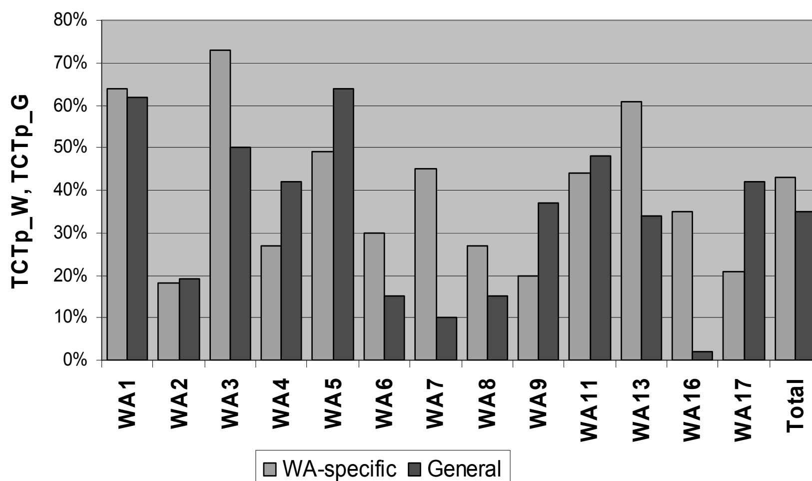


Figure 11. WA-specific files Vs general files

3.4. Factors that affect the cloning level

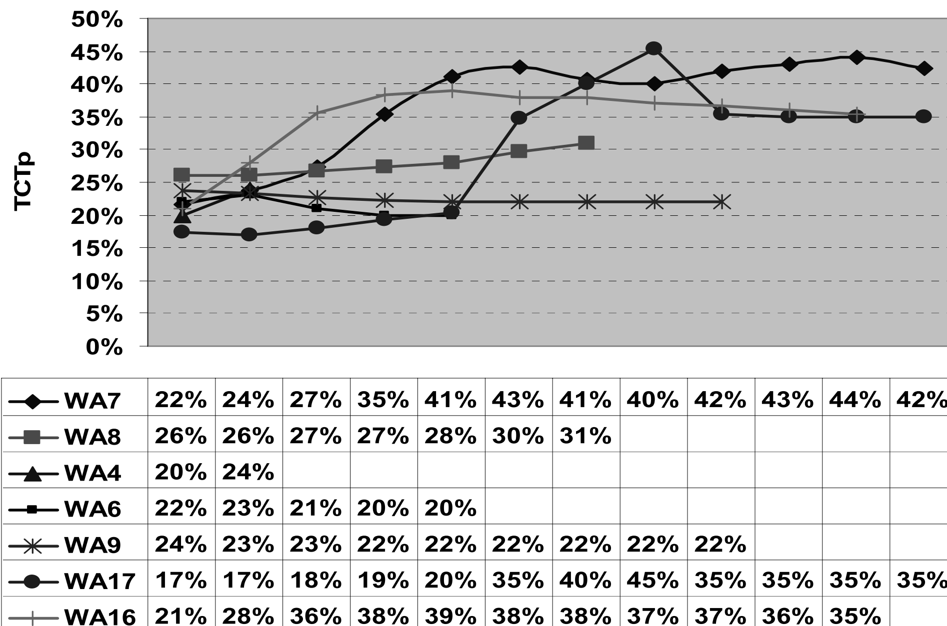
Our investigation also included collecting quantitative data on different factors that might affect cloning in WAs. We started by investigating whether system size has any effect on the cloning level. However, a comparison of average cloning level in small, medium, and large WAs (Table 3) showed that cloning level does not significantly depend on the system size.

Table 3. Average cloning for WAs of different size

Size (in # of files)	Avg *TCTp*	Std. Deviation
Small (size < 100)	40%	21%
Medium (100 ≤ size < 1000)	42%	14%
Large (size ≥ 1000)	40%	16%
All	41%	15%

Continuing, we also investigated the progression of cloning level over time. For this, we used seven of the WAs for which at least four past releases were readily available. All seven suitable WAs were open source, and of medium or large size. In the Figure 12, we show the moving average (calculated by averaging three neighboring values) of *TCTp* over past versions, up to the current version. According to this graph, all WAs show an initial upward trend in the cloning level. Some WAs have managed to bring down the *TCTp* during the latter

stages, even though current levels still remain higher than the initial levels. This indicates that the cloning level is likely to get worse over time. WA9, and to a smaller extent WA6, are the only exceptions, but this may be due to non-availability of the versions corresponding to the initial stage.



WA7	22%	24%	27%	35%	41%	43%	41%	40%	42%	43%	44%	42%
WA8	26%	26%	27%	27%	28%	30%	31%					
WA4	20%	24%										
WA6	22%	23%	21%	20%	20%							
WA9	24%	23%	23%	22%	22%	22%	22%	22%	22%			
WA17	17%	17%	18%	19%	20%	35%	40%	45%	35%	35%	35%	35%
WA16	21%	28%	36%	38%	39%	38%	38%	37%	37%	36%	35%	

Figure 12. Movement of cloning level over time

3.5. Identifying the source of clones

Finally, we attempted to obtain some quantitative data that could be useful for devising a solution to the cloning problem. We were interested to find which of the following file categories contributed most clones

i. Static files (STA) - files that needs to be delivered 'as is' to the browser. Includes markup files, style sheets and client side scripts (e.g., HTML, CSS, XSL, and JavaScripts).

ii. Server pages (SPG) - files containing embedded server side scripting. These generate dynamic content at runtime (e.g., JSP, PHP, ASP, and ASP.NET).

iii. Templates (TPL) - files related to additional templating mechanisms used.

iv. Program files (PRG) - files containing code written in a full fledged programming language (e.g., Java, Perl, C#, Python)

v. Administrative files (ADM) - build scripts, database scripts, configuration files

vi. Other files (OTH) - files that do not belong to other five types.

Figure 13 gives the contribution of each file type towards system size while Figure 14 gives the contribution of each file type towards cloning. The rightmost column of each graph shows the overall situation (aggregation all the WAs). The salient feature of these graphs is that there is no single file type that clearly dominates the composition of the system, or the composition of the clones. At least three types (STA, SPG, and PRG) shows dominant influence, while the influences of TPL and ADM are smaller, but not negligible. This shows that a successful solution to the cloning problem has to be applicable equally to the entire range of file types. Moreover, the high influence of WA-specific types (STA, SPG and to a lesser extent, TPL) suggests that a solution rooted in TAs might not be successful in solving the cloning problem in WAs.

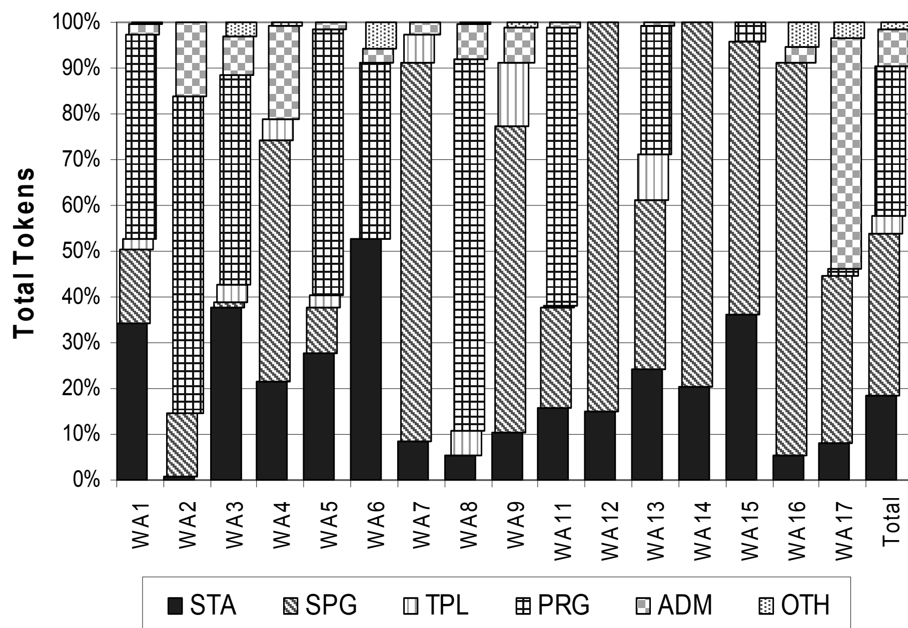


Figure 13. Contribution of different file types to system size

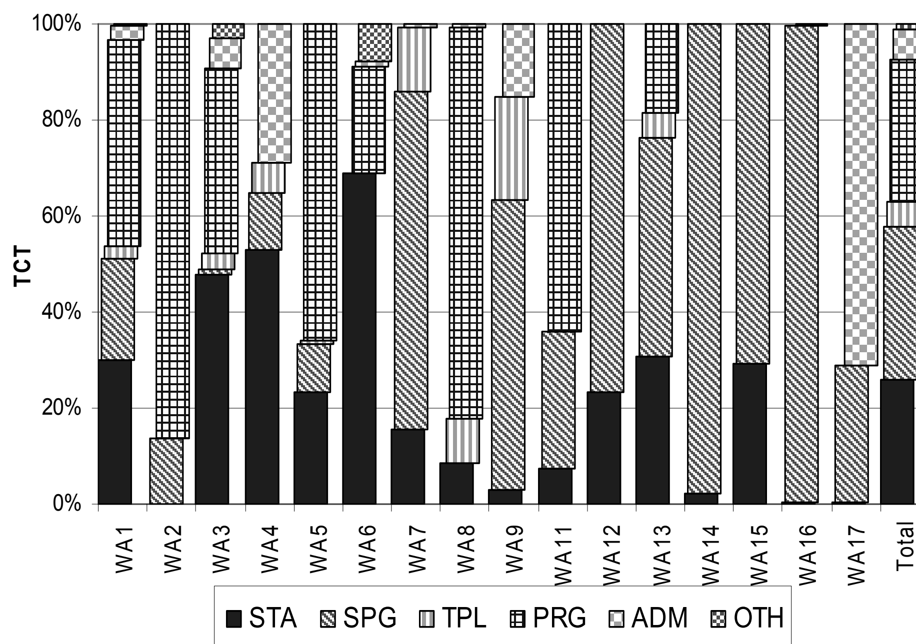


Figure 14. Contribution of different file types to cloning

3.6. Chapter conclusions

Our study found cloning rates 17-63% in both newly developed and already maintained web Applications. Most of the clones are substantially long, well concentrated and unlikely to be false positives.

With the aid of a web-specific clone detector, we substantiated our hypothesis that actual cloning level could be even higher than the levels reported here. We also showed that cloning equally affect small, medium or large WAs, and cloning gets worse over time.

More importantly, we showed that cloning in WAs could be even worse than that of traditional applications.

Our findings provide the concrete evidence of cloning in WAs we set out to produce at the start of this study. In doing so, we confirm the potential benefits of addressing cloning problem in the web domain.

Chapter 4.

More Evidence of Tenacious Clones

Even the best planning is not so omniscient as to get it right the first time.

- Fred Brooks

This chapter describes two case studies in which we found a number of tenacious clones in two popular public domain libraries: Java Buffer library, and the C++ Standard Template Library. These case studies were done before we narrowed our focus to the web application (WA) domain. Although these were not WAs, the issues raised are still applicable to WAs, and we think these two case studies are sufficient to support our claims about tenacious clones, not just in WAs, but in a much wider context.

In a previous case study [JL03] it was found that at least 68% of the code in the Java nio.* Buffer library was contained in clones. The Buffer library was built before generics was introduced to Java. An interesting question is how much of the cloned code could be eliminated by applying generics? In our first case study we looked into this problem. We observed that type variation triggered many other non-type parametric differences among similar classes that could not be handled by Java generics.

For the second case study, we chose the SGI implementation of the C++ Standard Template Library (STL)[25]as it provides a perfect case to strengthen the observations made in the first case study. Firstly, parameterization mechanism of C++ templates is more powerful than that

[25] http://www.sgi.com/tech/stl

of Java generics. Secondly, the STL is widely accepted in the research and industrial communities as a prime example of the generic programming methodology. Still, we found much cloning in the STL that have escaped C++ template mechanism.

Our overall observations is that while generics provide an elegant mechanism to unify a group of similar classes through parameterization, in practice, there are many tenacious clones that also call for generic solutions, but available generics mechanisms are not capable enough to tackle them.

The organization of this chapter is as follows:

Section 4.1 briefly describes the first case study involving the Java Buffer library.

Section 4.2 similarly describes the case study involving the STL.

Section 4.3 summarizes the types of tenacious clones we found in the two case studies.

Author wishes to acknowledge that this chapter is based on two joint papers [BRJ05a][BRJ05b] by the Author, Basit, H. A., and Jarzabek, S.

Our contribution contained in this chapter is:

Contribution 3. It shows more evidence of tenacious clones using two case studies (this is a joint contribution with Basit, H. A.)

4.1. Case study 1: Java Buffer library

The Buffer library in our case study is part of the java.nio.* package in JDK since version 1.4.1. The concept 'buffer' refers to a container for data to be read/written in a linear sequence. The (partial) class diagram of the Buffer library given in Figure 15 shows the explosion of the many variant buffers that populates the library, a classic incarnation of the feature combinatorics problem [Big94]. Even though all the buffer classes play essentially the

same role, there are 74 classes in the Buffer library (In this analysis, we only consider the 66 classes that contribute to code duplication, leaving out helper classes, exception classes etc.).

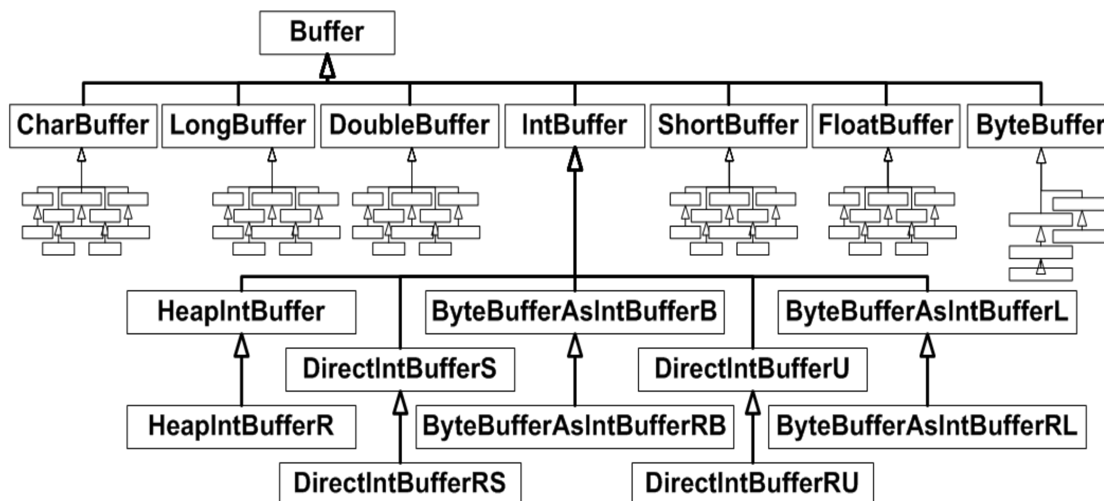


Figure 15. Partial class hierarchy of Buffer library

Feature diagrams [Kan90] are a common approach used in domain analysis to illustrate the variability of a concept. Feature diagram for the Buffer library is given in Figure 16. It shows four mandatory feature dimensions and one optional feature dimension. 'Element Type' (T) is a mandatory feature dimension that represents the type of elements held in the buffer. It has seven alternative features corresponding to seven valid element types: `int`, `short`, `double`, `long`, `float`, `char` and `byte`. To describe the feature diagram, we use the concept of 'peer classes':

Peer classes

> Peer classes is a set of classes that differ along a given feature dimension only. For example, classes `HeapIntBuffer` and `HeapDoubleBuffer` are peers along the Element type dimension because the only variation between the two buffers is element type.

Feature dimension 'Access Mode' (AM) has two alternative features corresponding to read-only buffers and writable buffers respectively. Writable `HeapByteBuffer` and read-only `HeapByteBufferR` are peers along this dimension.

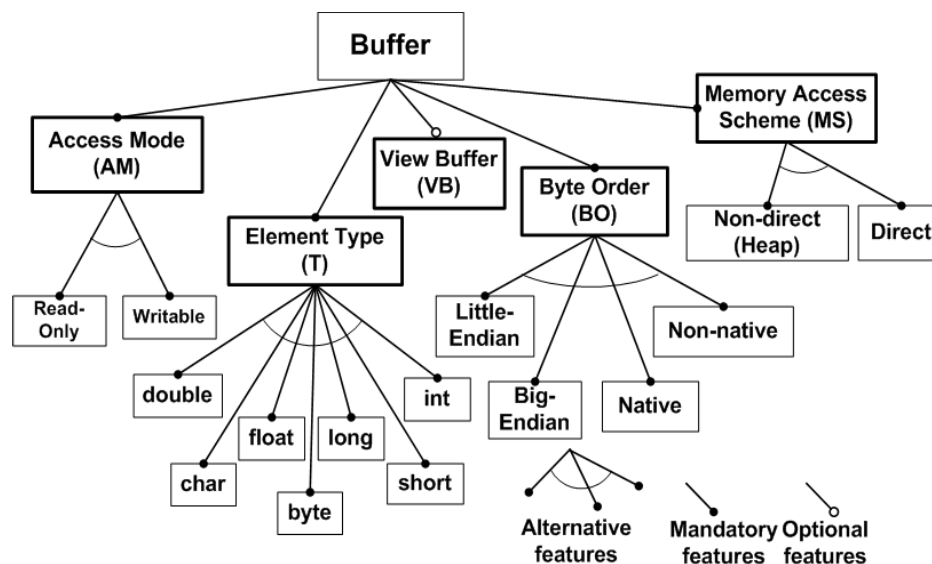


Figure 16. Feature diagram for Buffer library

Other feature dimensions with alternate features are 'Memory Access Scheme' (MS) and 'Byte Order' (BO). Feature dimension 'View Buffer' is optional. Each legal combination of these feature dimensions yields a unique buffer class. (e.g., `DirectIntBufferRS`, represents the combination T = int, AM = read-only, MS = direct, BO = non-native, and VB = false)

Analysis method

For this case study, we manually analyzed the Buffer library to identify groups of similar buffer classes. Then, we studied differences among classes in each group, and attempted to unify groups of similar buffers with suitable Java generics. Upon careful observation of the Buffer library, we found that only 15 buffer classes fall neatly into the generics-friendly layout and could be replaced by 3 generic classes `Buffer<T>`, `HeapBuffer<T>`, and `HeapBufferR<T>` (The solution can be viewed at [XVCL]). According to this result, generics can reduce only 40% (calculated in terms of physical LOC, excluding comments, blank lines and trivially short lines) of redundant code from the Buffer library. This solution still relies on wrapper classes for primitive types (as Java generics do not allow parameterization with primitive types).

A detailed analysis of the different types of tenacious clones that we encountered in the Buffer library will be given in section 4.3. More information on this case study can be found at http://xvcl.comp.nus.edu.sg/xvcl_cases.php.

4.2. Case study 2: Standard Template Library

The Standard Template Library (STL) is a general-purpose library of algorithms and data-structures. It consists of containers, algorithms, iterators, function objects and adaptors. Most of the basic algorithms and structures of computer science are provided in the STL. All the components of the library are heavily parameterized to make them as generic as possible. A major part of the STL is also incorporated in the C++ Standard Library.

Generic containers form the core of the STL. These are either sequence containers or associative containers. Among the containers, we selected the associative container slice for detailed analysis because of its high level of cloning. Feature diagram of Figure 17 depicts features of associative containers in the STL. 'Ordering', 'Key Type' and 'Uniqueness' are the feature dimensions. Any legal combination of these features yields a unique class template (eight in total). For example, the container `set` represents an associative container where [Storage=sorted], [Uniqueness=unique], and [Key type=simple].

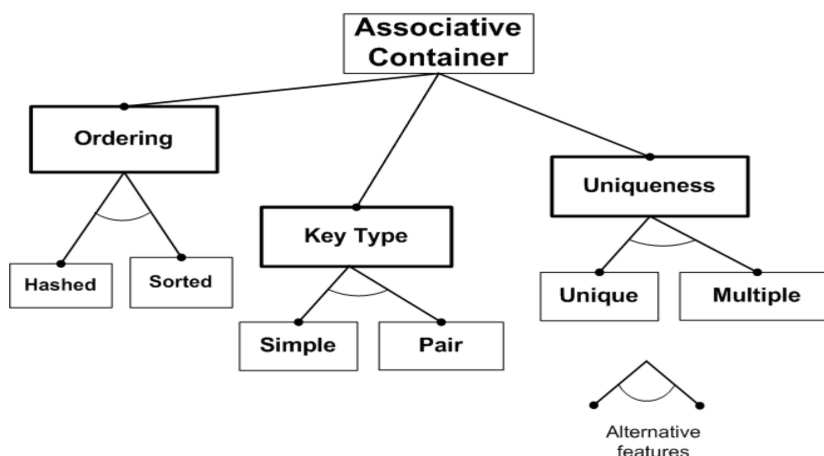


Figure 17. Feature diagram for associative containers

Analysis method

We analyzed the STL code from the SGI website. For clone detection we used CCFinder [KKI02] and Gemini [UHK+02]. Having identified clones, we studied the nature of variations among them, and tried to understand the reasons why cloning occurred.

In our analysis of associative containers, we found that if all four 'sorted' associative containers and all four 'hashed' associative containers, were unified into two generic containers, the Reduction in Related Code (RRC) is 57%. RRC is an approximation calculated by comparing the LOC of clones before and after a meta-level unification. A detailed description of this meta-level unification is presented in [BRJ05a]. In container adaptors - stack, queue and priority queue - we found that 37% of the code in stack and queue could possibly be eliminated through clone unification. Cloning in the algorithms (in file `stl_algo.h`) was localized to the set functions, i.e., we found that set union, intersection, difference, and symmetric difference (along with their overloaded versions) form a set of eight clones that could be unified into one (RRC=52%). Iterators were relatively clone-free, but the supporting files `type_traits.h` and `valarray` exhibited excessive cloning. In the `type_traits.h` header file, a code fragment had been cloned a remarkable 22 times (RRC=83%).The header file `valarray` contained eight different code fragments that had been cloned between 10 to 30 times each (137 times in total, where RRC=83%).

4.3. Examples of tenacious clones

In this section, we illustrate the situations in the two case studies, in which the generics were unable to unify similar program structures.

Non-parametric variations

It is common to find non-parametric variations in code. Extra or missing code fragments between similar program structures are such variations not addressed by generics. For example, `CharBuffer` of the Buffer library has some additional methods not present in other buffer types. On the other hand, `DirectByteBuffer` is missing a method common to all its peers. 'Extra' or 'missing' code fragments can be of any granularity as shown by the next example.

`CharBuffer` class implements an extra interface none of its peers implement, resulting in the class declaration code shown in first part of Figure 18. Now compare it with the declaration clause of `DoubleBuffer` given second to note the offending extra bit of code in `CharBuffer`. Figure 19 provides an example of a non-parametric variation of keywords between iterators for `Map` and `Set` in STL.

```
...
public abstract class CharBuffer
    extends Buffer implements Comparable, CharSequence{
...
```

```
...
public abstract class DoubleBuffer
    extends Buffer implements Comparable {
...
```

Figure 18. Declaration of class CharBuffer and DoubleBuffer

```
  iterator begin() const {  return _M_t.begin();  }
```

```
  iterator begin(){ return _M_t.begin();  }
```

Figure 19. Keyword variation example

Some algorithmic differences are too extensive to be parameterized. For example, `toString()` method of `CharBuffer` differs semantically from `toString()` method of its peers, as shown in Figure 20.

```
//In CharBuffer:
public String toString() {
 return toString(position(), limit());}
```

```
//In IntBuffer,FloatBuffer,LongBuffer etc.
    public String toString() {
    StringBuffer sb = new StringBuffer();
      sb.append(getClass().getName());
      sb.append("[pos=");
      ...
      sb.append("]");
      return sb.toString(); }
```

Figure 20. Method toString() of CharBuffer and its peers

Due to this reason, we cannot use generics to unify `CharBuffer` with its peers despite the similarity of the rest of the code. A solution based on inheritance looks feasible, but not without adding another layer to the already complex inheritance hierarchy. Another option is to use template specialization, but Java generics do not support this feature.

```
template <class _Tp>
inline valarray<_Tp> operator+(
               const valarray<_Tp>& __x, const _Tp& __c) {
    typedef typename valarray<_Tp>::_NoInit _NoInit;
    valarray<_Tp> __tmp(__x.size(), _NoInit());
    for (size_t __i = 0; __i < __x.size(); ++__i)
        __tmp[__i] = __x[__i]  + __c;
    return __tmp;}
```

```
template <class _Tp>
inline valarray<_Tp> operator+(
              const _Tp& __c, const valarray<_Tp>& __x) {
    typedef typename valarray<_Tp>::_NoInit _NoInit;
    valarray<_Tp> __tmp(__x.size(), _NoInit());
    for (size_t __i = 0; __i < __x.size(); ++__i)
      __tmp[__i] = __c + __x[__i];
    return __tmp;}
```

Figure 21. Clones due to swapping

One interesting type of non-parametric variation we spotted in STL is due to swapping of code fragments in order to make overloaded operators symmetric. Figure 21 gives an example. Note how the parameter pair (`const valarray<_Tp>&, const _Tp& __c`) and operand pair (`__x[__i], __c`) are swapped between the two clones.

Non-type parametric variations

Some parametric variations cannot be represented by types and hence cannot be unified using Java generics. A prime example of a non-type parametric variation is constants. The clone given in Figure 22 is repeated several times inside the Buffer library with different constant values (2, 3, and 4) for @size.

```
private long ix(int i) {
  return address + (i << @size);
}
```

Figure 22. Generic form of method ix()

Though parameterization using constants is supported in C++, the question remains whether we should force the user to specify this parameter manually when the value is inferable from another type parameter. One solution is to use traits template idiom [Mye95], at the expense of increased complexity, to encode the type dependent information into the type and pass it as a parameter.

Another parametric variation not supported by generics is keywords. In stl_iterator.h, the clone given in Figure 23, @access was private in one instance while it was protected in the other (a possible case of inconsistent updating).

```
template <class _Tp @moreParams  >
class ostream_iterator {
public:
  …
  ostream_iterator<_Tp>& operator*() { return *this; }
  ostream_iterator<_Tp>& operator++() { return *this; }
  ostream_iterator<_Tp>& operator++(int) { return *this; }
@access:
  @streamType* _M_stream;
  const @stringType* _M_string;
};
```

Figure 23. Access level variation example

At times, code fragments differed in operators, as illustrated in the example from the Buffer library shown in Figure 24. @operator is '==' in DirectDoubleBufferS but it is '!=' in DirectDoubleBufferU. Such variations also cannot be unified with Java generics.

```
public ByteOrder order() {
return ((ByteOrder.nativeOrder() @operator
    ByteOrder.BIG_ENDIAN)?ByteOrder.LITTLE_ENDIAN:
         ByteOrder.BIG_ENDIAN);}
```

Figure 24. Generic form of method order() in direct buffers

An indirect solution of the above problem can be through function objects. The different operators can be turned into function objects and passed on to the generic class as a parameter. But this indirect solution may create more clones among the different function objects.

A similar problem is found in STL with operator overloading in the associative containers of STL. Figure 25 shows a generic form of such clones. `@op` was replaced by different operators (e.g. `'=='`, `'<'` etc.) in different instances of the clone. Since these code fragments relate to operator overloading, function objects cannot be used to unify these clones.

```
template <class _Key, class _Compare, class _Alloc>
             inline bool operator@op (
                    const set<_Key,_Compare,_Alloc>& __x,
                    const set<_Key,_Compare,_Alloc>& __y) {
      return __x._M_t @op__y._M_t;
}
```

Figure 25. A clone that vary by operators

Also, copyright notices that appear in all STL files exhibit non-type parametric variations.

Restrictions on type-parametric variations

Type parametric variations between code fragments are the ideal targets for code reuse through generics. Yet idiosyncrasies of generic implementations can sometimes get in the way, even in these ideal situations. For example, parameterization using primitive types (int, short, long, double, etc.) is not allowed in Java.

In STL iterators, we found another case of restrictions on type parameters for templates. In this clone (shown in Figure 26) the only variation point `@type` is a type (`int`, `float`, `long`,

`bool`, `char`, `short` … 22 types in all). These clones are template specializations for 22 types. Therefore, they cannot be unified by usual template techniques.

```
__STL_TEMPLATE_NULL struct __type_traits<@type> {
    typedef __true_type    has_trivial_default_constructor;
    typedef __true_type    has_trivial_copy_constructor;
    typedef __true_type  has_trivial_assignment_operator;
    typedef __true_type    has_trivial_destructor;
    typedef __true_type    is_POD_type;
};
```

Figure 26. Generic form of a clone found in ‘type_traits.h’

Coupling

Coupling among classes and modules can also play a role in restricting the use of generics. Given in Figure 27 is an example of this situation from the Buffer library.

```
 public int get(int i) {
  return Bits.swap(
    unsafe.getInt(ix(checkIndex(i))));}
```

```
public float get(int i) {
  return Bits.swap(
   unsafe.getFloat(ix(checkIndex(i))));}
```

Figure 27. Method get(int) of DirectIntBufferS and DirectFloatBufferS

To unify these two methods into a generic method, we need to unify `getInt()` and `getFloat()` methods as well. Sometimes this is not possible: these two methods can be out of scope or they can be generics-unfriendly. Now, we have two ways to proceed. The first is to convert the variant functions into function objects and ask the user to furnish the required function object as a parameter. But this breaks the basic design, since this parameter is not one of the feature dimensions. The second is to find a way (possibly using run-time type information) to infer the proper function to call based on the type parameter. This will introduce further indirections and runtime overheads.

4.4. Chapter conclusions

As shown by these two case studies, even well designed software can contain clones that are difficult to unify within the confines of conventional clone unification techniques.

Chapter 5.

Mixed-Strategy

Simple things should be simple and complex things should be possible.

-Alan Kay

This chapter describes the mixed-strategy, and the XVCL meta-programming language which is at the core of the mixed-strategy.

When the conventional clone unification techniques fail to unify certain clones, it is worthwhile to look for non-conventional solutions. Mixed-strategy, an approach our research group has been working on since 2000, advocates complementing the conventional methods with the meta-programming technique of XVCL (XML-based Variant Configuration Language) [XVCL]. With the aid of XVCL's powerful parameterization and composition capabilities, mixed-strategy can effectively unify (at meta-program level) any clone deemed 'non-unifiable' using conventional techniques, and effectively tackle intentional clones.

Section 5.1 introduces the basic concepts of XVCL meta-programming language.

Section 5.2 gives an overview of how the mixed-strategy works.

Section 5.3 outlines benefits and drawbacks of using mixed-strategy.

Section 5.4 lists some past case studies in which mixed-strategy showed promising results.

Section 5.5 summarizes how mixed-strategy helps us to attack non-unifiable clones and intentional clones, which are types of tenacious clones.

Section 5.6 discusses mixed-strategy's applicability to the web application domain.

5.1. Introduction to XVCL

As XVCL (XML-based Variant Configuration Language) is at the core of the mixed-strategy, it is best to introduce XVCL before going on to describing the mixed-strategy. Following description of XVCL has been adapted from the XVCL website [XVCL].

> XVCL is a meta-programming language that adds unrestrictive parameterization and 'composition with adaptation' mechanism to base programming languages.
>
> **Basset frames: precursor to XVCL …**
>
> In 1979, Bassett applied frame concepts in software engineering context, giving rise to Frame Technology™ [Bas97]. Bassett's frames represent generic, reusable software building blocks as parameterized structures. In addition, Bassett's frames are active, in the sense that they specify rules for adapting and composing other frames to form custom programs. Netron Inc. achieved considerable success in applying Frame Technology to evolve multi-million-line, COBOL-based information systems. An independent analysis showed that Frame Technology has reduced large software project costs by over 84% and their times-to-market by 70%, when compared to industry norms at the time [Bas97].
>
> **Enter XVCL…**
>
> XVCL was developed in 2000 at the Software Engineering Lab of National University of Singapore, in a joint Singapore-Ontario research project between National University of Singapore, ST Electronics, University of Waterloo and Netron Inc. XVCL refines original Basset frames into a general-purpose method that blends with contemporary programming and design paradigms. The current form of XVCL can be seen as an assembly language for generic design. XVCL's explicit and direct

articulation of similarities and variations is the source of its expressive power, e.g., we can unify arbitrary types of variations across similar program structures.

In general, we can apply XVCL on top of any artifact that has a textual representation, independently of the language syntax and semantics, and no matter what the language is used for. XVCL structures express the syntax and semantics of similarities and variations, not the syntax and semantics of programs.

X-frames and x-frameworks …

Code written in a programming language is partitioned into XVCL structures called 'x-frames'. An x-frame may correspond to any program unit (or part of it) such as a subsystem, interface, program component, group of classes, class, method, attribute declarations, fragment of method implementation – just anything. XVCL's independence from the semantic constraints of the underlying program allows us to unify (at meta-program level) clones that are non-unifiable at the program level. An XVCL solution consisting of a set of interrelated x-frames is called an 'x-framework'.

X-frames turn conventional program units into generic, adaptable and reusable program building blocks: A small number of x-frames can represent many similar instances in the target application – thus enabling us to use it as a clone unification mechanism. Partitioning of the code into x-frames can be guided by clone unification concerns and need not be constrained by the rules of the underlying programming language.

XVCL's independence from the semantic constraints of the underlying program allows us to unify (at meta-program level) clones that are non-unifiable at the program level.

A simple XVCL example

Now let us look at a simple example of applying XVCL to unify a clone. Figure 28, and Figure 29 shows two instances of a clone found in the IntBuffer.java and DoubleBuffer.java respectively, from Java Buffer library (explained in section 4.1). The two methods differ in a primitive type parameter (shown in bold). Since Java generics cannot handle variations of primitive types, we apply XVCL to unify this clone at meta-program level.

```
public final int[] array() {
  if(hb == null) throw new UnsupportedOperationException();
  if(isReadOnly) throw new ReadOnlyBufferException();
  return hb;
}
```

Figure 28. array() method for int – found in IntBuffer.java

```
 public final double[] array() {
  if(hb == null) throw new UnsupportedOperationException();
  if(isReadOnly) throw new ReadOnlyBufferException();
   return hb;
}
```

Figure 29. array() method for double – found in DoubleBuffer.java

Figure 30 shows the unification of these two clones. We use a simplified (XML-free) notation in these examples to reduce clutter. First, we create an x-frame called array to represent the generic array method. It consists of the common code of the cloned method, where the variation point is marked with an XVCL variable called ElementType (marked as @ElementType in the figure). In the x-frames that represent IntBuffer and DoubleBuffer, we replace the cloned array method with an <adapt> command[26] that

[26] <adapt> is the XVCL command used to include contents of a specific x-frame (or an x-framework) into the generated code, somewhat similar in an include directive in other programming languages. See appendix for a description of XVCL commands.

points to the generic array x-frame. Setting the XVCL variable appropriately before the `<adapt>` command generates the original Java code for the two buffer classes.

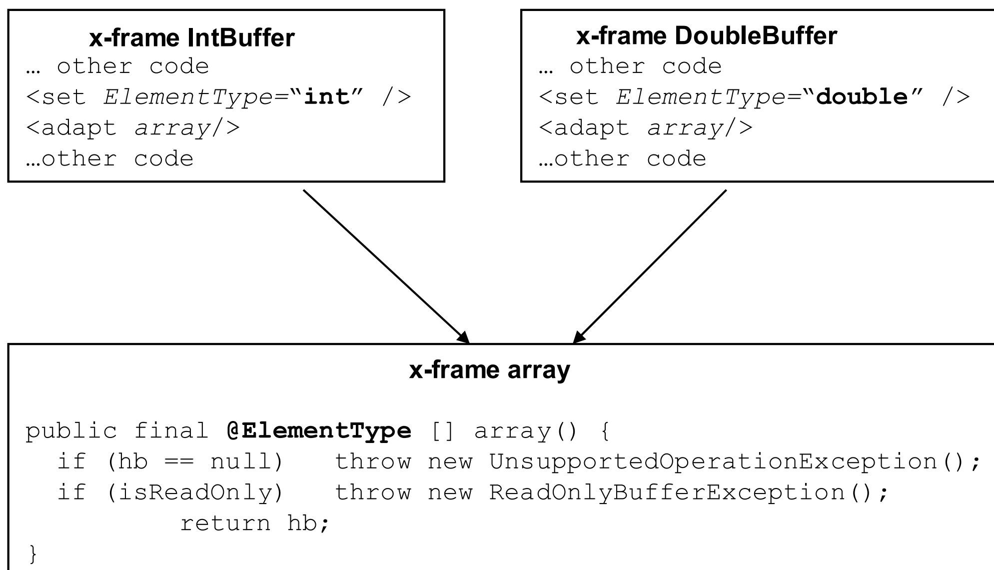


Figure 30. X-framework for unifying the array() clone

Running the x-framework in Figure 30 through the XVCL processor (XVCL processor is a preprocessor for XVCL code, freely available from the XVCL web site) generates the two Java classes containing the two methods given in Figure 29. This process is shown in Figure 31. The top level x-frame, typically called the SPC (to mean 'specification frame'), is an x-frame that acts as the handle to the x-framework.

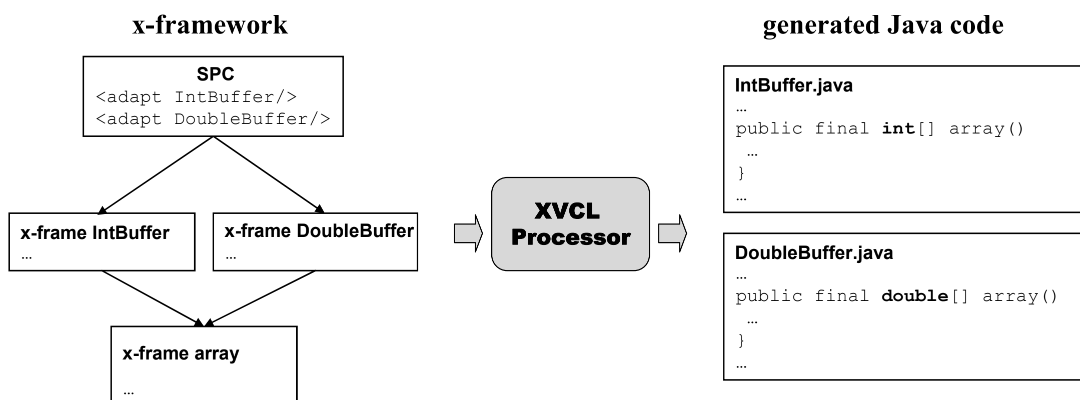


Figure 31. Generating two array() methods from the x-framework

XVCL is capable of much more complex variation handling than the parameterization illustrated in the preceding example. Appendix A gives a summary of essential XVCL features, and XVCL web site [XVCL] has more XVCL learning resources.

5.2. Overview of mixed-strategy

Mixed-strategy is the synergistic application of XVCL to complement the generics mechanisms in conventional implementation technologies. The name 'mixed-strategy' emphasizes that XVCL complements, rather than competes with conventional genericity mechanisms; XVCL is used only when it offers a clear advantage over alternative conventional clone unification techniques.

Jarzabek [XVCL] describes mixed-strategy as follows:

> While there may be many other ways to tackle the problem of similarities, mixed-strategy attempts to do so in domain-, language-, and platform-independent way, with only modest extensions to today's programming techniques. In a nutshell, mixed-strategy (1) represents each significant group of similar program structures (i.e., clones) in a unique, generic, but adaptable form, (2) delineates the differences among specific program structures in each group as deltas from their generic form, (3) records the exact location of each such structure in a program, and (4) automates derivation of specific structures from their generic forms to produce an executable program.

As shown in Figure 32, a mixed-strategy solution consists of co-evolving application code, and XVCL meta-code that correspond to the application code (or part thereof). The figure shows the conceptual view of how clones $\mathbf{A_1}$ and $\mathbf{A_2}$ are managed using the mixed-strategy. The generic form of the clone is **A** which is a part of the XVCL portion of the mixed-strategy application. It also specifies the deltas specific to instances $\mathbf{A_1}$ and $\mathbf{A_2}$ (internal structure of the

XVCL code is not shown in this diagram). Running the XVCL code through the XVCL processor generates the exact application code as before (i.e., before applying XVCL), having both instances of the clone. As per this scenario, we say that "$\mathbf{A_1}$ and $\mathbf{A_2}$ are unified to **A** at meta-program level, using XVCL".

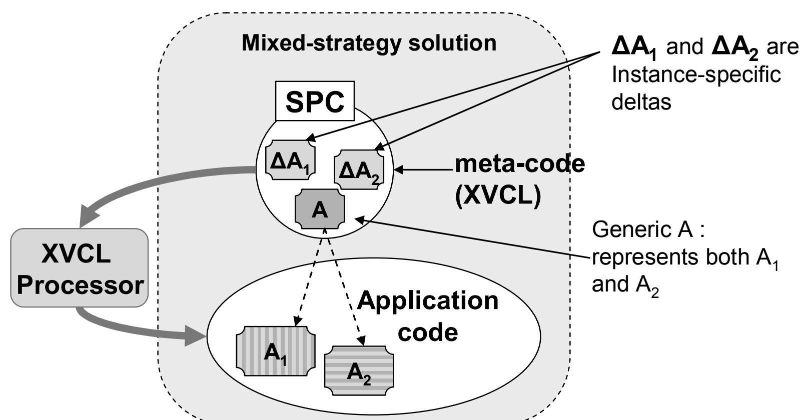


Figure 32. Clone unification in a mixed-strategy application

> A mixed-strategy solution consists of co-evolving application code, and XVCL meta-code that correspond to the application code

5.3. Benefits and drawbacks of mixed-strategy

Mixed-strategy is expected to positively affect the maintenance in the following ways:

- Modification points are reduced. Since modifications can be applied to the XVCL version of the clone and propagated automatically to all instances, this reduces the risk of update anomalies.
- Similarities and variations (i.e., commonalities and differences) are explicitly expressed using XVCL. This helps in finding the clones that needs to be modified, as well as in deciding whether variations call for the modification to be applied slightly differently.

- Reduces the code size that needs to be maintained. Since the XVCL code does not contain duplicate code, the entire system can be understood by examining the much smaller XVCL code.

Mixed-strategy does enhance certain aspects of maintainability, but it does not come for free, as the following caveat from the XVCL literature [XVCL] warns.

> As we relax the coupling between the parameterization mechanism and the rules (syntax and semantics) of the underlying programming language, the power of the parameterization mechanism increases. For example, with C++ templates we can unify a wider class of variations as compared to Java generics. At the end of this spectrum, there are techniques such as XVCL that manipulate a program as text, with no regard to language rules. By separating genericity issues from the core constructs typically supported by programming languages, we can address genericity concerns without compromising program runtime properties. But as we move towards less restrictive parameterization mechanisms, we also decrease type-safety of parameterized program solutions. Therefore, there are important trade-offs to consider. Designing generic, reusable and maintainable solutions is always a challenge which requires more talent and skill than building a concrete program. Mixed-strategy is not a substitute for thinking, on the contrary, it requires more thinking and up-front investment for future benefits. Mixed-strategy targets long-lived programs that undergo extensive evolutionary changes, or need be tailored to requirements of multiple customers.

5.4. Mixed-strategy success stories

Given next is some of the successes achieved using mixed-strategy in the past.

Buffer library case study [JL03]**:** In a case study of the Java Buffer library[27], mixed-strategy was able to reduce the size of the code by 68% by unifying the duplicated code.

STL case study [BRJ05a]**:** By applying mixed-strategy we were able to reduce certain parts of the SGI's C++ STL by up to 50% (described in section 4.2).

J2EE product line case study [YJ05]**:** Application of mixed-strategy on a J2EE product line enabled a reduction of 61% in the managed code.

ASP web portal product line [PJ05]**:** When used in an industrial ASP web portal product line having nine members, mixed-strategy reduced the overall managed code lines for nine portals to 22% less than the original single portal. New portal modules could be built by writing as little as 10% of the total code, while the rest of code could be reused from existing similar modules, resulting in an estimated eight-fold reduction of effort.

A few other mixed-strategy successes are described in [ZJ03a], [ZJ03b], and [ZJLR03].

5.5. Mixed-strategy and tenacious clones

Our past experiences with mixed-strategy (summarized in section 5.4) have shown that mixed-strategy can unify clones (at meta-program level) that are otherwise non-unifiable using conventional techniques. This is not surprising, given that XVCL works at one level above the program, free from restrictions of the implementation language/technique/paradigm. This freedom is further enhanced by the non-restrictive parameterization ability of XVCL.

[27] The same Buffer library was used in another case study (described in section 4.1)

Further, mixed-strategy does not remove clones from the application code. This suits well with situations where we deal with intentional clones. Therefore, mixed-strategy allows us to unify intentional clones (at meta-program level), while still keeping them in the application code.

5.6. Why choose mixed-strategy?

Given the extensive base of experience we have with mixed strategy, and our past successes with it, mixed strategy was a natural choice for us to use in this research. Pros and cons of mixed strategy, and how it compares to other competing techniques have been extensively discussed in previous published case studies [JL03][BRJ05a][YJ05][PJ05] [ZJ03a][ZJ03b][ZJLR03]. In this section we mention why we think mixed strategy is particularly suitalble for the area of web applications.

While applications written in several languages are certainly nothing new, multilingualism is taken to a new level in web application development [SCD03]. Web applications are implemented using a mixture of content types (ASP, C#, CSS, DTD, HTML, Java, JavaScript, etc.). In our survey of web applications (described in Chapter 3 and [RJ05b]), we found 59 content types in 17 web applications (we considered all text files that are likely to be maintained by hand); on average, one web application involved 10 different content types. While each content type may have its own clone unification facilities, some of these content types are special purpose scripting languages and markup languages. Clone unification mechanisms from such languages tend to be weaker than that of fully pledged programming languages such as C++.

Furthermore, some clones can involve multiple content types intertwined with each other. This further exacerbates the difficulties in unifying such clones. Since XVCL is independent

of the underlying language, mixed-strategy is ideal for situations where clones involve multiple content types.

Another factor that increases mixed-strategy's applicability to the web domain is that the meta-programming approach of XVCL is not entirely new to web developers. Server page techniques such as ASP/JSP/PHP indeed use embedded scripts of one language to generate code of another (usually HTML). The main difference is that server pages are a runtime technique (code generation happens on the fly, when the page is requested) whereas XVCL is a construction-time technique (code is pre-generated, before deployment).

One more factor in favor of mixed-stategy in the web domain is that XVCL is fully XML-based. XML and other markup languages are very familiar to web developers, and hence they are likely to find learning XVCL easier.

5.7. Chapter conclusions

In this chapter we described the fundamentals of the mixed-strategy, i.e., complementing conventional techniques with the powerful meta-programming technique of XVCL.

A number of previous studies have successfully used mixed-strategy to unify clones when other clone unification techniques failed. Therefore, we accept mixed-strategy is a viable solution to unify clones that are difficult to unify using conventional clone unification techniques, or those clones created intentionally.

Chapter 6.

Unification Trade-offs

Good, fast, cheap; choose any two

-Anonymous

This chapter describes a proof of concept experiment that illustrates how the mixed-strategy overcomes most of the unification trade-offs incurred by other clone unification techniques.

In some situations it is possible to unify clones using conventional techniques, but such elimination forces us to compromise other desirable qualities of the software development process (e.g., ability to rapidly evolve the software), and the final product itself (e.g., performance). We call these trade-offs 'unification trade-offs'. Unification trade-offs are typically incurred when we unify clones that provide some benefit.

We studied the unification trade-offs problem using an empirical study of alternative designs of the same web application (WA). We applied conventional design techniques (such as design patterns) to build generic solutions into the web application, progressively achieving a generic, clone-free system. Overall, we were able to unify most of the clones, resulting in a significant reduction of the code size, and a lesser risk of update anomalies. Yet, throughout the experiment we identified a number of unification trade-offs. Further, we found that mixed-strategy too could achieve similar levels of clone unification. More importantly, we could show that the mixed-strategy approach avoided the trade-offs we observed when using conventional techniques.

The organization of this chapter is as follows:

Section 6.1 we describes our experiment in detail. We start by describing the WA we developed, the details of the experiment method, followed by explanations of the four alternative implementations. Finally we give an overall comparison of the four implementations and how the experiment would have differed if it was done using other implementation platforms.

Section 6.2 analyzes the trade-offs we observed. For each trade-off we discuss the WA engineering realities that set the context for the trade-off, concrete examples from the experiment to illustrate how clone unification forces the trade-off, how mixed-strategy can avoid the trade-off, and the applicability of the trade-off to other platforms.

Section 6.3 discusses our observations during the experiment, specifically, the reasons behind mixed-strategy's ability to avoid trade-offs.

Our contributions contained in this chapter are:

Contribution 4. It illustrates and analyzes the trade-offs in applying conventional clone unification mechanisms to unify clones in the web application domain. It shows how mixed-strategy avoids most such unification trade-offs.

6.1. Case study: Project Collaboration Environment

We illustrate the problems created by unification trade-offs, using an empirical study of alternative designs of the same WA. We used a WA called Project Collaboration Environment (PCE) as the basis of this study. We built PCE based on requirements from one of our industry projects [PJ05], so that PCE represents a realistic WA in terms of functionality. We used the Server page technique of PHP in the study. During domain modeling and similarity analysis, we identified similarity patterns in PCE that had a potential to create clones. Then, we applied conventional design techniques (such as design patterns) and features of PHP to build such solutions into PCE, progressively achieving a generic, clone-free WA. We did

three consecutive implementations of PCE, where each implementation was a refinement of the previous one. Overall, we were able to unify most of the clones as we moved from the first implementation to the third. This resulted in a significant reduction of the code size (by 78%), and a lesser risk of update anomalies (number of modification points dropped from 251 to 8 for certain changes).

Yet, throughout the experiment we identified a number of trade-offs incurred by our efforts to enhance genericity of PCE. While some of these trade-offs were well known and applicable to traditional software as well, majority were less obvious, and sometimes specific to WAs (or more applicable to WAs). These trade-offs resulted from the interplay between clone unification measures, realities of WA development (such as fuzzy requirements, dramatically short development schedules, constant evolution, and shortened revision cycles[DMG+02]), and desirable engineering qualities of WAs (such as high performance, high information content, and good aesthetics). We believe some of the compromises that had to be made when unifying clones would be unacceptable in many WA development situations. Further analysis hinted that our observations could be equally valid for more comprehensive WA building platforms like .NET and JEE (formerly known as J2EE).

Then we compared the PHP solution to a solution in which we used the mixed-strategy to unify clones. Our experiment found that mixed-strategy too could achieve similar levels of code reduction at meta-program level. More importantly, we could show that the mixed-strategy approach was able to avoid the trade-offs we observed when using PHP.

6.1.1. Project Collaboration Environment (PCE)

Project Collaboration Environment (PCE), used as an example here, is a WA created based on requirements from our industry partner, ST Electronics [PJ05]. It supports project record keeping, task assignment to staff, task progress tracing, and a range of other activities related to project planning and execution.

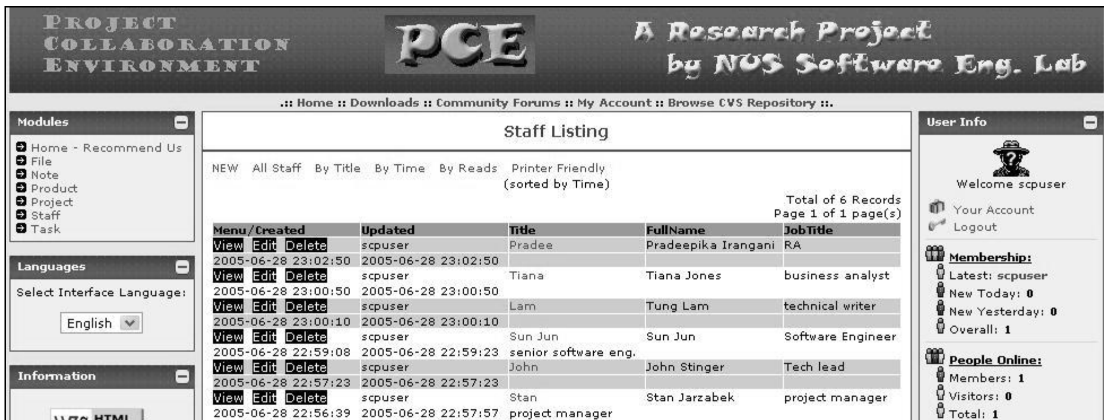


Figure 33. A screenshot from the Staff module

PCE has six modules corresponding to six entity types in PCE domain, namely **Staff**, **Project**, **Product**, **Task**, **Notes**, and **File**. PCE maintains records of those entities and relationships among them. For example, **Staff** module maintains records of staff members, **Product** module tracks the status of project deliverables, **Staff** and **Project** modules maintain info about which staff members belong to which project teams, and **Task** and **Staff** modules maintain info about project tasks assigned to staff members. A screenshot from the **Staff** module given in Figure 33 shows a listing of staff members. Figure 34 depicts main PCE entity types and relationships among them.

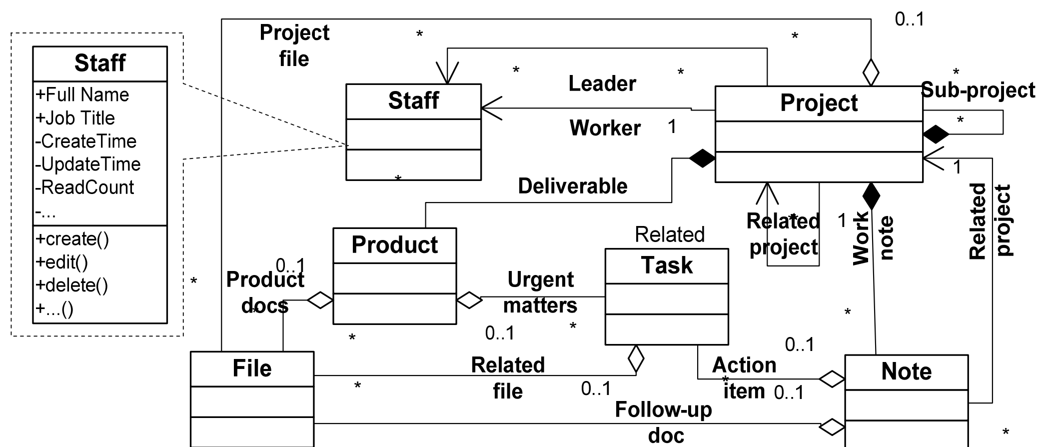


Figure 34. Domain model of PCE

Figure 35 depicts similarities and variations between PCE modules as a feature diagram [Kan90]. A typical module M in PCE has a name (e.g., `Staff`), and a number of attributes (e.g., `Staff` module has attributes `Full Name`, `Job Title`, … *cf* Figure 35). Some

attributes are common to all modules, while others - optional - are module-specific. Each module supports actions (`create`, `edit`, `delete`, …). Some actions are further divided into sub-actions, some of which are optional. A module may optionally have relationships (association, and either 'strong' composition or 'weak' composition - but not both).

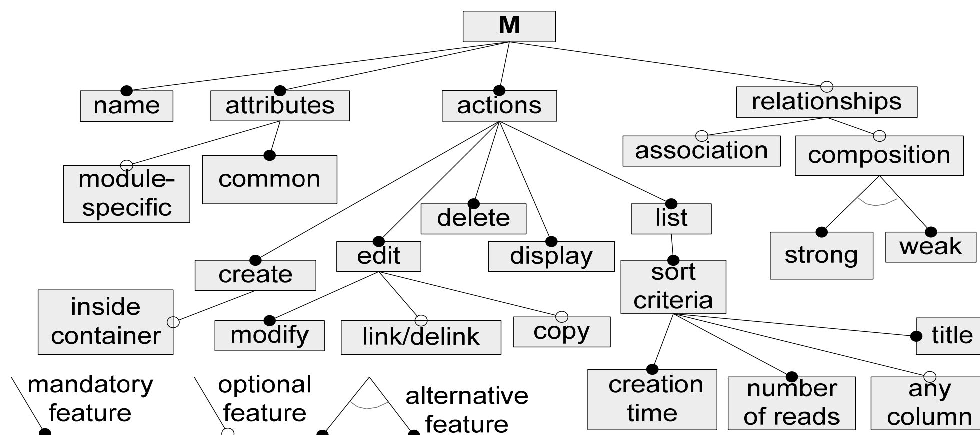


Figure 35. Feature diagram of a PCE module

Functionality of a given PCE module is a combination of features given in the feature diagram. The high proportion of mandatory features implies that modules are highly similar to each other, creating a possibility of cloning. However, optional features and alternative features inject some variations between the modules. Our feature diagram only depicts high level, inter-module variations. There are also lower level variations among modules. For example, create action of the `File` module carries extra functionality to upload a file. And at a finer granularity, there are intra-module similarities. For example, `copy` action and `edit` action are very similar, as both involve retrieving an existing record, editing it, and storing it in the database (overwrite in the case of `edit`, save as a new record in the case of `copy`). These intra-module similarities were also analyzed and duly noted.

6.1.2. Experimental method

We used Server pages as the implementation technique, and selected PHP to implement Server pages. PHP is a free, popular (22 million web domains used PHP by September

2005)[28], and versatile scripting language specifically geared for WA development. Although PHP started out as a simple scripting language, today it has evolved into an industrial strength WA technology, used by complex WAs such as sourceforge.net.

For the Foundation and General User Modules, we reused CPG-Nuke code as is whenever possible, and with minimal changes when necessary. The six PCE modules were deployed as another set of User modules. The reuse of CPG-Nuke reduced the PCE implementation to just these six modules. We built them in conformance with the Foundation requirements, so that they too could use Foundation services, and could be managed using the Foundation (e.g., we used the Foundation services for implementing a common look and feel). With the reuse of CPG-Nuke we hoped not only to reduce the implementation workload, but also to ensure that our implementation was based on an industry-accepted architecture.

Rather than building the whole PCE from scratch, we reused CPG-Nuke[29] as the basis of PCE implementation. CPG-Nuke is an adaptation of PHP-Nuke[30], a popular open source WA, averaging ½ million downloads per year for last three years. Figure 36 shows the high level architecture of PCE. The Foundation consists of Admin Modules (used for administration of PCE) and Service Modules (used to provide various infrastructure services like database connectivity, logging, etc.). Foundation acts as a platform on which we deploy various User Modules. It provides a framework for implementing modules, and administration facilities to manage those modules. General User Modules provide common facilities to users (e.g., polls, message boards, preference management, etc.).

---

[28] PHP usage statistics, http://www.php.net/usage.php

[29] CPG-Nuke home, http://www.cpgnuke.com/

[30] PHP-Nuke home page, http://phpnuke.org

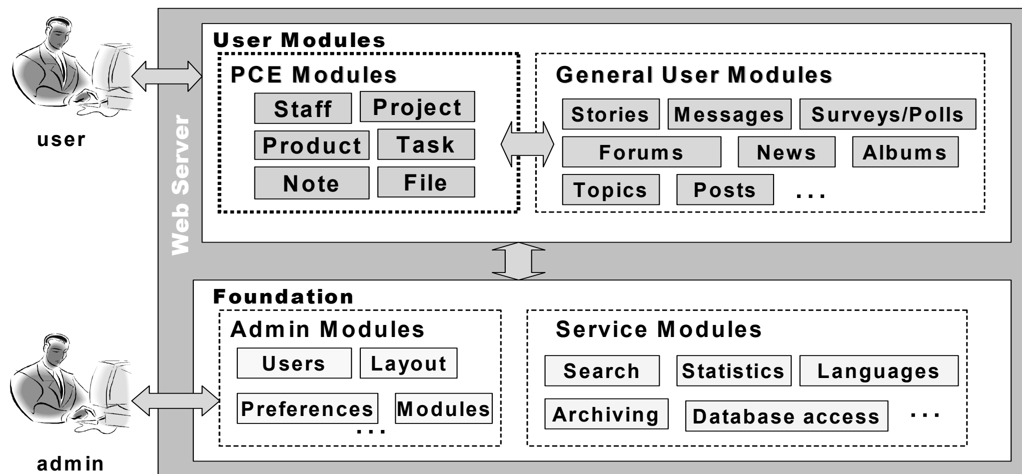


Figure 36. High level architecture of PCE

We carried out four different implementations of PCE (see Figure 37), each one functionally equivalent to the others. The first implementation was based on a very simple design, without much effort to avoid clones. We call this PCEsimple. The second implementation, PCEpatterns, tried to avoid clones by applying suitable design patterns to PCEsimple design. The third implementation, called PCEunified, built upon the design of PCEpatterns, was focused on unifying the remaining clones. In the fourth implementation, we applied mixed-strategy on PCEpatterns as a direct alternative to PCEunified.

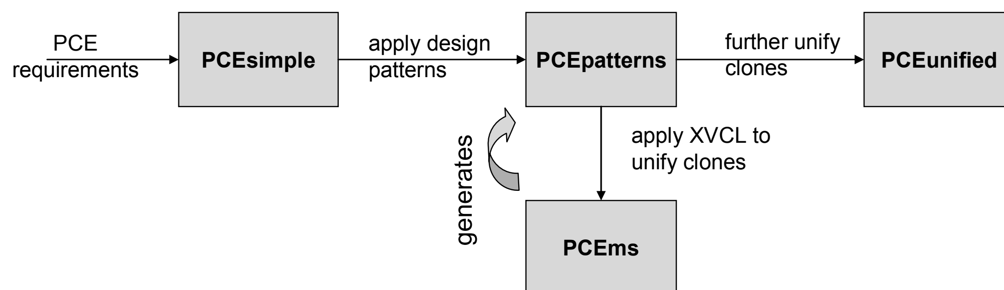


Figure 37. The four PCE implementations

All four implementations were done by the author who is also a trained software engineer having industry experience and technical certifications.

6.1.3. PCEsimple

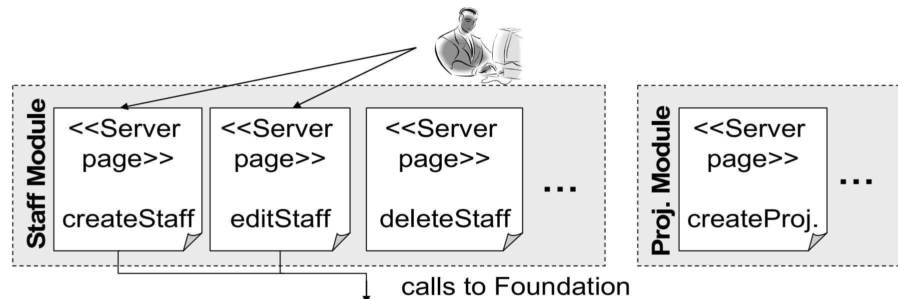


Figure 38. Design of PCEsimple

We followed the so-called KISS principle (i.e., Keep It Simple, Straight-forward) when implementing PCEsimple. This initial version of PCE exemplifies a first-cut solution that is likely to emerge when developing a new WA under time pressure. The priority was to "get PCE done", with maintainability concerns taking a backseat.

Each action (or sub-action) of the module in PCEsimple was implemented as a single independent Server page (see Figure 38). For example, `createStaff.php` page implemented the create action for `Staff` module. Cloning was liberally used when dealing with intra/inter-module similarities. For example, we implemented one module and used it to implement other modules by simply copying it and modifying it. Two forces heavily influenced the design of PCEsimple:

1. Architectural guidelines implied by the Foundation - Although our design was the simplest possible, we still adhered to the guidelines implied by the Foundation.

2. Conceptual design of a similar WA implemented by our industry partner [PJ05] (source code was not available as it was a commercial application) - PCE conceptual model (Figure 33), direct mapping of modules to entities, and the page-per-action organization in PCEsimple were results of following this design.

With the above two, we expected our PCE to closely match an industrial implementation.

Figure 39 visually represents some intra/inter-module clones in PCEsimple (Note that this only a visual representation of relative proportion/location of similarities and variations, created based on the actual clones; Text contained in clones is not meant to be read):

[a] `createFile.php` is an inter-module clone of `createProject.php` (white regions represents duplicated text, variations from `createProject.php` to `createFile.php` are marked as dark regions)

[b] An identical intra-module clone caused by similar preprocessing done for each action

[c] Intra-module clone caused by similarity between the create action and the edit action (mentioned in section 6.1.1).

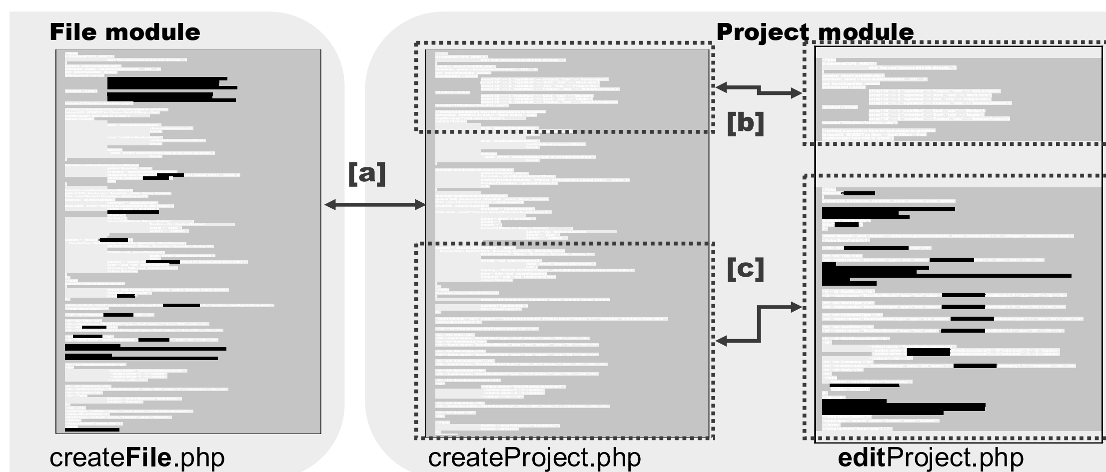


Figure 39. Some clones in PCEsimple

6.1.4. PCEpatterns

The objective of PCEpatterns was to make the UI more generic, using patterns that reduce cloning. First, we identified major cloning situations within a PCE module using the CCFinder clone detection tool [KKI02] and Gemini clone visualization tool [UHK+02]. Then we selected the appropriate patterns to unify the clones by comparing the clone scenarios against patterns used in industry best practices. It should be noted that application of patterns is not an exact science, and very much dependent of the experience of the developer. It is

difficult to automate this process or define a rigorous procedure for this purpose. However, we drew upon JEE recommendations [ACM03], .NET recommendations [Tro03], and platform-independent recommendations [Fow03]) when selecting the pattern to apply. Some examples of clone situations we found and the matching patterns we chose are given below (the rest is omitted for brevity).

- Similar preprocessing sequences were repeated for each page request (e.g., session validation, parameter decoding). We applied the *Front Controller* [ACM03] [Tro03] [Fow03] pattern to unify this clone into a single location.
- Fragments of UI recurred in multiple places (e.g., attribute display code was cloned in `edit page` as well as in `display page`). We applied the *Composite View* [ACM03] pattern to unify this clone.
- Data retrieval code was cloned in multiple views. We used the View Helper [ACM03] pattern to unify the cloned code into a common helper class.

We applied these patterns within the scope of a module, repeatedly applying the same patterns to each module. The rationale for this was to keep the modules independent from each other so that each one can evolve independently if so required. The resulting meta-model of an Entity is shown in Figure 40, while Figure 41 shows a module designed by following this meta-model. Our pattern-based design was a composite of a number of design patterns, organized around the well known *Model-View-Controller* (MVC) pattern. At an abstract level, each entity consisted of a Model, a number of Views, and a number of Controllers that updated the model and selected the appropriate View to visualize the Model (see Figure 40). We now describe the organization of other design patterns around MVC.

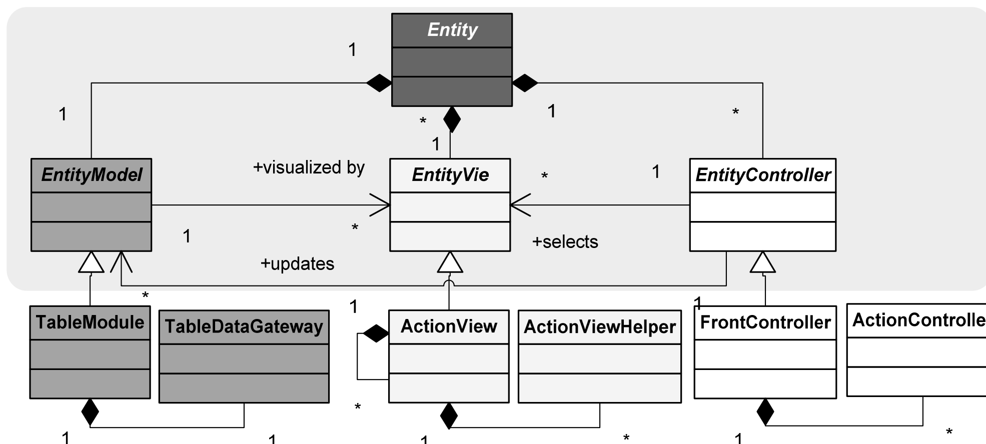


Figure 40. Meta-model of a module in PCEpatterns

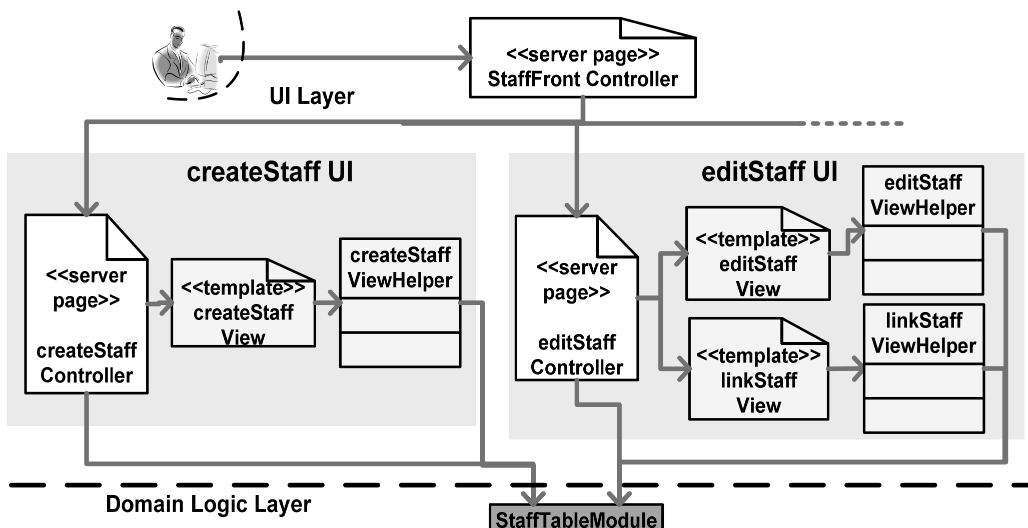


Figure 41. Design of Staff module in PCEpatterns

Model: Model spans the domain logic layer and data source layer. Since our main focus was on the UI, we tried to keep the other two layers simple; each module has minimal domain logic and is stored in a single table in the database. As recommended by [Fow03] for such situations, we used the *Table Module* pattern and the *Table Data Gateway* pattern for the Model portion (not shown in Figure 41).

View: Views in PCE are organized around actions a module supports. We used the *Template View* [Fow03] pattern, Composite View [ACM03][Tro03] pattern, and *View Helper* pattern [ACM03] for the View portion. Following the Template View pattern, views are constructed as templates (`ActionView` in Figure 40, `createStaffView` in Figure 41). Following the

View Helper pattern, each View may be aided by a helper class (`ActionViewHelper` in Figure 40, `createStaffViewHelper` in Figure 41). Following the Composite View pattern, these views may be formed by smaller Views (not shown in diagrams).

Controller: We used a combination of the Front Controller [ACM03][Fow03][Tro03] pattern and the *Page Controller* [Fow03][Tro03] pattern for the Controller portion. Following the Front Controller pattern, each module has one Front Controller to receive all requests from the user and to perform control tasks common to all requests (`FrontController` in Figure 40, `StaffFrontController` in Figure 41). Following the Page Controller pattern, each Front Controller uses a number of Page Controllers, one for each action (`ActionController` in Figure 40, `createStaffController` in Figure 41), to perform action-specific control tasks.

6.1.5. PCEunified

This implementation was an all-out effort to unify any remaining clones, by tweaking the design and using PHP features. We applied the following steps:

1. Identify remaining intra-module clones in PCEpatterns - We used CCFinder [KKI02] and Gemini [UHK+02] in this step.

2. Unify identified clones - We used a combination of the following techniques:

- Extract duplicated code fragments into functions
- Unify similar functions by adding extra parameters and conditional branches to handle, and using Template Method pattern [GHJ97]
- Use PHP scripts to handle variations in HTML clones
- Further, and more intensive, application of Composite View pattern to unify common parts of Views

3. Unify all six modules into one generic module - this was the major unification done in this phase. We pulled Front Controller out of the module and broke it into two layers of Controllers (see Figure 42). The top layer consisted of a common Front Controller for handling common control tasks. The second layer consisted of six module-specific controllers (e.g., `StaffFrontController`, `ProjectFrontController`, … in Figure 42). At the implementation level, we used the same techniques (given in the previous paragraph) to handle inter-module variations.

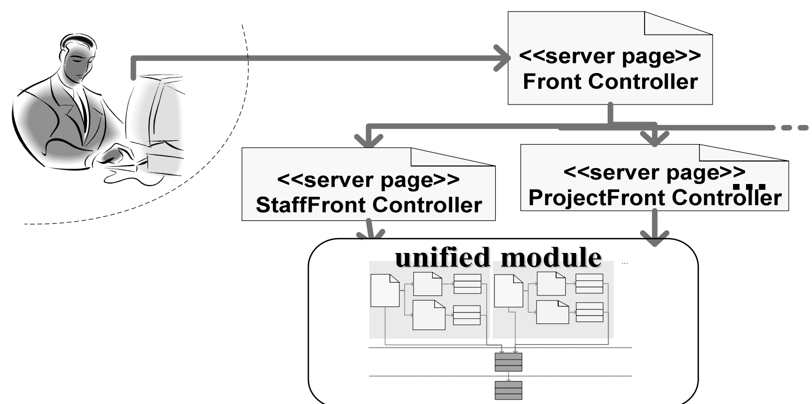


Figure 42. Design of PCEunified

6.1.6. PCEms

As described in Chapter 5, mixed-strategy suggests using the optimum mix between conventional techniques and XVCL. To simulate a mixed-strategy solution, we first accept PCEpatterns as the optimum level of clone unification we can achieve with conventional design techniques without compromising other desirable qualities. Then, we created PCEms by using XVCL to unify (at meta-program level) any remaining clones in PCEpatterns. Creating PCEms could be done as a downstream activity, typically after PCEpatterns is deployed and stable.

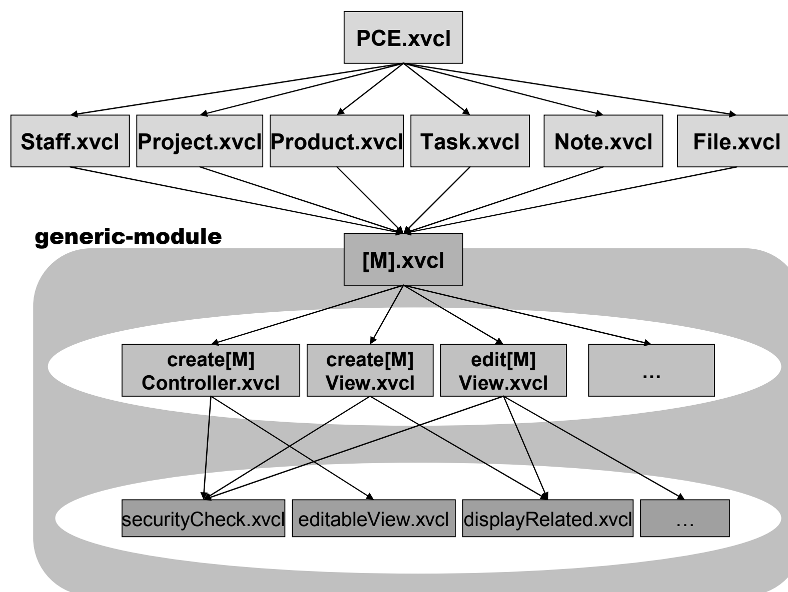


Figure 43. X-framework for PCEms

Figure 43 illustrate a simplified version of the PCEms. Since structuring of XVCL solutions is described in more detail in coming sections, we do not go into a detailed description of PCEms x-framework here. In brief, the x-frame `[M].xvcl` and the x-framework under it represent a generic PCE module. X-frames at the level of `Staff.xvcl`, `Project.xvcl`, and so on represent module-specific variations that need to be injected to the generic module when generating respective modules in PHP. The `PCE.xvcl` is the root x-frame that represents the whole system. Arrows in the figure represents `<adapt>` commands.

Our decision to consider PCEpatterns as the optimum conventional solution is a subjective one, used only as an illustration. Other scenarios are possible, for instance, we could have used PCEsimple as the optimum conventional solution and applied mixed-strategy on top of it.

6.1.7. Overall comparison

We start by comparing the size and the cloning level in these four implementations of PCE. To measure the cloning level, we use the percentage of non-unique (i.e., cloned) code, calculated based on clones detected by CCFinder tool [KKI02]. This measure is directly

related to the risk of update anomalies. For instance, if 35% of the system is non-unique, any changes to that 35% of the system carry a risk of an update anomaly. To avoid distortions created by false-positives and trivially short clones, only exact duplicates that are longer than 20 tokens were counted as clones. The details of this calculation can be found in section 3.1 (given as *TCTp*).

Table 4 summarizes the cloning percentage (C%), LOC count, and number of files (#F), calculated for a typical intra-module (we chose `Project` module as the typical module, since it was used as the blueprint for creating other modules), and for all modules (i.e. within and across all modules). The last column shows the inter-module cloning level (we chose `Project` module and `Product` module to calculate this metric). Let us consider the three conventional solutions first. Table 4 indicates a very high (98%) overall cloning level in PCEsimple, i.e., almost all code in PCEsimple is repeated in more than one place. This is because we copied existing modules to create new modules, resulting in many inter-module clones. This number is also comparable with findings of our industry case study [PJ05], which reported that up to 90% of a new module may be implemented by reusing code from existing modules.

Table 4. Size and cloning level comparison

	intra-module			**all modules**			**Inter-mod C%**
	C%	**LOC**	**#F**	**C%**	**LOC**	**#F**	
PCEsimple	55	1085	10	98	5244	55	80
PCEpatterns	32	931	21	86	5095	120	74
PCEunified	15	838	20	26	1128	32	-
PCEms	17	864	24	15	1063	34	-

We also see that there is a noticeable drop in intra-module cloning from PCEsimple to PCEpatterns (from 55% to 15%). This shows that application of patterns has helped to avoid some clones. However, the repeated application of same patterns for each module has created many inter-module clones, maintaining the overall cloning levels still high (*cf* last column of

Table 4). Further unification of intra-module clones, followed by unification of modules has reduced both intra-module and overall cloning levels in PCEunified. A manual examination revealed that the remaining clones in PCEunified are either too small to warrant unification, or not practical to unify (section 6.2.3 gives an example).

Cloning level of PCEms is close to PCEunfied than others, showing that clone unification level achieved by the mixed-strategy is comparable to that achievable by pushing conventional techniques to the limit.

Comparison of maintainability

Firstly, there is a significant drop in the size of code to be maintained from PCEsimple to PCEunified. There is a 33% reduction in code size (in terms of LOC) within a module, from PCEsimple to PCEunified. The overall system size has dropped much more (by 78%) largely due to unification of six modules into one.

Secondly, the risk of update anomalies has reduced. Table 5 shows the distribution of the impact of the following three hypothetical evolutionary changes, when carried out for one module, or for all modules.

Change 1. Hyperlink all attribute names to a Glossary page.

Change 2. Show the 'last edited time' in a different location.

Change 3. Record each request to PCE in a log file.

Table 5. Change propagation comparison

		PCEsimple		PCEpatterns		PCEunified		PCEms	
	#M	#F	#L	#F	#L	#F	#L	#F	#L
Change 1	one	7	49	5	35	2	8	2	8
	all	37	251	29	195	2	8	2	8
Change 2	one	3	6	1	2	1	2	1	2
	all	18	36	6	12	1	2	1	2
Change 3	one	9	9	1	1	1	1	1	1
	all	55	55	6	1	1	1	1	1

It is clear from Table 5 how the number of modified files (#F) and modified locations (#L) decreases from PCEsimple to PCEunified, reducing the risk of an inconsistency during the update. When comparing PCEunified to PCEms, Table 5 shows that the effort required for doing changes 1, 2, and 3 remains the same between PCEunified and PCEms.

Therefore, based solely on the code size and the typical number of modification locations, it may appear that:

(1) The general maintainability has improved from PCEsimple to PCEunified

(2) PCEunified and PCEms show similar maintainability.

However, such a conclusion may be premature, as code size and number of modifications are only *some* of the indicators of maintainability. We shall revisit the maintainability of PCEunified and PCEms in section 6.2.2.

6.1.8. PCE on other platforms

Two other popular platforms for implementing WAs are JEE and .NET. They provide rich sets of general infrastructure services in WA development (e.g., for managing security, transactions, resources). In our PHP solution, the Foundation provides similar service, but in addition, it also provides more application-specific infrastructure services. Therefore, PCE implemented on the .NET or JEE follows the same high level architecture shown in Figure 36, possibly with a thinner Foundation (since some services are provided by the platform itself). As the design patterns we applied are also applicable to both platforms, the cloning properties of the three PCE designs should remain unchanged across PHP, .NET and JEE platforms. Also, the basic role of Server pages remains the same whether we use PHP, ASP.NET or JSP on JEE. However, .NET and JEE offer additional technologies fine-tuned for specific UI implementation tasks. Such UI technologies in JEE include Servlets (suitable

as Controllers of MVC), Java Beans (suitable as View Helpers), and Java Server Faces (a set of controls for WA UIs). Java Standard Tag Library (JSTL), the Template tag library, Custom tags, and Expression Language all help in implementing the Template view pattern and Composite View pattern. Similarly, .NETs' Code-Behind feature provides clean separation of HTML from page generation logic, while Server Controls aid in rapid constructing of WA UIs. ASP.NET also has built in support for Page controller pattern. ASP.NET Webparts and Java Portlet API provide building blocks for content aggregation in Portal type WAs. When we analyze the trade-offs in the next section we also comment on how each trade-off may be applicable to JEE and .NET platforms.

6.2. Trade-off analysis

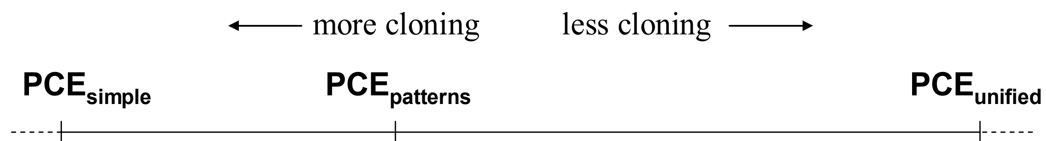


Figure 44. Cloning level in three PCEs

In order to generalize our observations, we place the three conventional PCEs along an axis denoting the cloning level (as illustrated in Figure 44). It shows how cloning level decreases as we go from PCEsimple to PCEunified, and possibly beyond. However, there are many other ways to design PCE, and the implementation of PCE in a given real production environment could land anywhere in this axis. From a purely clone reduction perspective, we would like to push it towards the right.

In our experiment, we observed how clone unification can lead to trade-offs in other WA properties that often should not be compromised. Such trade-offs can push the final result towards the left. We also observed that most of these trade-offs can be avoided using the mixed-strategy. Next, we present in detail each of the trade-offs we observed, and how mixed-strategy could avoid those.

6.2.1. Performance

Some WAs operate in the highly competitive environment of the Internet. As slower performance may drive away users/customers, "criticality of performance" is one important characteristic for such WAs [DMG+02]. Unfortunately, clone unification can affect performance negatively by introducing additional function calls, function parameters, and 'include' directives. As an example, a simple comparison of page generation time for five home pages of `Staff` module is shown in Figure 45 (all other things being equal, averaged over 10 page requests). In all cases, page generation times of the three PCEs follow the pattern: PCEsimple <<< PCEpatterns < PCEunified. On average (see last column), PCEunified is more than three times slower than PCEsimple. This example shows how clone unification, although feasible, can force performance trade-offs. In cases where common code is stored in a database, the additional database accesses are likely to degrade performance even more significantly.

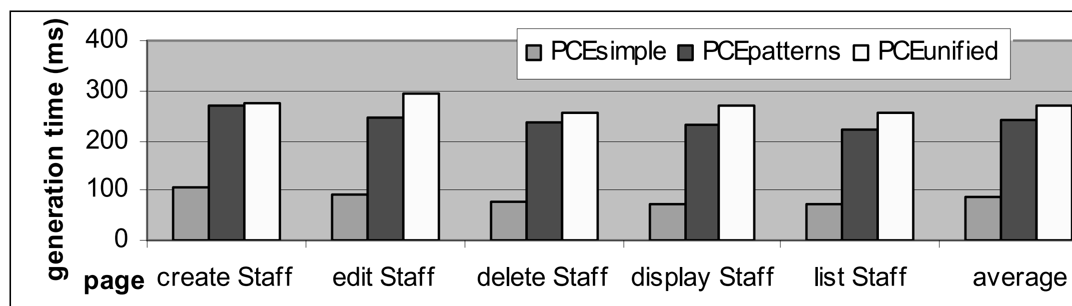


Figure 45. Page generation time comparison

Since mixed-strategy works at the meta-program level and not at the source code level, it does not affect the performance at all. In our case, the performance of PCEms would be the same as PCEpatterns, since the source code deployed is identical to PCEpatterns. If we wanted even better performance, we could have applied mixed-strategy on PCEsimple. Thus, mixed-strategy can entirely avoid the performance trade-offs involved in clone unification.

Unlike PHP which is interpreted at runtime, both ASP.NET and JSP use compiled code at runtime. There are many other performance boosters in JEE and .NET (e.g., translation time

inclusion, sophisticated caching mechanisms). However, the general cause of the performance loss is the increase of indirection introduced by clone unification from PCEsimple to PCEunified. Such extra indirection is inevitable during clone unification, and likely to degrade performance, on any platform. A simulation of the PCE clone unification scenarios in .NET and JEE confirmed this hypothesis, showing that our observations on performance trade-offs are equally valid for these two platforms.

6.2.2. Rapid prototyping/evolution capabilities

Although clone unification is typically considered a maintenance activity, it can also be done during initial implementation stages. Such proactive preventive clone management however can interfere with the agility of the development process. This is particularly true for WAs since they have, "compressed development schedules" [DMG+02] (typically, less than 3 months [MW01]) and "insufficient requirement specification" [DMG+02], encouraging many cycles of rapid iterative development. These conditions call for a simple design that could be rapidly implemented. One frequently quoted reason for cloning is that it is a quick and simple technique [DFST04]. A comparison of the three PCE designs supports this argument; there are more concepts, more indirection, and more layers as we go from PCEsimple to PCEunified. This increases the complexity, requires more planning, more analysis, and more modeling. Therefore, despite the drop in LOC, the upfront development effort and time-to-market increases as we go from PCEsimple to PCEunified. Although PCEunified is the smallest of the three, such a high degree of clone reduction is unlikely to be achieved in the first attempt. Rather, it would require an iterative approach such as we used. This shows that clone unification in this manner forces a trade-off in the ability to rapidly build the WA.

Clone unification similarly affects the ability to quickly evolve the WA. This line of argument may appear to contradict section 6.1.7, in which we illustrated how the number of modified files/locations decreases as we unify more clones (*cf* Table 5). The explanation is that while

clone unification eases modifications involving multiple clones (such as those in section 6.1.7), the additional complexity makes localized changes more difficult. This is precisely the kind of trade-offs we are trying to highlight here. As an illustration of this point let us consider the effort required to add a new localized feature to the three PCEs. Table 6 shows some data for adding the 'strong composition' feature to a module (assuming it only supported 'weak composition' to begin with). As per these data, the number of files involved (files that may be affected by this new feature), number of files actually modified, number of independent locations modified, and the number of LOC modified tend to increase as we go from PCEsimple to PCEunified. Functionality of all six modules needs to be tested in PCEunified, even though the change affects only one module. This could be a major burden, given the immaturity of WA testing techniques.

Table 6. Effort for adding 'strong composition'

	files involved	**files modified**	**locations modified**	**LOC modified**	**modules to test**
PCEsimple	1	1	3	n	1
PCEpatterns	7	3	4	~ n	1
PCEunified	9	4	4	> n	all

In contrast, mixed-strategy can be used in such a way that ability to quickly develop and evolve the WA is retained and enhanced. First, application of mixed-strategy can be delayed as much as necessary. This means during initial periods where the code is volatile, developers can concentrate on quickly building the WA without worrying about mixed-strategy, or clones. Cloning can be used during this stage if it speeds up the development process, since these clones can be tackled later using mixed-strategy. This way, mixed-strategy can in fact improve the speed of development rather than hinder it. Second, evolution of a mixed-strategy solution need not be done through the XVCL code alone. If there is a need to change something urgently in the WA, one can always work on the generated code until the change is stable and released. Migrating of the changes to the XVCL version can be done later, after the

urgency is no longer in effect. This way, mixed-strategy can be used to tackle clones without compromising the ability to rapidly develop/evolve WAs. However, the success of this approach depends on tool support in the area of back-propagating the changes from generated code to XVCL code.

This trade-off is applicable independent of the platform or the application domain. For example, both JEE and .NET have mechanisms to tackle similarities in UI controls (e.g., by developing custom controls, custom tags), but these mechanisms require much upfront work, forcing trade-offs in the rapid prototyping/evolution capabilities of the platform.

6.2.3. Framework conformance

It is typical to build WAs by using available platforms/frameworks, rather than build from scratch. However, each such platform/framework has conformance requirements. For instance, some of them require certain code/file to be physically present in a given location. We encountered two such examples in our experiment:

1. PCE Foundation required a certain security check to be placed at the beginning of each file, to prevent direct access to it.
2. PCE Foundation required each module to be in a separate folder (bearing the same name as the module), and a file named `index.php` to be present in each such folder.

Clone unification can interfere with such framework requirements. In the first example, we could not unify the security check; it remained cloned in every file in PCEunified. In the second example, we had to modify the Foundation (generally a risky, and an undesirable option) to remove that requirement.

Similar to the performance trade-off, this trade-off can be avoided entirely by using mixed-strategy to tackle clones, since the applications source code is not affected by mixed-strategy. For example, both above cases were easily unified in PCEms.

6.2.4. Tidiness in source distribution

Often, the Server page portion of a WA is delivered in source form. In such a case it is desirable to eliminate all the unused parts from the delivered code. This may be due to space/time efficiency concerns. For instance, due to severe space constrains on the server or to avoid transfer of unused client-side scripts over the network. Or this may be to minimize impact of modifications. Most WAs are accessed globally, and need to be available 24/7. Downtime caused by updates to the unused code is unacceptable for such WAs. Unfortunately, clone unification sometimes injects unused code. For example, in PCEunified, Staff module uses only 77% of the unified module. If the unified module is reused in another WA to serve as a Staff module, it results in carrying over 23% of the code that will not be used at all. Therefore clone unification can sometimes force delivery of unused code.

This is another trade-off that can be avoided entirely by using mixed-strategy, since the mixed-strategy solution can generate customized code for each use containing the minimal code required, avoiding the overhead of library code that is never used.

Both JEE and .NET support ‘hot updating’ a running WA (hot updating is the updating of the application without taking it offline). This can mitigate the problem of downtime caused by updates to unused code. However, hot updating is not recommended for production environments.

6.2.5. Indexing by search engines

Success of some WAs depends on how easy it is for search engines to index them. Since dynamic contents are less likely to be indexed by search engines, it is unacceptable for such WAs to unify static clones using Server pages. A good example is an e-commerce application preferring not to unify cloned static pages in its product catalog. This trade-off is applicable to any WA, whenever indexing by search engines is critical to its success.

A mixed-strategy solution is ideal to avoid this trade-off, since it can unify static HTML clones at meta-level without introducing program logic that hinder indexing.

This trade-off is applicable no matter which platform we use to implement the WA.

Note: This trade-off (section 6.2.5) is not directly related to PCE experiment. It was highlighted by one of our industry collaborators.

6.2.6. WYSIWYG editing

Three important characteristics of a WA are "aesthetics", "information content", and "constant evolution" [DMG+02]. Therefore, the creation and maintenance of WA UI require continuous involvement of multimedia authors (e.g., graphic designers), content authors (e.g., technical writers), and programmers. The first two categories typically prefer to work with WYSIWYG authoring tools. Overzealous clone unification however, can interfere with such WYSIWYG editing. For example, the PCE UI was constructed as an HTML based template, and the program logic was placed in helper classes. Typically, a graphic designer creates the UI template using a WYSIWYG editor (e.g., Macromedia Dream Weaver), while a programmer builds helper classes. 'Hooks' (very short PHP scripts) in the template extract the dynamic parts from the helper class. Except during the time programmer places hooks in the template, both experts work in parallel. Figure 46 illustrates this situation.

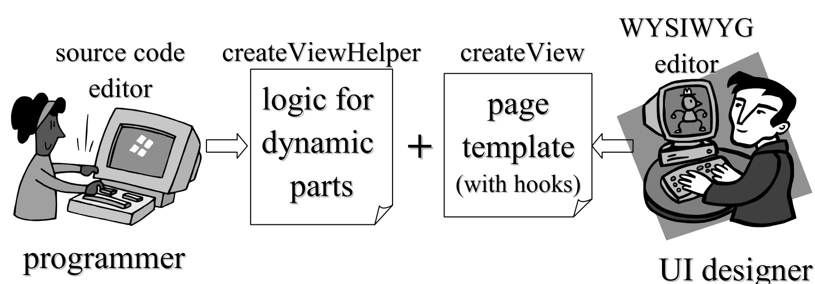


Figure 46. Parallel editing of dynamic pages

We observed that the intensive clone unification in PCEunified had a negative impact on this setup. It brought more programming logic into the template (in the form of extra parameters,

conditional branches, function calls), fragmented the template (e.g., when using Composite View pattern), and made the rendering of WYSIWYG editor increasingly different from the actual result. An example is shown in Figure 47 where (a) depicts how the page is displayed in the browser while (b) depicts how it is shown in a WYSIWYG editor[31].

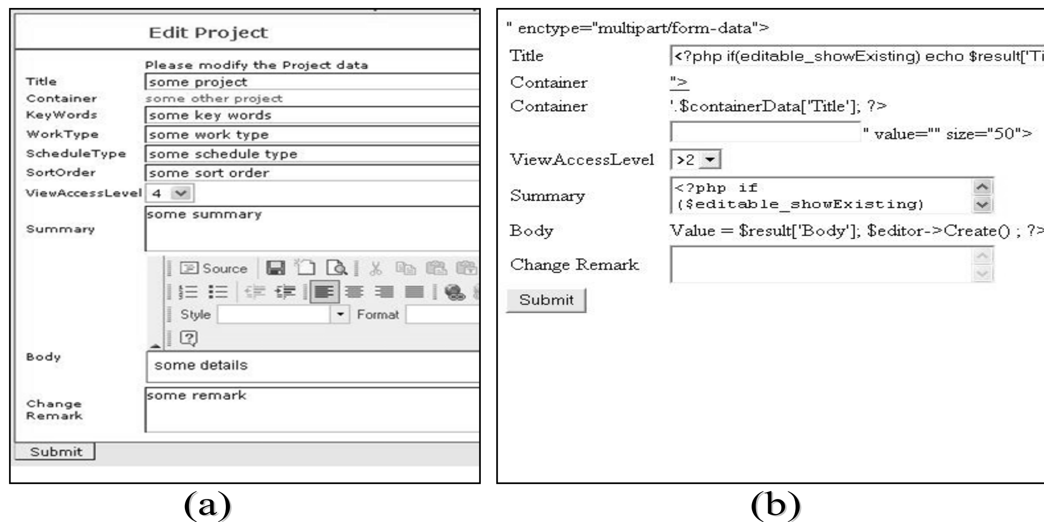


Figure 47. Effect of clone unification on WYSIWYG editing

When using mixed-strategy, this trade-off can be minimized by generating an extra WYSWYG- friendly version which has less program logic. Non-programmers will work on this additional version, while programmers are in charge of back-propagating the changes to the XVCL version and then forward propagating it to the real source code. This scenario is illustrated in Figure 48.

Trade-offs in WYSIWYG editing capabilities depend on the sophistication of the editor used, not the platform or the application domain. We note however the existence of editors claiming to have high-end WYSIWYG editing capabilities for .NET (e.g., Visual Studio.NET) and JEE (e.g., NitroX[32]).

[31] We used Microsoft Frontpage editor to generate this view

[32] NitroX home page: http://www.m7.com

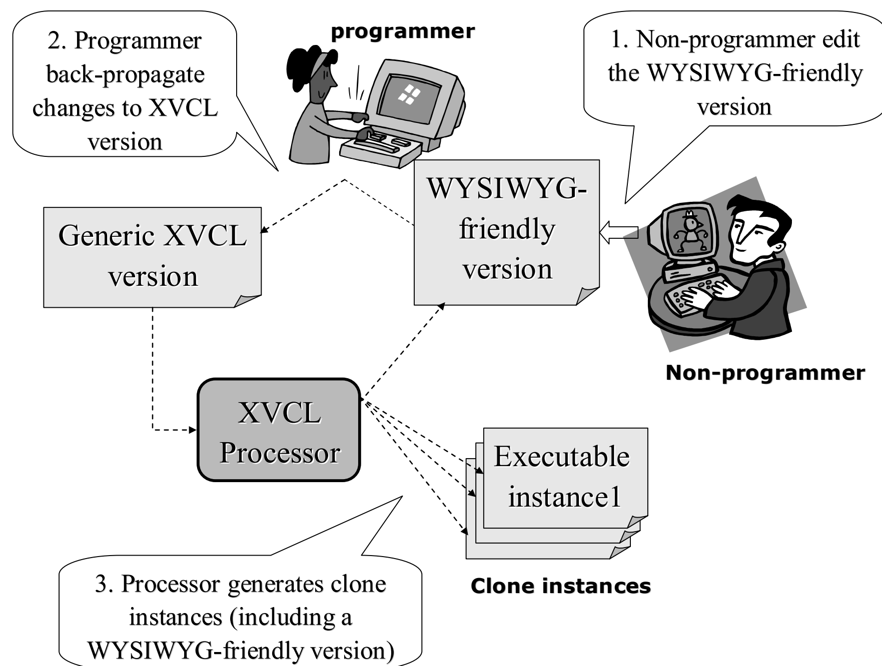


Figure 48. WYSIWYG editing when using mixed-strategy

6.2.7. Difference in runtime structure

In some rare cases it may be necessary to clone an existing system and change its runtime structure. Possible reasons for this include:

- To fit a new API/framework/platform (e.g., to deploy PCE modules on a different Foundation)
- For better performance (e.g., PCEsimple Vs PCEunified)
- For compatibility with other legacy systems at the deployment-site (e.g., to integrate with a legacy system that uses an old version of PHP)

These situations lead to clones across WAs with different runtime structures. Although our reasons for having three PCEs were quite different from those given above, we too found ourselves in a similar scenario: We had to maintain three separate WAs having drastically different runtime structures, yet having much similarity among them. For example, 55% of the code of PCEsimple was found to have a cloned counterpart in PCEpatterns.

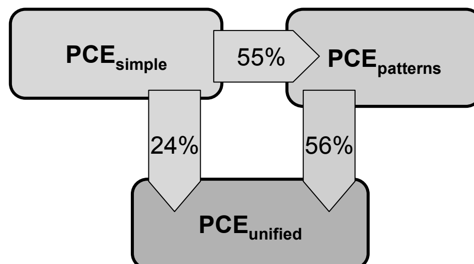


Figure 49. Similarity across three conventional PCEs

Unification of such clones requires some alignment in the runtime structures, forcing compromises to the objectives of having different runtime structures in the first place.

As we know, XVCL is agnostic to the semantics of the runtime code. Hence we can easily use mixed-strategy to unify highly similar WAs that have different runtime structures. For example, we can have one generic PCEall that generates all three PCEs (see Figure 50)

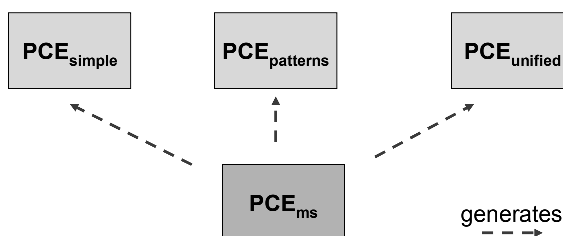


Figure 50. Using XVCL to unify all three PCEs

Again, this is a trade-off that does not depend on the specific platform or the application domain.

6.3. Discussion of results

It is not practical to find a silver bullet solution to the unification trade-offs problem. There could be many other trade-offs not mentioned in this chapter, and often trade-offs have interdependencies among them. Although we showed that mixed-strategy can avoid the observed trade-offs, it also adds its own trade-offs, notably the extra effort required to create the XVCL version.

From our analysis of the trade-offs in the previous section, we see that there are three fundamental reasons behind the mixed-strategy's ability to avoid the trade-offs.

1. XVCL does not remove clones from runtime – This means trade-offs incurred when removing clones from the runtime code are avoided (e.g., 6.2.1.Performance, 6.2.3.Framework conformance, 6.2.5. Indexing by search engines, 6.2.7.Difference in runtime structure).

2. XVCL can maintain different versions in sync – This allows us to keep multiple versions of the software for different requirements (e.g., 6.2.6.WYSIWYG editing, 6.2.4. Tidiness in source distribution)

3. Clone unification can proceed independent of source code – This means clone unification can be done when there is less schedule pressure (e.g., 6.2.2.Rapid prototyping/evolution capabilities)

We have reasons to believe that the limits involved in using Server pages, at least in the context of situations discussed in 6.2, apply independently of the platform on which these techniques are used. However, further work is required to support or dismiss the above claim. Other related technologies requiring similar further work include web application frameworks (e.g., Struts, Ruby on Rails), incarnations of Server page technique in other languages (e.g., ColdFusion[33]), template engines (e.g., Velocity[34]), and transformation techniques (e.g., XSLT).

As a final note, a caveat about generalizing our findings is in order. Since our study was conducted as a controlled lab environment, it carries the inherent weaknesses of generalizing from a lab study to industry practice. We tried to mitigate this shortcoming by using

---

[33] http://www.adobe.com/products/coldfusion/

[34] http://jakarta.apache.org/velocity/

functional requirements and the conceptual model of a real WA to build PCE, by reusing industry accepted architectures and frameworks as the core of our implementation, by following the design best practices used in the industry in PCE design, and by maintaining a tight feedback loop with our industry partner throughout the experiment. In the end, the size and the cloning level of PCE were also comparable to a similar WA built by our industry partner [PJ05].

6.4. Chapter conclusions

Using an empirical study, we showed that it was technically feasible to use conventional techniques to unify most of the clones, yet such unification forces trade-offs in many important WA properties.

Similar clone unification levels could be achieved using the mixed-strategy, more importantly, without incurring such trade-offs. Therefore, mixed-strategy offers a viable approach to avoid unification trade-offs.

Chapter 7.

Structural Clones

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

-Alan Kay

This chapter illustrates the concept of structural clones and shows how a particular definition of structural clones can help in managing fragmented structural clones.

Structural clones are higher level clones that represent repeated structures, resulting from a repetition of a high-level concept. Most structural clones consist of a large number of fragments at the implementation level, as a result of what we call 'clone fragmentation'. Clone fragmentation is the phenomenon of coarse-grained clones actually manifesting as scattered patterns of finer-grained clones (i.e., similarity patterns), resulting in an intractable number of small clones that we have to deal with. This fragmentation is a result of decomposition forces of the implementation technology, further exacerbated by injection of variations.

The organization of this chapter is as follows:

Section 7.1 illustrates the concept of structural clones using examples from various software systems. **Author wishes to acknowledge that this section is based on a joint paper (currently under review) by the Author, Basit, H. A., and Jarzabek, S.**

Section 7.2 first discusses how clone fragmentation adversely affects clone management. Then it shows how Basit's definition of structural clones can help in managing fragmented structural clones, using Java Adventure Builder model application as an example.

Our contribution contained in this chapter is:

Contribution 5. It illustrates the concept of structural clones using examples from various software systems. It shows how fragmented clones can be treated as structural clones.

7.1. Some examples of structural clones

This section illustrates the concept of structural clones with a number of examples, taken from real systems or case studies.

7.1.1. Example 1: a file-level structural clone

Figure 51 shows three structural clones we found in a PHP web portal. In this example, program entities that make up the structure are functions (we show only two functions here) and HTML code fragments (e.g., `staff_html`, `task_html`, `project_html`). Entities shown in the same shade are clones of each other (e.g., `staff_fn1`, `task_fn1`, `project_fn1`). This structural clone consists of three entities occurring in the same file, regardless of the order in which they appear. The three host files `editStaff.php`, `editTask.php` and `editProject.php` perform similar tasks, but belong to three different modules (i.e., `Staff` module, `Task` module, `Project` module).

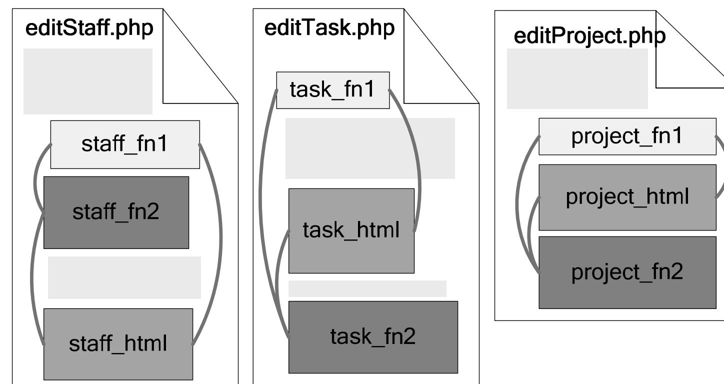


Figure 51. File-level structural clones

File-level structural clones are a common phenomenon in software. Duplication of certain parts or whole of a file (e.g., interface, logic, method call structure) or a set of files (e.g., a module) is a common practice. To illustrate this case, consider a situation where a child class needs to be added to an abstract parent class. An already existing child class may be cloned to create a sibling, in order to duplicate the code related to overriding abstract methods of the parent. Another likely cause for file-level structural clones is cloning in piece-meal fashion. That is, only a part of the file is cloned initially, but as the developer realizes more and more similarity between the reuse context and reused context, more and more code fragments are cloned from original file to new file. These new fragments may be inserted into locations of the new file that are different from the original file (e.g., a method copied from the original class may be pasted in a different location in the destination class), resulting in different ordering in cloned fragments across files (as the case in Figure 51).

7.1.2. Example 2: a module-level structural clone

A structural clone of higher granularity can be made up of structural clones of lower granularity. For example, a module-level structural clone can consist of file-level structural clones. Such a situation is illustrated by the structural clone we found in a web portal implementation, (shown in Figure 52). In this portal, files belonging to each module are

stored in a separate folder. Each module contains a set of files providing module-specific implementation of certain common functionalities (e.g., create, display, edit, delete). When the module functionalities are similar, each of these common files ends up being file-level structural clones of their counterparts in other modules. One such case was the basis for our previous example. At a larger granularity, the modules `Staff`, `Task` and `Project` can be considered structural clones where each structure has four files `create[M].php`, `display[M].php`, `edit[M].php`, `delete[M].php` (`[M]=Staff, Task, Project`) Note that the `Project` module does not carry a `deleteProject.php` file. Still, there was enough similarity among `Project` and other modules to consider all of them structural clones of each other.

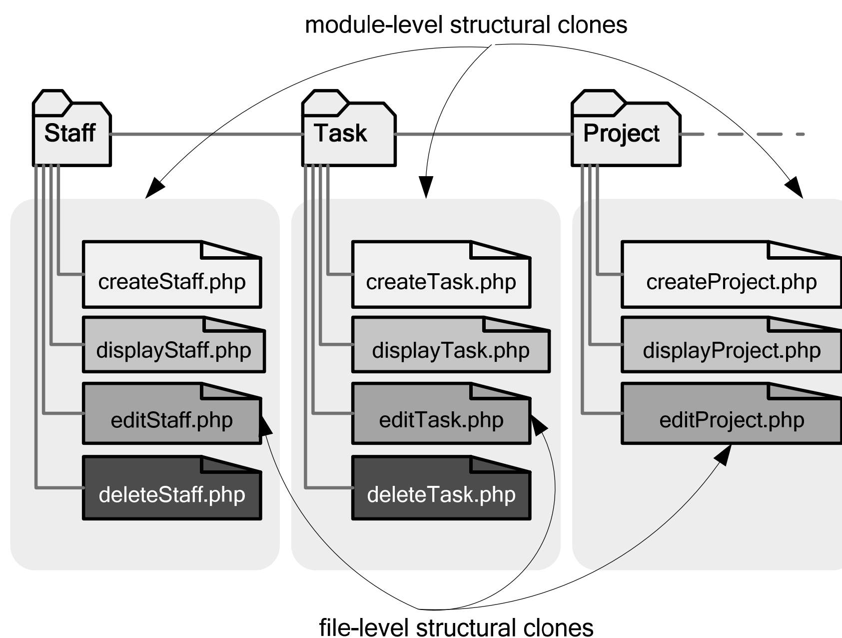


Figure 52. Module-level structural clones

Similar to file-level structural clones, similar modules of an application may be created by copying the whole module and modifying contents, leading to module-level structural clones.

7.1.3. Example 3: multiple structural clones in the same file

Multiple structural clones can also exist in a single file. Figure 53 shows an example we found in the C++ Standard Template Library (this case study is described in [BRJ05a], a summary is given in section 4.2). The four structural clones are structures of code fragments that are part of the different templates representing various hashed associative containers.

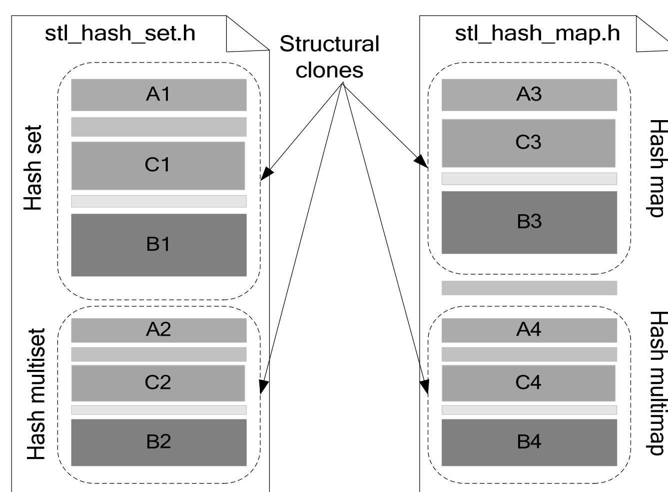


Figure 53. Multiple structural clones in one file

7.1.4. Example 4: crosscutting structural clones

Structural clones can crosscut files (or classes, modules etc.), as the example in Figure 54 shows. This example involves three PHP files belonging to PCE that supports two similar crosscutting features. This results in two structural clone instances, each consisting of code fragments belonging to one of the two crosscutting features.

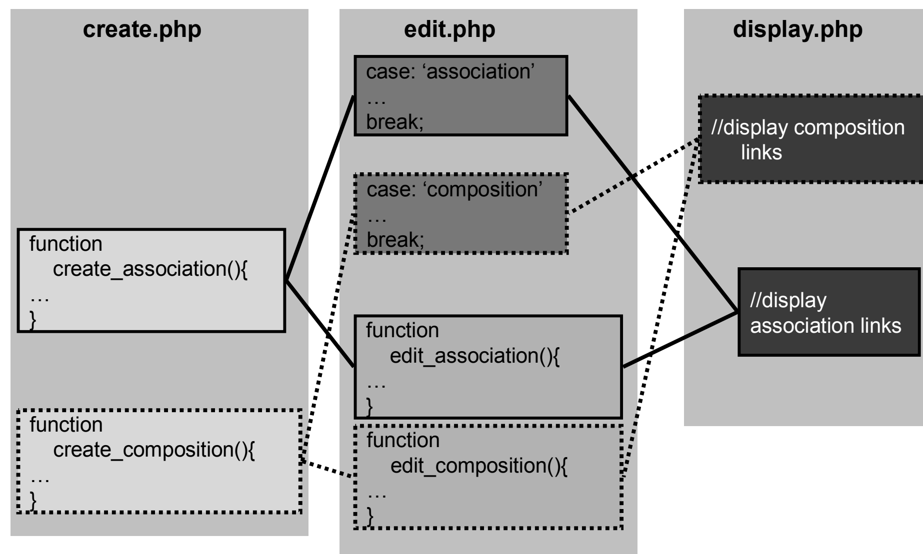


Figure 54. Two crosscutting structural clones

7.1.5. Example 5: heterogeneous entity structural clones

Sometimes, entities that constitute a structural clone may represent different types of program units. This is exemplified by a structural clone we found in another web portal. In this case (shown in Figure 55), the module-level structural clone involves a set of four files and a code fragment in a database script file common to all modules. Not only are the entities of this structural clone of different type and granularity (files and code fragments), but also they are implemented in different languages (PHP, SQL). Also note that four PHP files and SQL file reside in different directories.

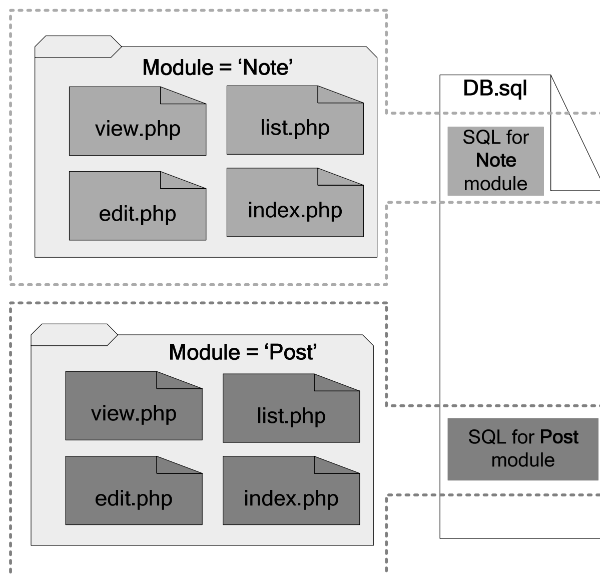


Figure 55. Structural clone with heterogeneous entities

7.1.6. Example 6: structural clones based on inheritance hierarchy

In object-oriented systems, a set of classes related by inheritance can be considered a structural clone. We found such a case in the Buffer library (java.nio.*) of J2SE 1.5. Figure 56 shows two instances (out of seven) of the structural clone, each consisting of seven Java classes. More information on the structure of the Buffer library was given in 4.1.

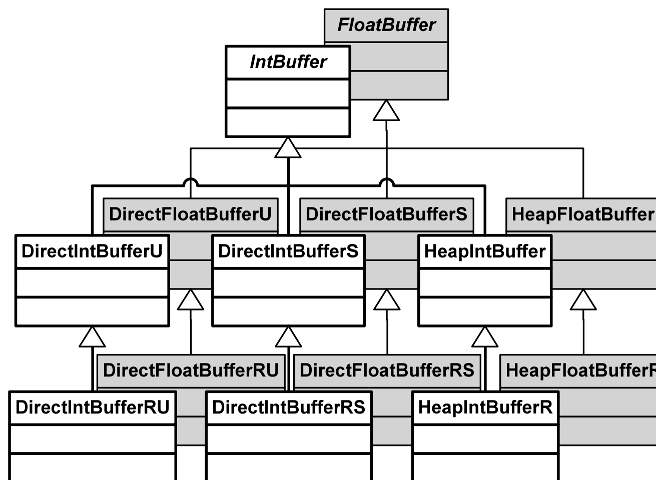


Figure 56. Structural clone based on inheritance

7.1.7. Example 7: a structural clone spanning multiple layers

The entities of a structural clone, especially in a one that is at a higher level and hence more beneficial, can be strewn across an entire system. The example given in Figure 57 (found in an industrial software system implemented by our industry partner ST Electronics using C#) shows how the constituent entities (related by association links) span multiple layers of an n-tier system.

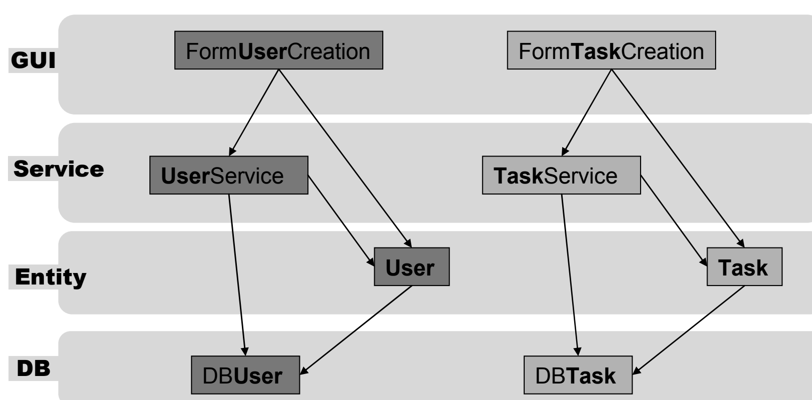


Figure 57. Structural clone spanning multiple layers

These clones arise from the following situation: The system is based on over 20 domain entities such as `User` or `Task`. The design and implementation of operations (such as `Create`) for various entities are characterized by a similar pattern of collaborating classes across GUI, service and database layers. Each box in Figure 57 represents an abstract entity consisting of a number of classes.

7.2. Structural clones and clone management

7.2.1. Fragmentation of structural clones

From the examples we saw in the previous section, it's clear that a structural clone is a repeated *structure* and each structure consists of a number of *entities* that are related to each

other in some way. For example, a structural clone could be a structure of code fragments in a file (where entities are code fragments, and the relationship between them is that they are 'in the same file').

A structural clone is a repeated *structure* and a structure consists of a number of related *entities*.

Structural clones by their very nature imply a fragmentation of the clone into multiple entities (modules, files, code fragments, etc.). This clone fragmentation is exacerbated by injection of variations, since variations further fragment the similarities of a structural clone to smaller pieces. Clone fragmentation negatively affects the clone management in the following ways:

- Fragmentation makes it difficult to identify the original coarse-grained clones. This poses major problems in clone detection. This sub-problem is addressed separately by Hamid Abdul Basit, a member of our research group (e.g. [BJ05]).
- Individually unifying a large number of clone fragments adversely affects the system structure.
- Reusing of small clone fragments is less beneficial, and more difficult to justify, than reusing the larger coarse-grained clone.
- Clone fragments are not always meaningful on their own, although a group of fragments might still make sense.

Effects of clone fragmentation is indirectly evidenced by the work of Synytskyy, Cordy, and Dean [SCD03], which declares that not all clones in a web site are worth unification; A substantial number of the clones are too small, or too insignificant to be of interest, and their unification would only complicate the existing structure of the site unnecessarily.

7.2.2. Clone fragmentation in web domain

Web application frameworks (section 2.3.2) that combines best of the breed web technologies also force the application logic to be fragmented into many number of configuration files, scripts, templates, etc. This exacerbates the clone fragmentation problem. For example, in the Adventure builder model application from Sun Microsystems (to be described in section 7.2.4) we found a clone consisting of XML configuration files, XSL style sheets, build scripts and Java source code which was also fragmented into numerous additional files as required by the EJB framework, and the web Services standards.

7.2.3. Structural clones as 'configurations of lower level clones'

Basit H. A. views structural clones as "higher level clones that are configurations of lower level (smaller) clones" in devising an approach to detect higher level structural clones from their fragments (work on this approach is currently in progress, initial results can be found in [BJ05]). The same view comes in handy when managing structural clones that have been fragmented into patterns of smaller clones. Using this view we can construct a hierarchy of clones (called an 'SC hierarchy') with cloned code fragments at the bottom level; Clones at each level consist of configuration of smaller clones (called 'clone structures') from the level below.

This idea of a SC hierarchy is illustrated in Figure 58 which shows one instance of an SC hierarchy (i.e., the structure shown has been cloned multiple times although only one instance is shown). The left half of the figure shows the physical view of the SC hierarchy while the right half of the figure shows the logical view. Suppose code fragments f1a, f1b, and f1c in file F1 make up a clone structure S1. Similarly, code fragments in files F2 and F3 make up clone structures S2 and S3, respectively. Then S1, S2 and S3 form a new clone structure S, at

the next higher level. For convenience we could decide to abstract S1, S2 and S3 into whole files F1, F2 and F3, provided these clone structures covers a considerable part of the respective files. Dashed arrows in the logical view indicate this abstraction. In summary, SC hierarchy of S consists of 9 clone fragments, and three levels.

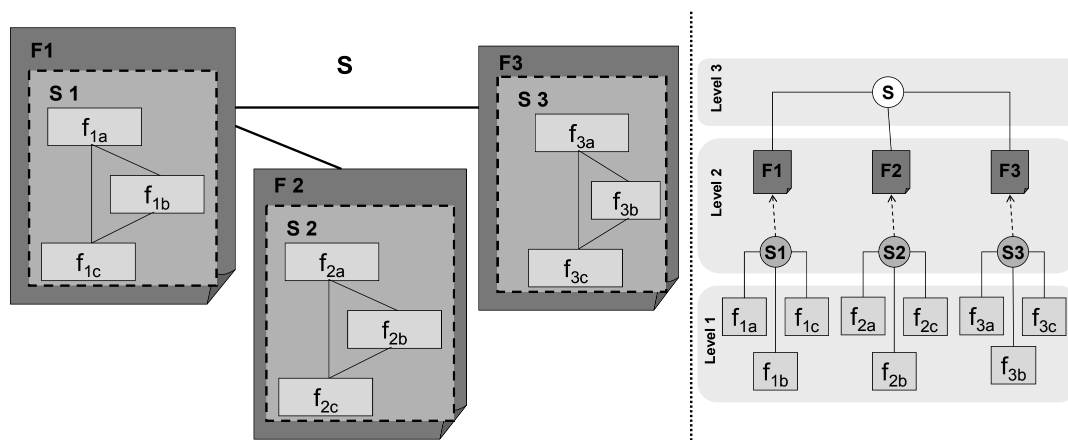


Figure 58. An SC hierarchy

7.2.4. A Complete example: structural clones in Adventure Builder

Next we give a real-life non-trivial example of a structural clone we found in a Java model application called the *Adventure Builder*.

Overview of Adventure Builder application

The Java Adventure Builder Reference application[35] is a set of sample applications that illustrate facilities in the J2EE 1.4, particularly the use of web services. The AB is developed by Java BluePrints group. At the center of AB is a fictitious e-commerce venture called *Adventure Builder Enterprise* that provides customers with a catalog of adventure packages, accommodations, and transportation options. From a browser, customers select accommodations, mode of transport, and adventure activities to build a vacation. Adventure

---

[35] http://java.sun.com/

builder enterprise also interacts with external partners, such as airlines, hotels, and activity providers, who supply the services or components of a vacation, and other partners, such as banks or credit card companies, who process payments for the enterprise.

The main part of AB, called adventure builder enterprise (ABE), consists of a front-end customer web site (CWS), which provides a face to its customers, and a back-end order processing center (OPC), which handles the order fulfillment processing. AB also include three external partner applications that provide lodging (LDG), air travel (AIR), and activity (ACT) components of the vacation, and a fourth external application that represents a bank. The high-level architecture of the AB sample application is shown in Figure 59.

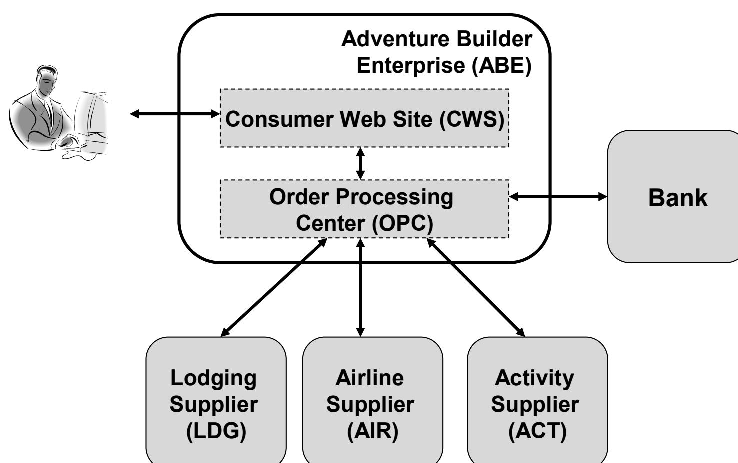


Figure 59. Architecture of the Adventure Builder application

Clone fragmentation in Adventure Builder

The three supplier subsystems in AB are conceptually similar, and an initial inspection suggests that the three supplier subsystems are clones. It was later confirmed that unifying those clones had the potential to reduce code by 50%, and reuse of the unified clone had the potential for 78% code saving when creating other suppliers.

Yet a manual inspection showed that the clone has been fragmented into 389 pieces of exact clones (from fragments shorter than a line, to fragments as large as a complete file) distributed in 27 files, and has 1070 fragment instances (distributed in 76 files). As these fragments were manually verified, the statistics does not include any false positives.

One option to sidestep the fragmentation problem is to filter out only the significant fragments and ignore the rest. However, the outcome of such an approach is unsatisfactory, as the following analysis shows.

- If we ignore fragments smaller than files, the clone degenerates into a single cloned file, covering only 3% of the actual cloned code.
- If we choose fragments that are semantic blocks (semantically meaningful blocks of code such as XML nodes, conditional blocks, or program statements) and longer than 5 LOC: the clone captures only 24 out of 389 fragments, covering only 28% of the code in the clone.
- If we lower the minimum block size to 2 LOC: the clone captures 65 fragments covering only 44% of the code in the clone (already too many pieces, yet too little coverage).

An interesting observation is that although fragments (not blocks) longer than 2 LOC cover 84% of the clone, most have to be discarded completely or trimmed at the beginning/end because they do not form semantics blocks. By adding the rest of the short fragments (i.e. fragments shorter than 2 LOC) that cover only14% of the clone, we can capture the full clone. This is because shorter fragments complement the larger ones in forming semantically meaningful blocks. This is why it is not wise to ignore the smaller fragments of the clone.

SC harvesting. We use the term 'SC harvesting' to describe the process in which an SC hierarchy is formed to explain how the clone fragments form the original coarse-grained clone. SC harvesting is guided by domain knowledge (i.e., knowledge of domain level similarities), and clone knowledge (i.e., knowledge of implementation level similarities). SC harvesting can be done in both top-down and bottom-up fashions. In the top-down approach, we identify coarse-grained entity structures. These entities are then broken into structures of finer-grained entities, until all entities are simple clones that match the simple clones found. In the bottom-up approach, we start from simple clones, and build the hierarchy upwards until we reach an entity coarsest possible granularity. Clone harvesting primarily based on domain knowledge favors a top-down approach, while clone harvesting based on clone knowledge favors a bottom-up approach. Typically a mixture of the two approaches is used in practice.

Clone harvesting in AB was based on both domain knowledge and clone knowledge. Our domain knowledge was based on AB documentation, and manual analysis of the implementation; Our clone knowledge was on data produced by CCFinder [KKI02], Gemini [UHK+02], and our own clone analysis programs. Our domain knowledge suggested that supplier systems LDG, AIR, and ACT performs very similar tasks and therefore likely to be a structural clones of each other (i.e. a structural clone class). This was further confirmed by clone knowledge, for example, we found high level of cloning across supplier systems (Figure 60 shows the percentage of code in one system that is duplicated in another).

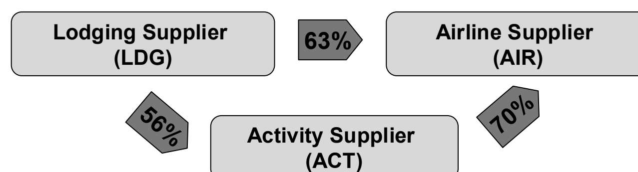


Figure 60. Cloning across three supplier system

Our next step was to try to move upwards in the SC hierarchy. For this, we looked for files related to three supplier systems, but residing outside supplier systems (i.e., in ABE

application). Such files, related to the integration between ABE and supplier applications, are also likely to be similar. There were indeed such files in both CWS and OPC subsystems. Then we did a three-way comparison of all the supplier-related files identified so far (results shown in Table 7) to find out the similarity level between corresponding files in the three systems. It uses four levels of similarity: exactly similar files (S), only parametric variations (P), extensive variations (E), totally different (D). 'O' indicates files with only one instance. Darker shades in last three columns indicate stronger similarity. Deep directory paths in 'Location' column indicates how widely dispersed the files were.

Table 7. Three-way comparison between files in the three structural clones

	Location	File	LDG	AIR	ACT
EXTERNAL SUPPLIER APPLICATION	[Home]\[S]supplier	1. build.xml	E	E	E
	[above]\src\conf	2. application.xml	P	P	P
	[Home]\[S]\[S]supplier-ejb	3. build.xml	P	P	P
	[above]\src\java\com\sun\j2ee\blueprints\[S]supplier	4. JNDINames.java	E	E	E
	[above]\pomessagebean	5. Invoice.java	E	E	E
		6. [S]MessageBean.java	E	E	E
	[above]\..\powebservice	7. InvalidOrderException.java	P	P	P
		8. OrderSubmissionException.java	P	P	P
		9. [S]Details.java			O
		10. [S]Order.java	D	D	D
		11. [S]POEndpointBean.java	P	P	P
		12. [S]POIntf.java	P	P	P
	[above]\..\purchaseorder\ejb	13. [S]DetailsBean.java			O
		14. [S]DetailsLocal.java			O
		15. [S]DetailsLocalHome.java			O
		16. [S]OrderBean.java	D	D	D
		17. [S]OrderLocal.java	D	D	D
		18. [S]OrderLocalHome.java	D	D	D
	[Home]\activitysupplier\[S]supplier-ejb\src\conf	19. ejb-jar.xml	E	E	E
		20. po-jaxrpc-config.xml	P	P	P
		21. sun-ejb-jar.xml	E	E	E
		22. webservicebroker-client-config.xml	P	P	P
		23. webservices.xml	P	P	P
OPC	[Home]\opc\opc-ejb\src\conf\	24. [S]supplier-client-config.xml	P	P	P
	[above]\..\java\com\sun\j2ee\blueprints\opc\purchaseorder\	25. [S].java	E		E
	[above]\ejb	26. [S]Bean.java	E		E
		27. [S]Local.java	E		E
		28. [S]LocalHome.java	E		E
	[above]\..\webservicebroker\provider	29. invoice-[S].xsl	S	S	S
	[above]\..\requestor	30. [S]SupplierClient.java	P	P	P
CWS	[Home]\consumerwebsite\web	31. cart_[S]_tab.jsp	O		
		32. [S]s.jsp	O		
	[above]\..\src\java\com\sun\j2ee\blueprints\catalog	33. [S].java	P		P

Altogether there were 82 files included in the clone. A summary of file similarity characteristics in AB is given in Table 8.

Table 8. Summary of file similarity characteristics in AB

	LDG	AIR	ACT	Total
S (exactly similar)	1	1	1	3
P (parametric variations)	12	11	12	35
E (extensive variations)	10	10	6	26
D (totally different)	4	4	4	12
O (only one file)	2	0	4	6
Total	29	22	31	82

With this knowledge, we formulated the first two layers of the SC hierarchy (see Figure 61). At the top is a coarsest-grained structural clone, marked as `SC1:SUPPLIER`, consisting of three entities `[S]ext` (external application for supplier `S`, where `S=LDG,AIR,ACT`), `[S]opc` (files in OPC that is related to the suppler `S`), and `[S]cws` (files in CWS that are related to the supplier S).

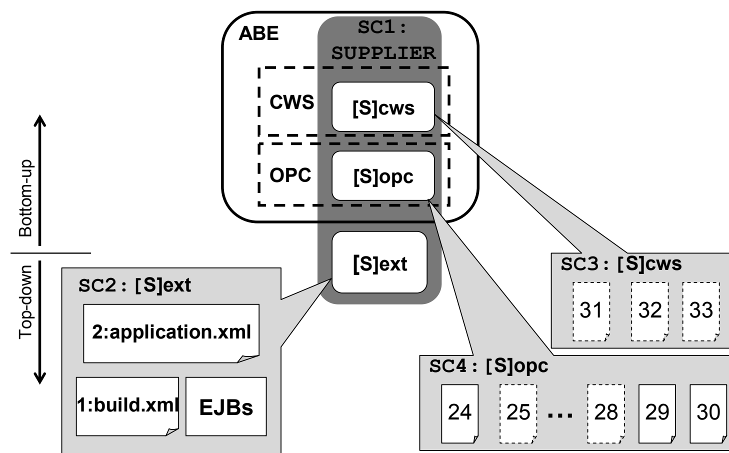


Figure 61. First and second tier structural clones in AB

In the second layer, we broke `[S]ext` into three finer-grained entities: file `build.xml`, file `application.xml` (files 1 and 2 in Figure 61) and a group of files called `EJBs` (consisting of files related to Enterprise Java Beans in `[S]ext`, i.e., files 3-23).This breakdown illustrates that all entities of the structure do not have to be of same type. It also shows that an entity does not have to map to a single physical entity, but can also be a conceptual entity such as a

group of files such as `EJBs`. Entities of `[S]opc` and `[S]cws` are files, some of which are optional (e.g., file 25), i.e., they may be omitted in some of the instances. Such 'optional entities' is a common structural variation between structural clones. We shall revisit common structural variations in section 8.7.

The entity `EJBs` can be further decomposed into 6 entities (4 folders, 2 files) as shown in Figure 62. The four folders contain file structures: folder `conf` contains a set of configuration files, folder `msg` contains source files for message oriented beans, folder `po` contains source files for purchase order entity beans, and folder `ws` contains source files for beans that handle web services.

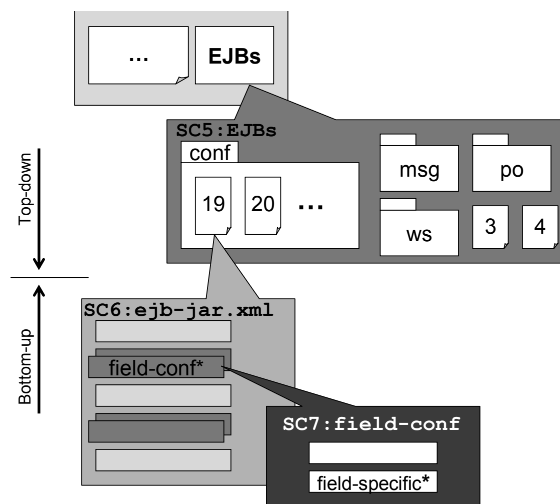


Figure 62. Third, fourth, and fifth tier structural clones in AB

Some files in the structural clones have extensive variations. Such files require yet more, finer-grained structural clones to explain the similarity patterns they contain. We illustrate this point using structural clones in `ejb-jar.xml`, which is an entity in the structural clone `conf`. Harvesting of structural clones in `ejb-jar.xml` was done in bottom-up fashion. We started with simple clones detected in `ejb-jar.xml` file and tried to fit them into structural clones that explains the contents of the file. This yielded the bottom two layers of this branch of the SC hierarchy. The structural clone at the bottom layer (`SC7` in Figure 62) consists of

xml code fragments, one of which is repeated for each field of the bean (marked with an *). Such entities that repeats a varying number of times from instance to instance are called 'repetitive entities'; another common structural variation. `SC7` itself is a repetitive entity in `SC6` (i.e., `ejb-jar.xml`). This is because field configuration has to be repeated for each bean included in the `ejb-jar.xml`.

Applying this technique to the rest of the clone fragments, we get the final complete structural clone that represents the whole of supplier subsystem, and map this high-level similarity to all 389 fragments of the clone. At the top of the hierarchy we have `SC1` which represents whole high-level clone as a single entity, abstracting over the 389 fragments.

7.3. Chapter conclusions

Coarse-grained clones get fragmented into patterns of finer-grained clones due to large number of variations introduced during the lifetime of a clone. This phenomenon, named as the 'clone fragmentation' by us, hinders clone management.

In this chapter we showed how the definition of a structural clone as a "configuration of lower-level clones" provides an excellent abstraction mechanism to explain the fragmented coarse-grained clones in terms of the resultant clone fragments.

Using Adventure Builder model application we illustrated the phenomenon of clone fragmentation, and how structural clone concept was used to compensate the effects of clone fragmentation.

Chapter 8.

SuM: Structural Clone Management Using Mixed-Strategy

Time and time again, I've found that by simply removing duplication
I accidentally stumble onto a really nice elegant pattern.

-Martin Fowler

This chapter presents SuM (Structural clone management using Mixed-strategy) as a holistic approach to overcome the challenges of tenacious clones, and clone fragmentation. It systematically describes the basic activities and techniques of applying SuM, and documents the basic SuM unification schemes.

A number of previous studies have successfully used mixed-strategy to unify clones when other clone unification techniques failed. Structural clones(SC) is an emerging concept that uses a hierarchical organization of clone fragments to form coarser-grained clones, thus overcoming the challenge of clone fragmentation. In SuM, we bring these two approaches together. In essence, SuM is an approach to systematically apply mixed-strategy within the SC paradigm, by aligning mixed-strategy solutions along SC boundaries.

A question might arise in the reader's mind as to why structural clone should be imposed on mixed-strategy to form SuM, since mixed-strategy on its own has already been applied successfully in a number of studies (e.g., [JL03], [BRJ05a], [YJ05], [PJ05]). The answer lies in SuM's aspiration to systematize the application of mixed-strategy. Although it is possible

to apply mixed-strategy to unify coarse-grained clones, the hierarchical organization of fine-grained clones into coarse-grained clones (represented by the final x-framework of the XVCL solution) is guided by intuition and best judgment of the CMP (Clone Managing Personnel[36]), rather than by systematic reasoning. Such an approach may work for small scale, moderately complex projects done by individual developers or small teams. But to scale up the mixed-strategy we need more systematic basis of structuring XVCL solutions.

Clone unification represents a vast problem space due to highly diverse nature of coarse-grained clones. When using mixed-strategy, XVCL's independence from the semantics of the underlying program greatly reduce this problem space. However, given a highly fragmented coarse-grained clone, still there are many ways to structure the mixed-strategy solution. In SuM we further reduce this diversity by first organizing the coarse-grained clone as an SC hierarchy and then aligning the mixed-strategy solution along the SC boundaries in the hierarchy.

Identifying the SC hierarchy requires identifying the generic form of the clone, together with its variation points. The challenge is to extract this information from a limited number of actual variants that are present. This chapter provides a good starting point to tackle this problem by identifying basic entity types, basic structure types, basic structural variation types and showing how SuM handles each.

The organization of this chapter is as follows:

Section 8.1 outlines the activities involved in managing clones using SuM.

Section 8.2 describes in detail the activities leading up to the clone unification using SuM, i.e., pre-unification activities of SuM.

---

[36] We use the term CMP as a generic reference to the person doing the clone management (can be developers, reengineers, or maintainers)

Section 8.3 gives a detailed description of the clone unification activity of the SuM.

Section 8.4 describes the activities involved in maintaining and reusing a SuM solution, i.e., post-unification activities of SuM.

Section 8.5 illustrates some of the basic techniques described previously, using the Java Adventure Builder model application.

Section 8.6 discusses how the diversity of clones affects clone management. It shows how XVCL helps to shrink this problem domain. It then identifies basic structure types and describes the SuM technique to handle each.

Section 8.7 presents the basic SuM unification schemes, that is, basic structural variations together with a recommended SuM solution for each variation.

Our contributions contained in this chapter are:

Contribution 6. It presents SuM, a combination of mixed-strategy and the structural clone concept to provide a systematic and holistic approach to unify and reuse tenacious, and possibly fragmented structural clones, without compromising their benefits.

8.1. Clone management using mixed-strategy

Mixed-strategy can be viewed as a powerful compensatory clone management technique since it does not remove clones from the code (i.e., it is not a corrective technique); Rather, it creates another meta-program level view of the clones in the system, so that we can compensate the negative effects of clones.

> Mixed-strategy is a powerful compensatory clone management technique

Mixed-strategy can be used to manage existing clones as well as potential clones (i.e., at the point of cloning).

Managing existing clones

When applying the mixed-strategy to manage existing clones, we can use a mix of corrective or compensatory clone management.

(a) Corrective - Use conventional techniques to remove the clones (e.g., use refactoring).

(b) Compensatory - Use XVCL to unify remaining tenacious clones at meta-level.

As long as only action (a) is sufficient to manage all the clones in the application, it remains a conventional application. If the action (b) was chosen at some point, the application becomes a mixed-strategy solution.

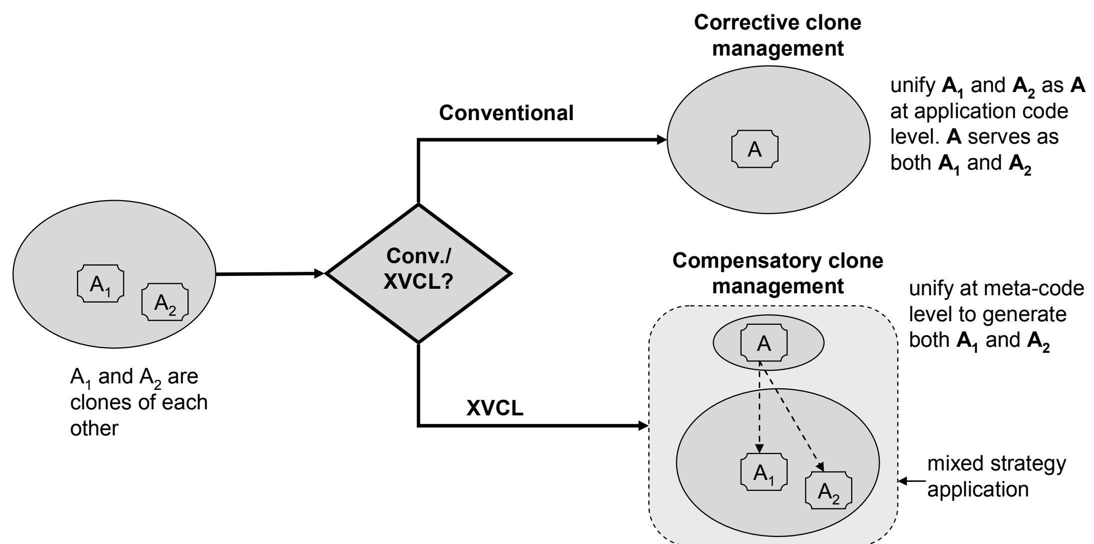


Figure 63. Applying mixed-strategy for managing existing clones

Generative clone management

Large clones with many repetitive variations, such as those found in product lines, are labor-intensive to create. This is more so if multiple variants need to be created. Mixed-strategy can be used in such cases for a new type of clone management that we call 'generative clone management'. Generative clone management aims to increase the productivity of creating clones by automating the clone creation process. Here we use Mixed-strategy's ability to "automate the derivation of specific structures from their generic forms". Figure 64 shows

how cloning an existing code B_1 to create two new instances (B_2 and B_3) can be managed by using generative clone management; i.e., we first bring the original code to meta-program level and use it to generate the new clones as variants of it.

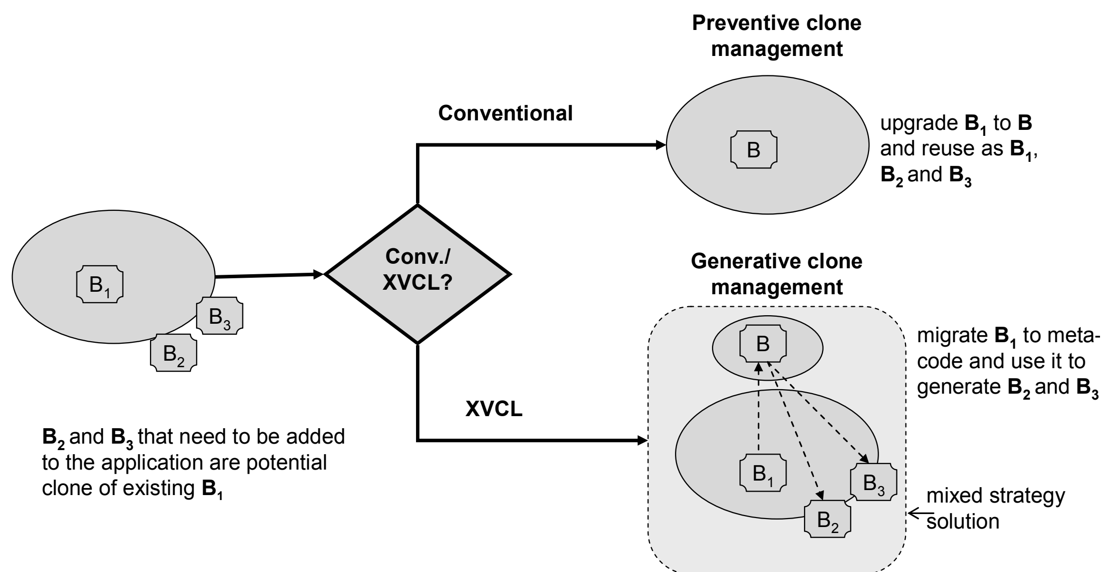


Figure 64. Applying mixed-strategy for managing potential clones

> Mixed-strategy can be used for a new type of clone management: generative clone management

Modern IDEs too offer inbuilt functionality to automatically generate clones (e.g., IDEA[37]). But unlike those, the XVCL approach applies compensatory clone management at the same time, mitigating the negative effect of the created clones.

Managing potential clones

A potential cloning situation arises when we need to create software that is similar to existing software (e.g., when adding a new module that is very similar to an existing module).

[37] http://www.intellij.com

Applying the mixed-strategy to such cases involves a mix of preventive and generative clone management.

(c) Preventive - Use conventional techniques to avoid the clones (e.g., by using runtime customization).

(d) Generative - Use XVCL to generate a clone from existing code.

Proactive and reactive clone management

Preventive and generative clone management actions represent proactive clone management, applied in potential clone situations. Corrective and compensatory clone management actions are their reactive counterparts, applied to existing clones. This situation is summarized in Table 9.

Table 9. Clone management actions using mixed-strategy

	choice: conventional techniques	**choice: XVCL**
Proactive (for potential clones)	preventive	generative
Reactive (for existing clones)	corrective	compensatory

Clone management using mixed-strategy consists of the following activities, also illustrated by Figure 65.

i. Pre-unification activities

a) Identify clones

b) Analyze clones: gather more clone info to aid in unification

ii. Unification

a) Choose technique: decide whether to use conventional techniques or XVCL

b) Apply the chosen technique: results in a mixed-strategy application

iii. Post-unification activities

a) Maintain: co-evolve application code and meta-code (this includes understanding and modifying the mixed-strategy application)

b) Reuse: when a unified clone is reused to accommodate another clone instance (generative clone management)

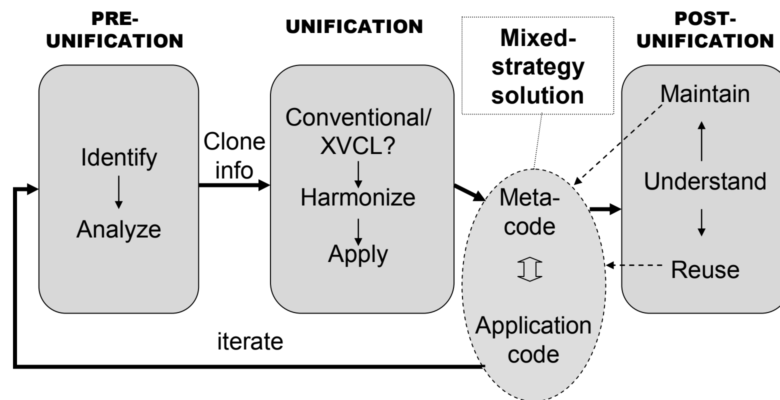


Figure 65. Clone unification activities using mixed-strategy

Usually these activities are done in iterative fashion. Emphasis on each activity varies depending on the context of usage (e.g., identification plays a major role during reengineering).

In the next subsections we describe those activities in detail. Where appropriate, we illustrate the activity using an example from the clone unification we did in PCE (described previously in section 6.1.1) and AB (described in previous chapter):

8.2. Pre-unification activities

8.2.1. Clone identification

Clone identification can be done using various means. Usually it is done using a combination of techniques.

- Manual:
    - Using domain knowledge – some clones are directly caused by similarities in the domain. A CMP can use domain expertise to track down such clones
    - Using prior knowledge – if the CMP is the same person who created the clones, she can identify them later from memory
    - Using manual inspection
- At the point of cloning
- Automatic: There are many tools available for clone detection (e.g., CCFinder [KKI02], Clone Miner [BJ05]).

Example from PCE, AB:

- In PCE, we used feature diagrams to identify potential clones early, during the analysis stage. This was useful in preventing similar things being implemented in different ways (which is worse than cloning!).
- Since PCE was a small, single-person project, some clones could be identified from memory.
- In both PCE and AB, we used CCFinder/Gemini to detect remaining clones after an iteration of clone unification. Using the plain text mode, we could run the detection for the whole of mixed-strategy application, including meta-code. This was useful in identifying overlooked clone instances of already unified clone classes.

8.2.2. Clone analysis

Clone analysis helps us to make informed decisions during clone unification. Usually clone identification and analysis are done together.

- Manual – manually analyze clone
- Semi-automatic – using tools like Gemini [UHK+02]

It involves following tasks (examples from PCE and AB are given where applicable).

- Calculating clone metrics – metrics help in categorization, prioritization, filtering. E.g., we used estimated code deflation[38] (reported by Gemini) and clone length for PCE and AB.
- Clone visualization – visual representation of the clones helps in analysis. E.g., we used the scatter plot produced by Gemini to locate highly cloned regions of PCE and AB.
- Clone pattern analysis – this helps to identify bigger patterns of clones (*cf* "structural clones harvesting" introduced in section 7.2.4). E.g., we used manual analysis (aided by Gemini) to harvest structural clones in PCE and AB.
- Clone categorization – this helps in prioritization and filtering. E.g., we categorized clones in PCE as intra-module and inter-module clones. In AB, we used the extent of variations (see Table 7) to categorize cloned files.
- Prioritization – this helps to decide the order of unification. Prioritization becomes important when working under tight time constraints. E.g., we prioritized our clone unification based on effort-to-benefit ratio (i.e., clone classes that were easier to unify and provided higher code deflation were unified first).
- Filtering – this helps to remove unimportant clones from the analysis. E.g., we used Gemini to filter out clone classes with lower code deflation potential.

[38] Estimated code reduction if the clone class were to be unified

8.2.3. Choosing the unification technique

When choosing the unification technique, we compare corrective clone management (i.e. using conventional methods such as refactoring using JavaScripts or PHP) against compensatory clone management using XVCL.

Factors typically considered when making this decision:

- Effort – Is the solution known, or does it require further exploration? How much additional effort is required? Does it require the introduction of a new technology? How much additional testing is required?
- Impact – Are the changes localized or dispersed? Does it require changes to unrelated code? Modification to widely reused code may have wider impact than we are prepared to tolerate.
- Risk – In most cases clone unification requires changes to existing code. Sometimes the risk of modifying operational code is unacceptable, especially when there is no functional benefit.
- Non-functional trade-offs – does the solution affect performance, maintainability, reusability, etc.? (*cf* section 6.2 for examples of trade-offs)

Examples from PCE, AB

In PCE experiment, we accepted PCEpatterns as the limit to which conventional techniques should be pushed to unify clones. This was an arbitrary decision, taken for the purpose of illustration. In AB, we accepted the current implementation as the optimum level of clone unification using conventional techniques, and applied XVCL to unify the remaining tenacious clones. This was because AB is a model application developed by a reputable software company, and it can be argued that any remaining clone exists for a valid reason.

8.2.4. Clone harmonization

Optionally, we may do minor refactoring to optimize the unification. We call this *clone harmonization*. This includes activities such as,

- Removing unintentional variations (e.g., by renaming variables)
- Reducing variation points by localizing variations (e.g., by introducing local variables)
- Align variation points to achieve symmetry of variations - (e.g., by reordering statements, adjusting whitespace)

```
void p(int a){
int temp = 0;
for(temp 0..a){
  //do something
}
```

```
void process(int times){
int iter = 0;
for(iter 0..times){
  //do something
}
```

Figure 66. Harmonization example

A situation where harmonization is beneficial is shown in Figure 66. Functions `p` and `process` are clones of each other, but the variations seem to be unintentional and unnecessary. This is a typical example where a clone has been improved after it has been created (in this example, `process` has been improved by replacing variable/parameter names with more meaningful names). The most appropriate harmonization action in this case is to change `p` to match `process`.

Except in trivial cases, harmonization involves regression testing. One advantage of harmonization is that it eases the task of automatically comparing the generated clones with original clone that we intended to generate. For example, if the generated clone differs from the original clone in terms of white space or a local variable name (i.e., they are logically equivalent, yet textually different), comparison tools may report this as an error. Harmonization helps to avoid such incidents.

Examples from PCE, AB

In PCE we performed all three types of clone harmonization described above. In AB, we renamed variables and adjusted whitespace to harmonize some clones.

8.3. Unifying clones using SuM

8.3.1. Representing an SCC with the master

'Master SC hierarchy' ('master' for short) is the generic reusable, representation of the structural clone class (SCC). It captures the commonalities and variation points. A variant instance (i.e., a member of the SCC) may be derived from the master by configuring predefined variations points or injecting new variations. In an incremental approach, master evolves as more variations come to light, and our understanding of the generic representation grows. Several strategies exist for choosing the starting point for a master (e.g., choose the most feature-rich instance, choose the most common parts, choose a typical instance). Determining the master is an important part of the clone unification process, because unifying the structural clones is in fact nothing more than converting the master into an XVCL representation.

Master is the generic, reusable, representation of the structural clone class.

Identifying the master is guided by the following:

- Concrete clone instances present and foreseen
- The perceived domain concept that is being repeated

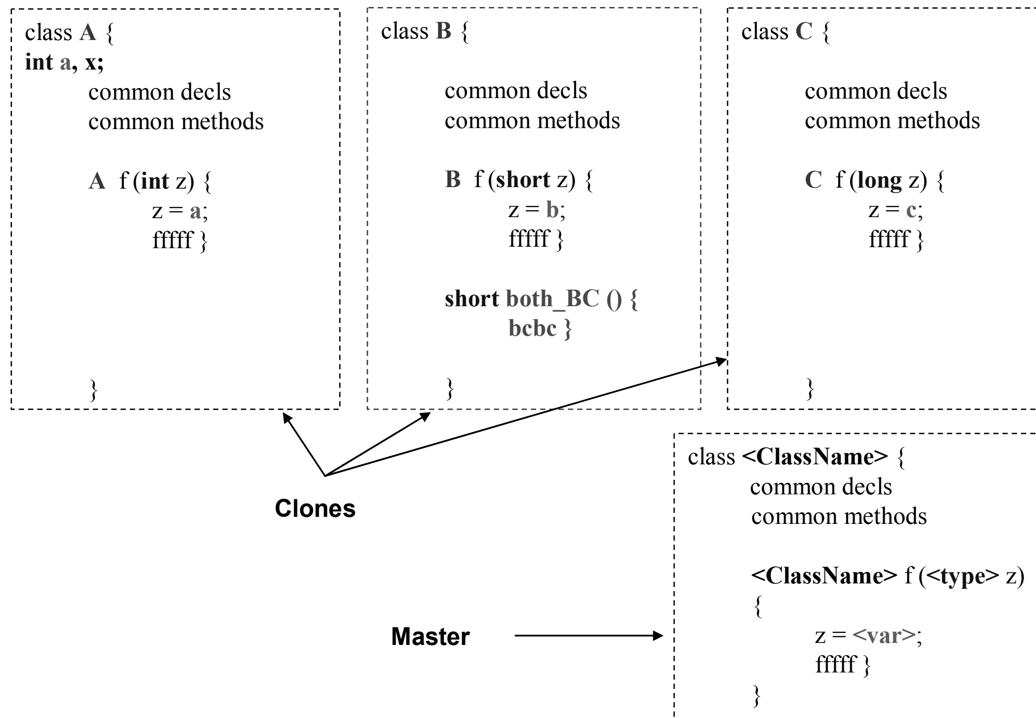


Figure 67. Choosing master based on clones, an example

In Figure 67 we see an example of how a master has been chosen based on three simple clones instances present. In this occasion, the strategy that has been followed d is to include only the parts common to all clone instances in the master.

8.3.2. Unification activities

Unification of existing clones (i.e., when applying compensatory clone management) using XVCL typically involves following tasks:

- Identify the master SC hierarchy – This includes identifying the variation points in the master and deltas for variants – diff tools (code comparison tools such as UNIX diff utility) can support this activity
- Further harmonization – Further harmonization may be required after identifying master, and variation points.

- Unify using XVCL (i.e., create meta-code that generates the clones). This can be done using following steps:
    - Frame the master – i.e., create the x-framework that generates the master. This is the generic representation of the clone.
    - Absorb other variants – In an incremental framing approach, first the master is framed, and then variants are absorbed in to the XVCL solution incrementally. In trivial cases this can be done in one shot, when framing the master.
    - Verify output – This involves verifying whether the code generated is same as the expected, which is usually the original application code. This can be done by a simple diff tool. More sophisticated support could be provided in an XVCL development environment.
    - Debug – In the cases where generated output differs from the expected, errors in framing need to be found and fixed, typically using an XVCL debugging tool.

Figure 69 shows a recap of the clone unification activities using SuM. Unification of potential clones (i.e., when applying generative clone management) involves similar tasks. Usually the existing code (the original) is chosen as the master. Unlike in the case of existing clones, the newly generated clone needs to be tested.

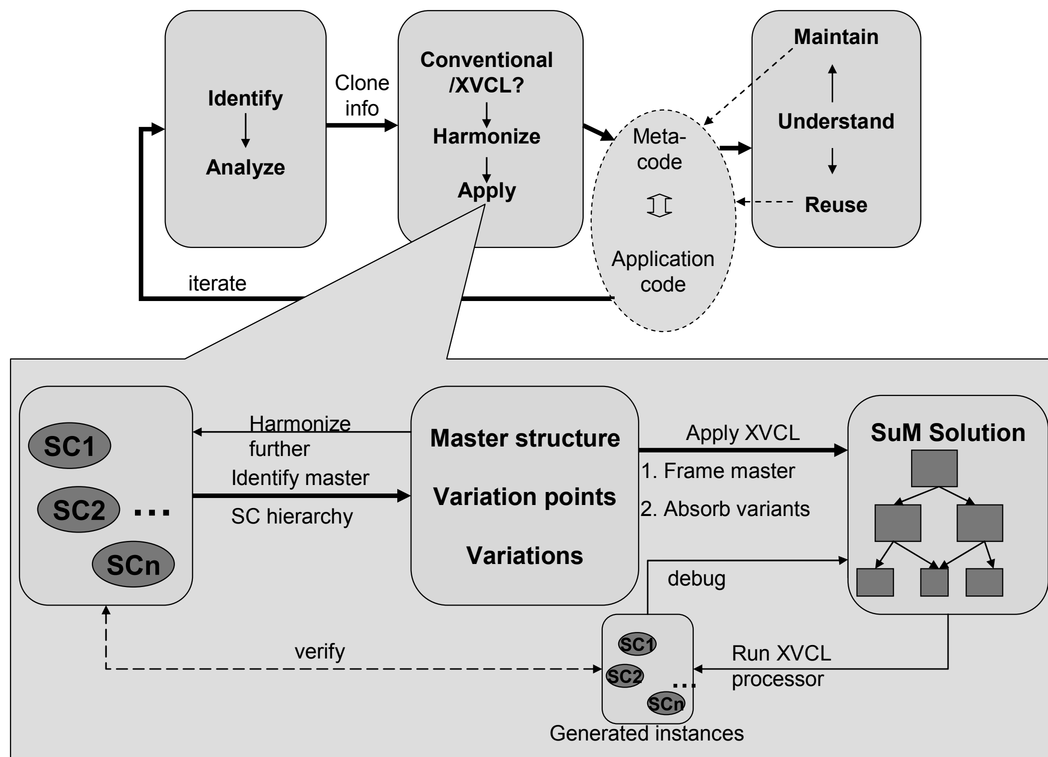


Figure 68. Unifying clones using SuM

Example from PCE, AB:

In both PCE and AB we used an incremental approach where a typical clone instance was chosen as the initial master. This initial master evolved as we progressively incorporated variations from the rest of the instances.

After implementing `Project` and `Product` modules in PCE, we chose `Project` module as initial master, and unified it with `Product` module. In AB we chose ACT as the initial master and used it to generate the other two. Deltas were found using the Araxis[39] diff tool.

[39] www.araxis.com

8.3.3. Bottom level – unifying simple clones

Clone unification is mostly a bottom-up process. Since the bottom layer of SC hierarchy contains only simple clones, we start by showing how XVCL unifies simple clones.

The general definition of a simple clone is a "contiguous code fragment of significant length and having significant similarity". Sometime this definition can be further constrained by the clause "representing a meaningful program entity". From an XVCL perspective, we interpret significant similarity as either "exactly the same" or "having no more than parametric variations". This is because XVCL can handle these two in a very straight forward manner. We also do not impose any constraints on the size of the code fragment, or what it represents, as this is immaterial to XVCL. Therefore, our simple clones come in two forms: exact simple clones and parametric simple clones. Exact simple clones, the simplest of structural clone types, are exact duplicates of code fragments (see Figure 69). Unifying exact simple clones can be as simple as converting the clone into an x-frame and `<adapt>`ing.

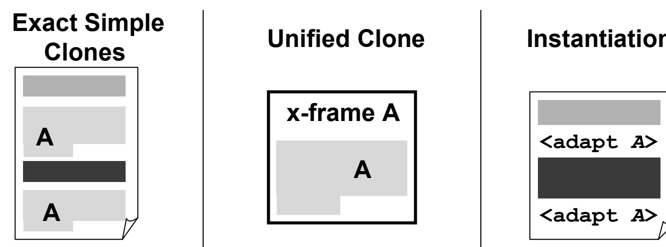


Figure 69. Unifying exact simple clones

In parametric simple clones, exactly matched fragments are interleaved with small (typically not longer than a few words) parametric variations, such as shown in Figure 70. Fragments `A` and `B` remain exactly the same, but `x` becomes `y` from instance 1 to 2 and disappears altogether in instance 3. Strictly speaking, instance 3 is not a parametric variation, but since XVCL can handle them the same way it handles parametric variations, we do not make any distinction between the two types. Parametric clones can be unified by converting the fragment to an x-frame and representing variation points with an XVCL variable (see Figure 70). Note that parametric variations acceptable to XVCL technique are more general than

variations which are typically unified using generics mechanisms such as C++ templates. For example, XVCL can handle "non-type parametric variations" described in section 4.3.

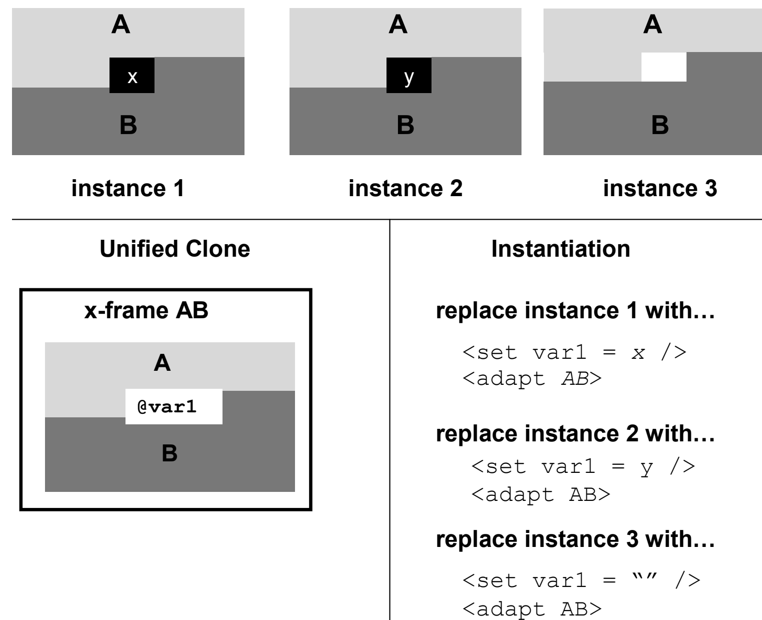


Figure 70. Unifying parametric simple clones

8.3.4. Building the hierarchy – unifying structural clones

As explained in section 8.3, a clone class (structural clones or otherwise) can be characterized in terms of a master structure that represents the inherent characteristics of the clone class. XVCL representation of the unified structural clone class consists of two parts: the unified (generic) structural clone, and the instantiation code:

- **The unified SC** - This is the reusable generic representations of the structural clone. This is essentially the XVCL representation of the master.
- **Instantiations** - This is the XVCL code that adapts the unified SC to a reuse context.

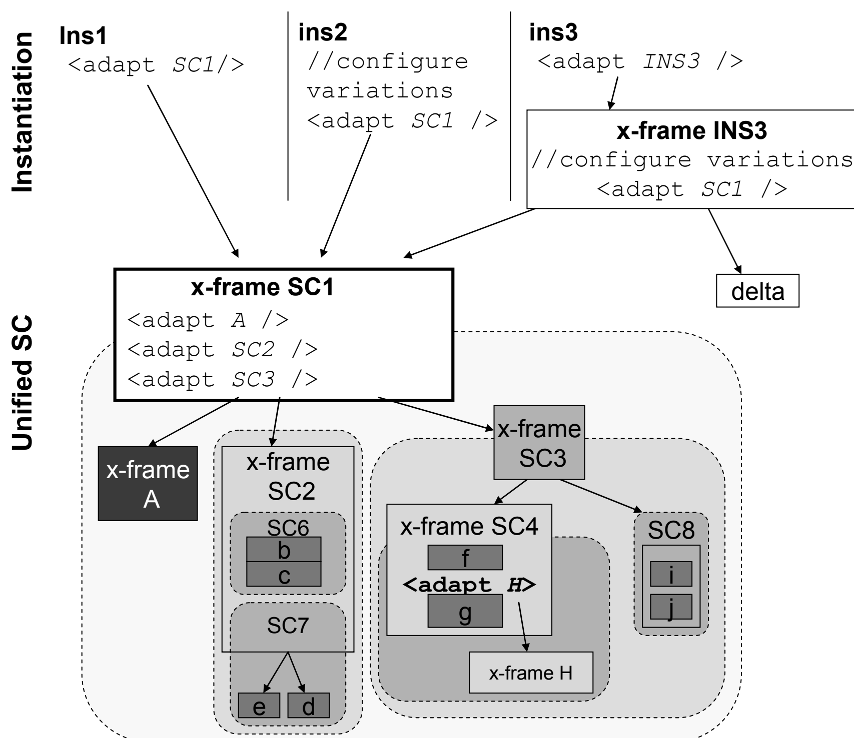


Figure 71. Unifying a structural clone using SuM

Figure 71 shows the partial SC hierarchy of the coarse-grained SC (named `SC1`) and 3 instantiations of it (named `Ins1`, `Ins2`, and `Ins3`). In this diagram SC boundaries are marked by hashed lines. `SC1` in turn consists of three lower level entities, two of which are structural clones themselves (`SC2` and `SC3`) and the other, a simple clone (`A`). `SC2` consists of two entities which are also structural clones (`SC6` and `SC7`). These two structural clones consist of simple clones (`b`,`c`) and (`d`,`e`) respectively. `SC3` has further layers of lower level clones which we do not explain in detail here. Simple clone visible in the SC hierarchy are `A`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, and `j`. As apparent from the above example, a unified SC may be an XVCL code fragment (e.g., `SC6`), a single x-frame (e.g., `SC8`), or an x-framework (e.g., `SC3`).

Instantiation may be as simple as `<adapt>`ing the unification frame, but usually instantiation is preceded by configuration of the variation points (e.g., by setting an XVCL parameter). When extensive customization is required, this configuration code may be encapsulated in additional x-frames (e.g., `Ins3`).

Some other points illustrated by this example:

- A simple clone can be a whole x-frame (e.g., `A`) or it can be a part of an x-frame (e.g., `b`)
- A single x-frame may include more than one structural clone (e.g., x-frame `SC2` contains `SC6` and a part of `SC7`)
- Entities can be in-lined in an x-frame (e.g., `f` and `g` are in-lined in `SC4`)

8.3.5. Unification root

The concept of *unification root* represents an important concept of SuM. We call the top level of the unified SC the unification root. This is where the elements of the structural clone are recognized as parts of the structural clone. Unification root is also the handle that lets us access the structural clone as a whole.

We illustrate the unification root using the example in Figure 71. When the structural clone is an x-framework, the top x-frame acts as the unification root (e.g., x-frame `SC1` is the unification root for the top level structural clone, x-frames `SC2` and x-frames `SC3` are the unification roots for structural clones in the layer below). The unification root either `<adapt>`s the entities of the structure (e.g. `SC1`), or the entities are in-lined within the unification root (e.g., `f` and `g` in `SC4`). If the structural clone is a single x-frame, then that x-frame itself is the unification root.

> *Unification root* is the handle that allows us to access a structural clone as a whole.

In-lined unification roots

Some structural clones are in-lined within x-frames (e.g., `SC6`). In such cases the code of the whole structural clone is considered the unification root. A structural clone can be an x-framework and still the unification root can be in-lined inside an x-frame (e.g., `SC7`). In-lining the unification root prevents us from reusing the structural clone and hence not recommended. For example, to reuse `SC3` we can just `<adapt>` its unification root (i.e., x-frame `SC3`), but

reusing `SC6` in that manner is not possible (first, we have to extract the in-lined unification root to an x-frame).

8.3.6. Aligning the solution along SC boundaries

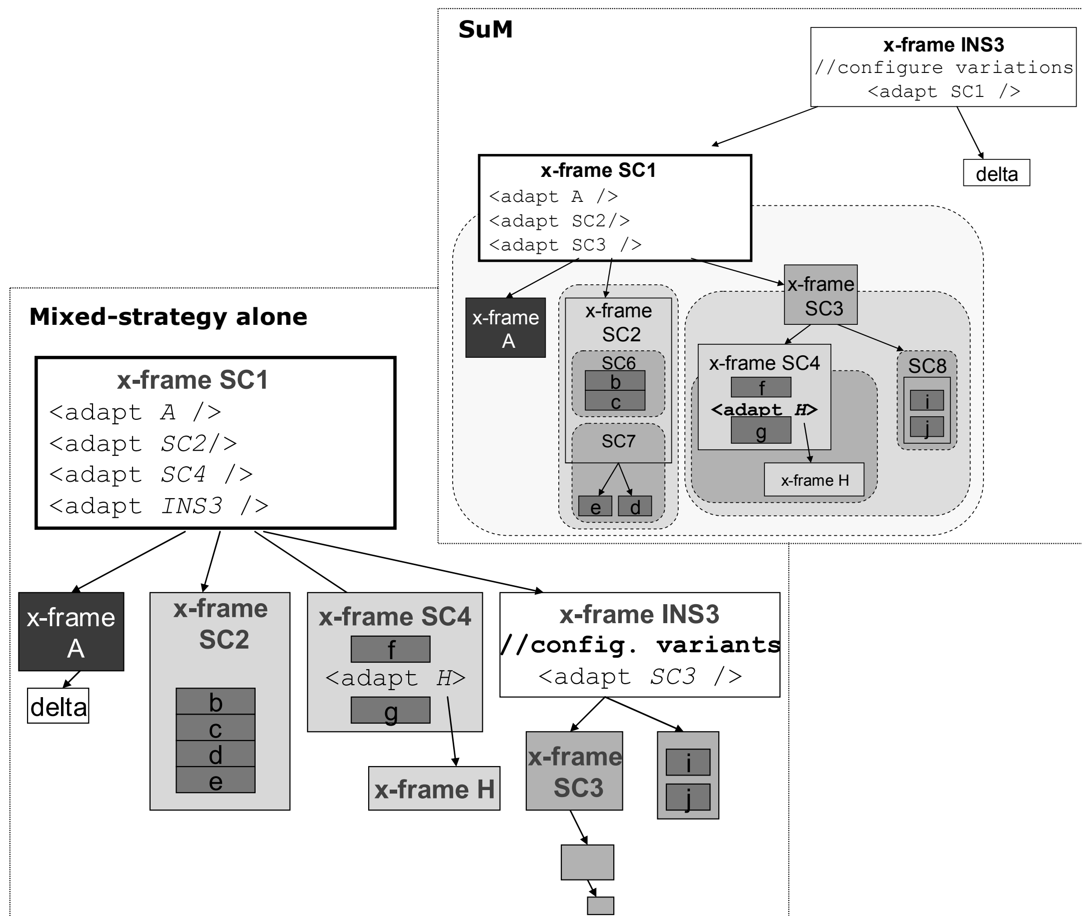


Figure 72. Unifying a structural clone with mixed-strategy alone

To illustrate what we mean by "aligning the solution along SC boundaries", we compare the SuM solution in Figure 71 with an alternative solution given in Figure 72 that uses mixed-strategy alone (i.e., not aligned along SC boundaries). Note how the hierarchy of the x-framework is different from one solution to the other. Although the level of clone unification is similar in both solutions, our opinion is that, assuming the SC harvesting was done correctly, SuM solution is better structured than the pure mixed-strategy solution. For example, note how the generic code in the mixed-strategy solution is polluted with instance-

specific code (e.g., x-frame `delta`), whereas those were kept outside of the generic code in the SuM solution.

8.3.7. Improving the quality of SC harvesting

There are number of ways we can improve the quality of the harvested SC hierarchy, and ultimately the quality of the SuM solution:

- By closely matching the structural clones to cloned concepts in problem/solution domains. This minimizes introduction of additional conceptual entities, thus making the solution easier to understand.
- By judiciously controlling the height and width of the SC hierarchy. Both too many levels, and too many entities within one level, should be avoided. Again, this makes the solutions easier to understand.
- By correctly identifying the variation based on the available variants. Identifying the correct generic representation (and the variations) based on a limited number of variants is a major challenge in SC harvesting. We address this issue later in this chapter, in section 8.7.
- By standardizing the unification method for common structural clone types. This helps to improve the SuM solutions, the same way design patterns (e.g., [GHJ97]) help to improve program design when faced with common design problems. Section 8.7 addresses this aspect as well.

8.4. Post-unification activities

8.4.1. Understanding mixed-strategy solutions

We have the following to aid us in understanding a mixed-strategy solution.

- Application code and related documents (e.g., UML models) describe the run time structure of the application – this is same whether we apply mixed-strategy or not
- Meta-code and related documents (e.g., meta-component architecture diagram) describe the construction time structure of the application. Meta-code explicates some extra information that is otherwise only implicit in the application code.

A simplistic approach is to look at application code and related documents to understand how the application works, and then look at XVCL docs and code to understand how it is constructed in terms of similarities and variations. A combination of the two can bring more optimal result, as illustrated by the below example.

Example from PCE, AB:

Documentation of PCE XVCL code shows that `Project` module is used as the master. Therefore, understanding the application code of `Project` module is a good starting point to understand the PCEms. Then we go back to meta-code to see the differences between `Project` module and other modules. This way we can understand the whole application faster (no need to go through the whole application code), and more accurately (minute variations between modules can be overlooked when going through the application code).

Similar strategy can be applied when understanding the mixed-strategy solution for AB, where `ACT` was used as the master.

8.4.2. Maintenance of mixed-strategy solutions

Maintenance of a mixed-strategy application requires keeping the application code and meta-code in sync. There are two approaches to achieve this.

- **Forward-propagation approach**: modify meta-code and propagate to application code using XVCL processor

  - o Advantages: less number of places to modify
  - o Disadvantages: meta-code is more difficult to modify due to XVCL tags

- **Back-propagation approach**: modify application code and migrate the changes to meta-code. This could be manual or semi-automated using a back-propagation tool.
  - o Advantages: editing/debugging application code is easy to do using familiar tools. Modified code can be deployed before the migration to meta-code (faster time-to-market)
  - o Disadvantages: extra effort needed for back propagation

It is theoretically possible to maintain the application code from the meta-code only, i.e., use forward-propagation only. However, our experience suggests that a mixture of the two brings optimal results. The proper mix is determined by considering the impact of modification (with respect to clones) and the ease of modification.

Impact of modification

This is a measure of how the modification propagates via clones. It is directly related to update anomaly risk. Given that a clone class of *n* members, of which *m* member are affected by the modification, we consider four categories. (For simplicity, we ignore modification that affects multiple clone classes)

- *all* (m=n): modification needs to be repeated in all members of a clone class
- *some* (n>m>1): modification applies to some (more than one), but not all, members of a clone class.
- *one* (m=1): modification applies to only one member of the clone class
- *none* (m=0): modification is done to a non-cloned part of the application. Generally XVCL can be selectively applied to the cloned part of an application. Hence this

category is rare in practice. However, when a file contains both cloned and non-cloned parts, the whole file needs to be represented as an x-frame (or an x-framework). In such cases non-cloned parts can creep into meta-code.

Ease of modification

This is a measure of the number of steps involved. We consider two categories.

- *one-step*: the CMP knows exactly what the required modification is. No debugging is expected. Only minimal testing is needed. Usually these are small changes that are contained to a few locations in a small locality. E.g., changing the value of a variable.
- *multi-step*: the modification requires exploratory, trial-and-error programming. A few rounds of modifications, testing and debugging may be required. Exact modification is uncertain at the beginning. E.g., performance tuning, fixing a bug of unknown origins.

Table 10. Typical approach for modification in different scenarios

		mixed-strategy solution (M)	**Conventional application (C)**	**Effort**
one-step	**all**	modify meta-code, forward-propagate to *m* clones	modify application code, find all affected clones, repeat modification	M<<<C
	some			
	one	modify meta-code, forward-propagate	modify application code	M≈C
	none			
multi-step	**all**	experiment with one clone in application code, backward-propagate, forward-propagate to remaining *m-1* clones	experiment with one clone, find all affected clones, repeat modification	M<<C
	some			
	one	experiment with the affected code in application code, backward-propagate	experiment with the affected code	M>C
	none			

Table 10 summarizes the steps involved in different kinds of modifications in a mixed-strategy application, side-by-side with the steps involved in doing a similar change in a conventional application. According to Table 10, category pairs (all,some) and (one,none) appear to be same. But in practice the modification to meta-code in each case requires different techniques. When comparing mixed-strategy applications to conventional applications, mixed-strategy application performs worse than a conventional application for multi-step changes that affect one/none of the clones (shaded). In all other cases, the effort is either almost the same or the effort for mixed-strategy application is considerably less.

8.4.3. Reuse within mixed-strategy applications

The situation considered here is that there is a need for another variant instance of an already unified clone class. This is similar to "potential clone" situation (discussed in section 8.1), except that it involves already unified clone class.

The typical approach is to configure existing variation points to match the new variant as closely as possible, and generate the code. Final adjustments (to arrive at the exact new variant) are done according to the previous section (as if it's the maintenance of an existing variant). In essence this is both generative and compensatory clone management in one shot.

Example from PCE:

In PCEms, we used unified module to create new modules.

8.5. Applying SuM to Adventure Builder

Now we resume our study of Adventure Builder (from where we left off in section 7.2.4) to show the application of some of the techniques we described so far in this chapter. Figure 73 shows the partial SC hierarchy we pieced together in section 7.2.4 for the supplier subsystem in Adventure Builder.

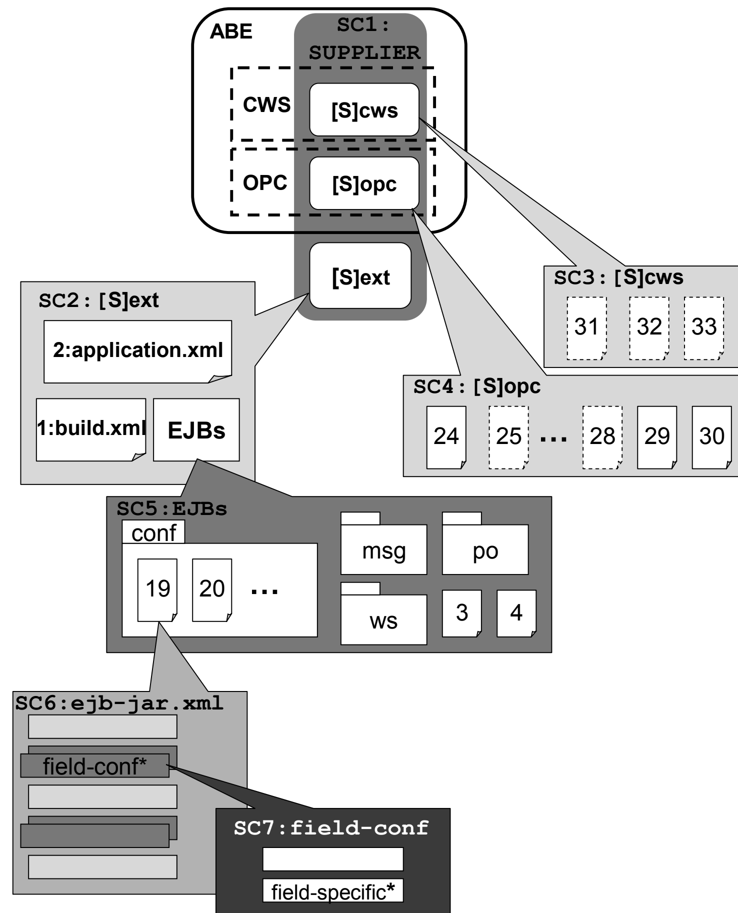


Figure 73. Partial SC hierarchy for Adventure Builder

Figure 74 illustrates the unification of `SC2:[S]ext` as an example. X-frame `[S]ext` is the unification root of this unified SC.

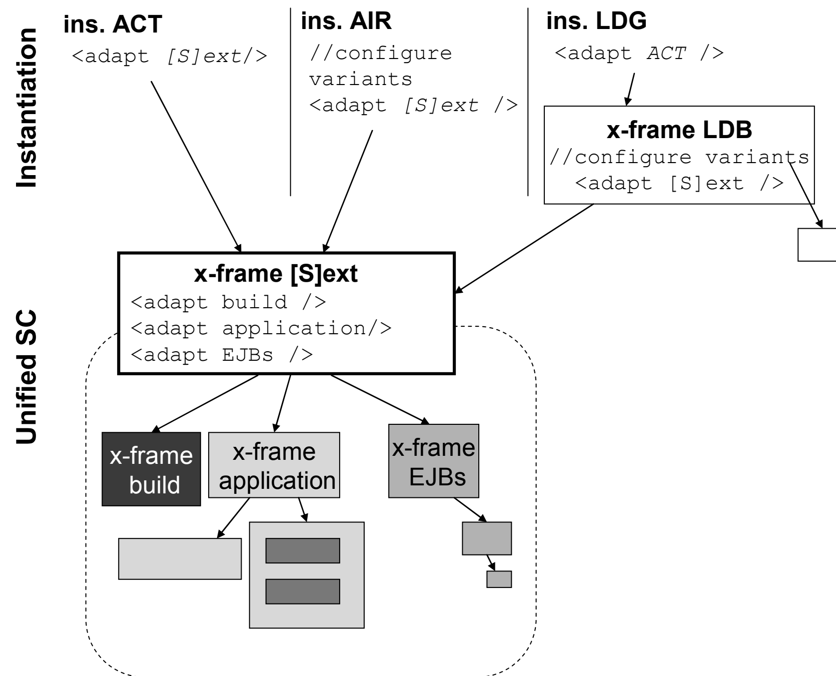


Figure 74. Unification of structural clone [S]ext

Figure 75 shows the partial x-framework for SC1:SUPPLIER and the SC hierarchy that lies beneath it. Unification frame SUPPLIER and the x-framework below that represent what is similar between the three instances. X-frames LDG, AIR, and ACT are the customization code for the three instances. With this arrangement of customization code, it is easy to see what is different between the three supplier instances. The area marked as 'extra' denotes x-frames that are injected to the unified SC at variation points. The area marked as 'common fragments' denotes small code fragments that are used in more than one structural clone (e.g., 'common imports' are used in several files, including extra x-frames).

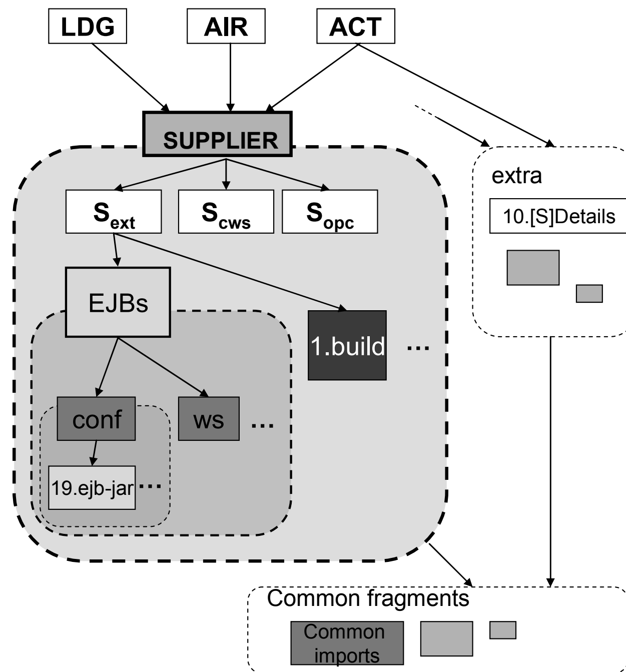


Figure 75. Partial x-framework for SUPPLIER

8.6. Conquering the diversity of structural clones

Generally, it is useful to know in advance which unification strategy is suitable for a given structural clone type. Unfortunately, and by necessity, the definition of structural clones is very wide. Hence such a categorization appears to be impractical at first, due to the vastness of the problem space. To overcome this problem we try to shrink the size of the problem space by identifying a small number of basic types that represents the whole spectrum of structural clones. As the starting point for this quest, this section identifies the basic types of entities, and structures that constitute structural clones. For each basic type in these three dimensions, we describe the equivalent representation in SuM.

8.6.1. Diversity in structural clones

The CMP has to consider many factors when selecting the best way to unify a clone using a conventional technique. Some of those factors are:

- Type of entities structural clone consists of

  - Implementation technology
  - Granularity
- Type of entity structural clone is being abstracted to, if any
- Physical characteristics of the structure
- Available features of the unification mechanism
- Nature of variations between instances
- Nature of anticipated variations in future instances

For instance, consider the two different types of structural clones in Figure 76 (only one member of each class is shown). When using clone unifications offered by the implementation technologies involved in `SC1` and `SC2`, the suitable unification of `SC1` will be different from unification of `SC2`. Further, it will depend on the actual contents of the structural clones. Selecting a suitable unification strategy requires very high level of expertise of the implementation technologies involved, and generally cannot be made without knowing the contents of the actual clones. Sometimes such a unification is not possible at all, due to the mixing of different technologies (such as in Figure 76). This diversity adds to the complexities of managing structural clones. Therefore, it is worthwhile to try to shrink the problem/domain space by abstracting over this diversity.

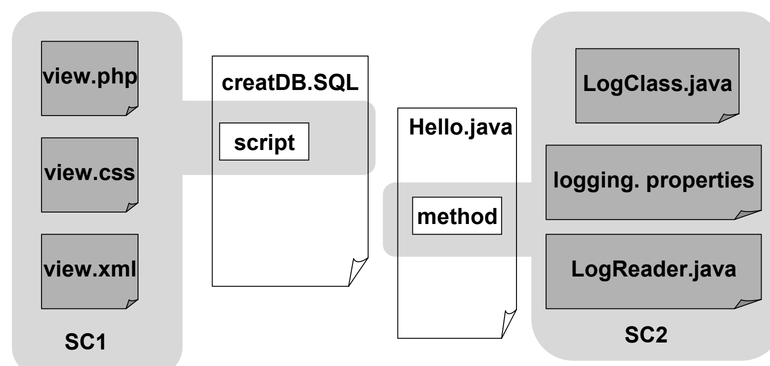


Figure 76. Two different structural clones

XVCL, as explained previously, can be applied independently of the semantics of actual contents of the structural clone. Therefore, selecting XVCL as the unification mechanisms considerably shrinks the problem space of structural clones. For example, since XVCL does not distinguish between PHP, CSS, XML, Java, and properties files or between SQL scripts and a Java method, the two different structural clones in Figure 76 can be considered as of same structure (see Figure 77): both are structural clones consisting of 3 files and one code fragment.

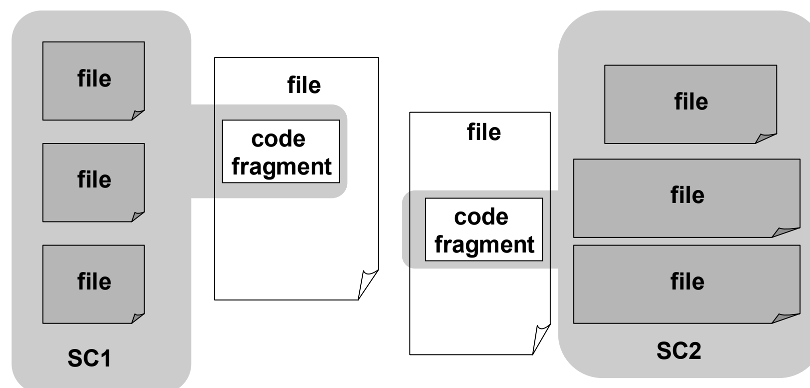


Figure 77. SC1 and SC2 simplified into two similar structural clones

8.6.2. Basic entity types

Table 11. Basic entity types

Basic entity type	Definition	May consists of …	Examples of actual entities
fragment(f)	contiguous code fragment	(inner) fragments	a function, a Java inner class
file(F)	a physical file	fragments	a Java source file, an HTML page
directory(D)	a physical folder	files, (sub)directories	a directory in a Java package tree
bundle(B)	arbitrary collection of entities	fragments, files, directories, and (sub)bundles,	a module, a component, a hyperlinked web page sequence

We define four basic entity types (Table 11). Any actual entity type can be mapped to one of the four basic entity types: fragment (f), file (F), directory (D), and bundle (B). Note that

bundle is a very general "catch all" entity type. Anything that cannot be mapped to the other basic entity types is mapped to a bundle.

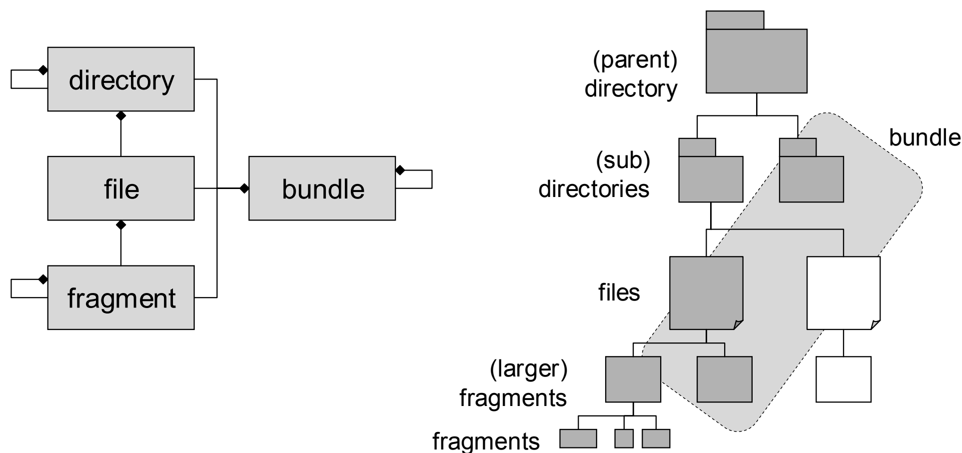


Figure 78. Composition model for entity types

The composition model in Figure 78 show how fragments, files, and directories form a clean hierarchical composition model where granularity is fragment<file<directory. Each of these entities may be decomposed into a structure of entities of the same type (except files; files cannot be nested) or entities of granularity one level below. Bundles, on the other hand, crosscut this hierarchy. The granularity of a bundle depends on the constituent elements.

An entity can be represented by an XVCL code fragment (for type f and B), an x-frame (for types f, F and B), or an x-framework (for types f, F, D, B).

8.6.3. Basic structure types

We know that the unified SC is in fact the XVCL representation of the master (master is the generic representation of the structural clone). Therefore, it is important to know how to represent different kinds of structures in XVCL. We start by categorizing the structure types (shown in

Table 12) based on the following criteria:

- Entity type - This can be f=fragment, F=file, D=directory, B=bundle, or h=heterogeneous (any mix of f, F, D and B)
- Entity type the structure is being abstracted to.
- Physical characteristics - The structure can be crosscutting files or directories. In some cases, the physical order in which fragments appear also matters.

Table 12. Basic structure types

Entity type	Abstracted to…	Crosscutting…	Example	structure type
f	larger f	-	A sequence of code fragments abstracted to a function	1
	F	-	A collection of functions abstracted to a class	2
	B	F	functions related to an aspect spread across a number of files	3
F	D	-	a set of files inside a directory	4
	B	D	database scripts spread across directories	5
D	parent D	-	a set of sub-directories inside the same parent directory	6
	B	D	The two Java packages abc.xxx.upload and abc.yyy.download	7
h	B	A bundle is crosscutting by default	A module consisting of directory for Server pages, config. file inside a common directory, and part of a SQL script	8
B	bigger B		a collection of modules belonging to a application	9

According to Table 12, there are 9 basic structure types. Now we shall explain how these structure types can be captured using SuM.

SuM technique for capturing non-crosscutting structures

Most non-crosscutting structures can be easily captured into XVCL, in the following manner.

Create an x-frame to be the unification root, and `<adapt>` the unification root of all its entities. If the entity is a simple clone, either we `<adapt>` the x-frame that contains it, or in-line the XVCL representation of it. This mechanism applies to the structure types 1, 2, 4 and 6 in Table 12 (i.e., non-crosscutting structures).

Note that similar to in-lining the unification root, in-lining simple clones is a shortcut that reduces its reusability. Both are used to reduce fragmentation of the XVCL solution, when no immediate reuse is foreseen.

> In-lining unification roots or simple clones prevents excessive fragmentation, but reduces reusability.

SuM technique for capturing crosscutting structures

Crosscutting structure types 5 and 7 do not contain any fragments. Therefore the only additional thing we have to do on account of their crosscutting nature is to make sure files and directories generated by the SuM solution go to the correct physical location. This can be easily done using the XVCL variables `outdir` and `outfile`. Hence capturing these structure types is almost as easy as their non-crosscutting counterparts.

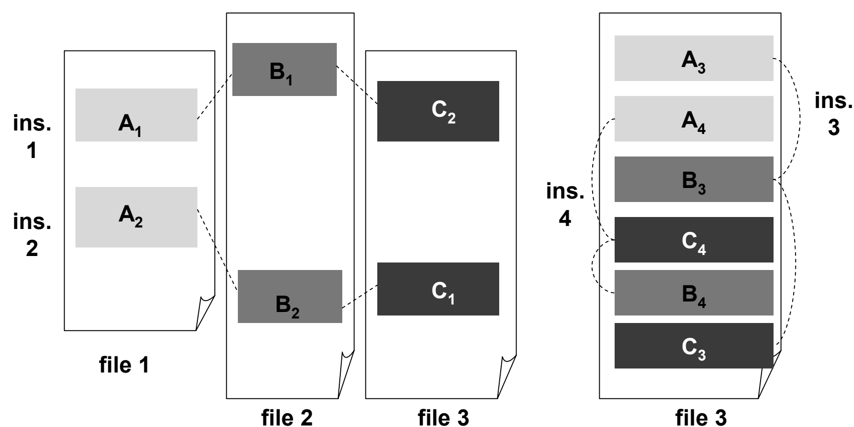


Figure 79. Fragment structures that crosscut files

Crosscutting structures that involves fragments are more difficult to capture into SuM solution, as we shall explain next. Let us take structures of type 3 first. These are similar to aspects in Aspect Oriented Programming. Fragments of these structures can be distributed among number of files, as illustrated by Figure 79.

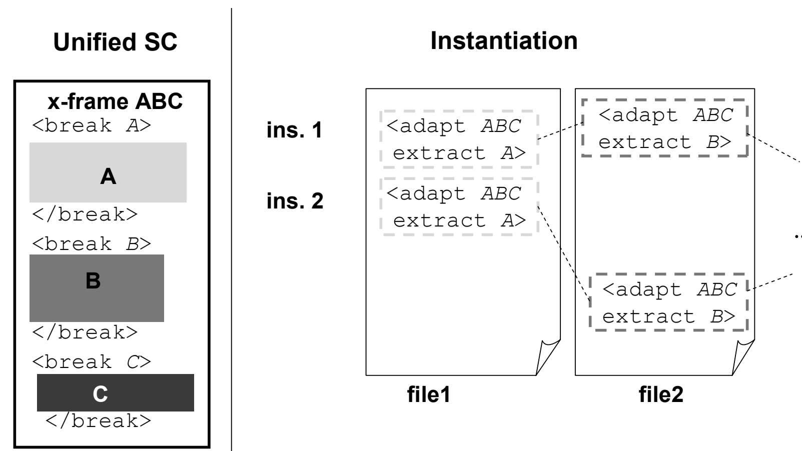


Figure 80. Unifying fragment structures that crosscut files

The strategy to capture (and instantiate) this type of structures is shown in Figure 80. We unify all fragments into one x-frame (e.g., x-frame `ABC`) and mark each fragment with `<break>` tags. During instantiation, we `<adapt>` the x-frame and `<extract>`[40] the appropriate fragment (e.g., `file1`, `file2`).

Structure types 8 and 9 involve bundles of heterogeneous entities, which may be bundles themselves. These can be captured into XVCL using a combination of techniques we used for the rest of the structure types.

[40] <extract> command is not currently supported by XVCL, but can be emulated using existing commands

8.7. Basic SuM unification schemes

In practice, CMP is given a number of structural clone instances, from which she has to arrive at the best SuM solution. The previous section explained how different structures may be represented in SuM, targeting the "Frame master" activity in Figure 81.

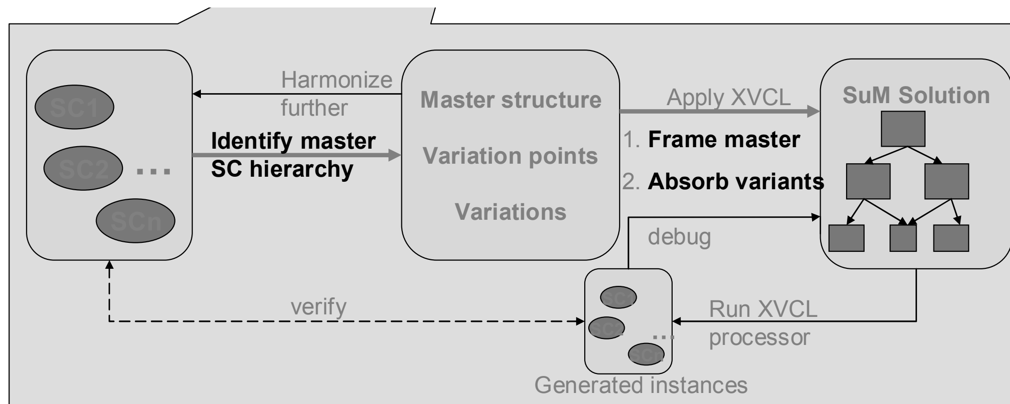


Figure 81. SuM activities described in this chapter

In this section we present materials that help in "Identify master SC hierarchy" and "Absorb variants" activities in Figure 81. The basis of our presentation is the basic types of variations found in structural clones, together with how to unify them using SuM (we call these 'basic SuM unification schemes'). The idea is that matching the actual variations found in the structural clone variants against these basic types of variations, CMP can get an idea about what the master SC should be. For example let us assume that a function is missing in all but one in a set of cloned source files. According to our definition of basic variations, this is likely to be an "extra entity" (to be described in section 8.7.1) which should not be made a part of the master. By providing a recommended solution to each variation we promote standardization of the SuM solutions. Further, we expect these schemes to promote a common vocabulary in identifying variation types, and provide a basis for developing best practices and tool support for SuM.

It should be noted that these solutions are provided as a guide only. They do not dictate the only way to unify a given structural clone type. Further optimizations may be possible, depending on the real context of the structural clone.

8.7.1. Extra entity

Definition: An extra entity is not considered an integral part of the structural clone; Rather, it is an external injection to some (typically a minority) of the variants.

Example: In Figure 82 we show SCC of fragment structures, where entity `D` is an extra entity present only in `instance 3`.

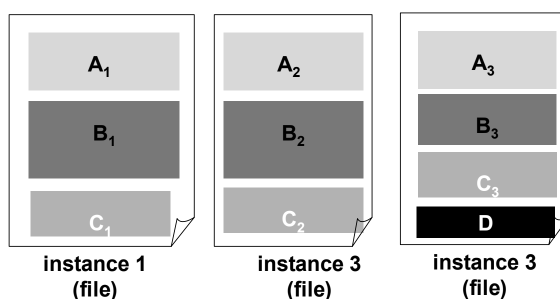


Figure 82. An example of an extra entity

Solution: Extra entities are not to be made a part of the master or the unified SC. We first unify all common entities (but not extra entities) into one x-frame or an x-framework (see Figure 83). We mark the place where extra fragments should appear with a `<break>`, such as shown in `line1`.

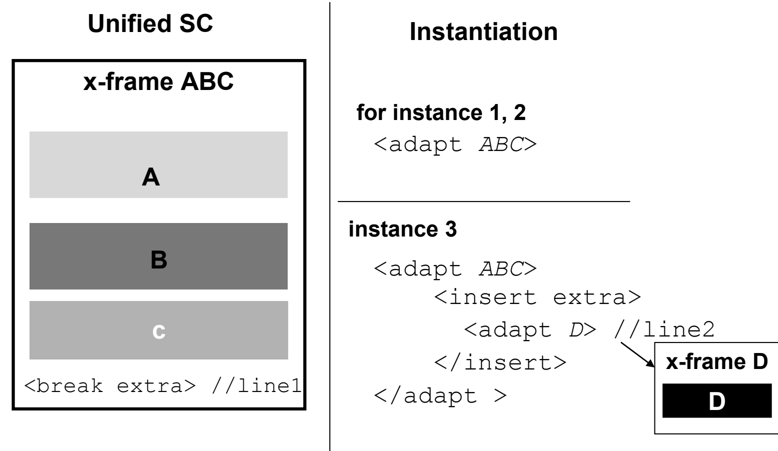


Figure 83. Solution for extra entity

Comments: Note that the exact place where the `<break>` appears does not matter unless the structure is a fragment structure. Extra fragment is inserted at the point of adaptation (see line 2), and never referenced from the unified SC itself (thus, emphasizing that it is 'extra' – not part of the structural clone).

8.7.2. Optional entity

Definition: An optional entity is an integral part of the structural clone, which may be omitted in some variants.

Example: Figure 84 shows an SCC of file structures (abstracted to a directory), where the entity `B` is missing from one instance (i.e., optional).

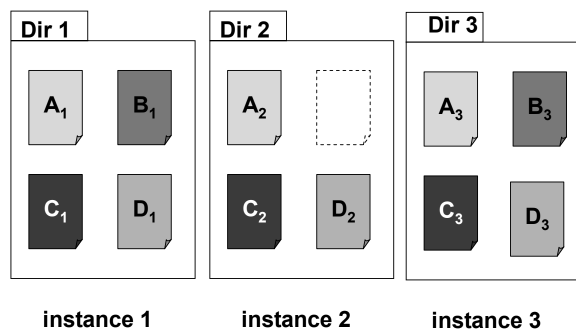


Figure 84. An example of an optional entity

Solution: We `<adapt>` the unification roots of all common/optional entities within one x-frame (e.g., x-frame `ABCD` in Figure 85). Optional entity is marked using `<break>` commands (see `line2`) so that it can be `<remove>`d at the point of adaptation (see `line1`)[41].

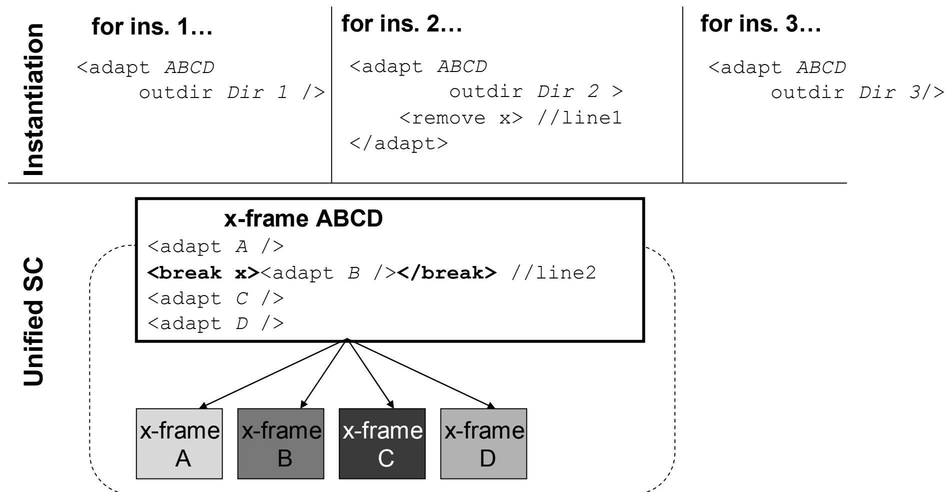


Figure 85. Solution for optional entity

8.7.3. Parametric entity

Definition: A parametric entity is an empty slot in the structural clone, expected to be filled for each variant with an instance-specific entity.

Example: Figure 86 shows an SCC of file structures (abstracted to a directory), where one entity varies parametrically from instance to instance.

[41] <remove> command is currently not supported, but can be emulated using an empty <insert> command

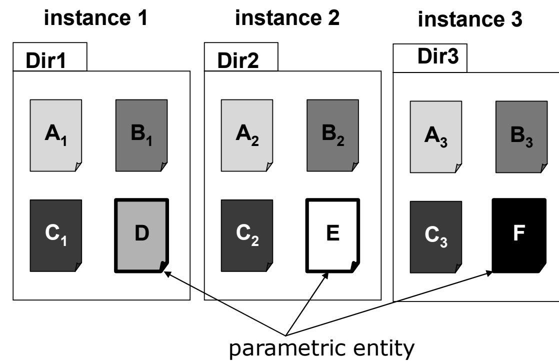


Figure 86. An example of a parametric entity

Solution: We <adapt> the unification roots of all common entities within one x-frame (e.g., x-frame ABCX in Figure 87). The slot for the parametric entity is marked using a <break> tag (see line 1). We can insert the appropriate entity to the slot at the point of adaptation, as shown for the instantiations 1 and 2. Alternatively, we can generate all variants using <while> loop controlled by two multi-valued variables (shown at top right of Figure 87).

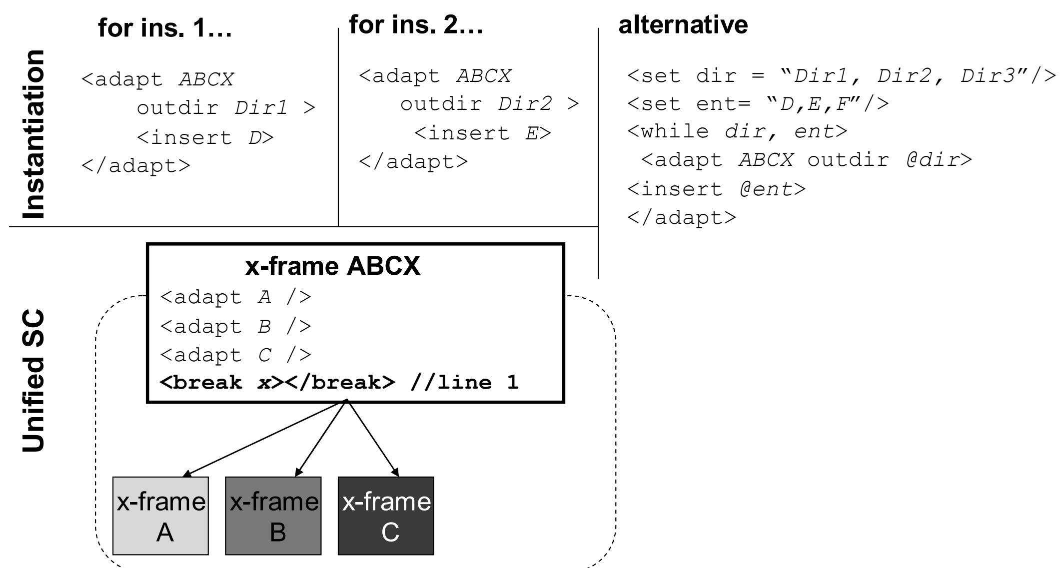


Figure 87. Solution to the parametric entity

8.7.4. Alternative entity

Definition: An alternative entity is a slot in the structural clone, expected to be filled from a predetermined set of entities. All alternative entities are considered as integral parts of the structural clone.

Example: In the three instances of fragment structures shown in Figure 88, entities `C` and `D` are alternatives competing for the same spot.

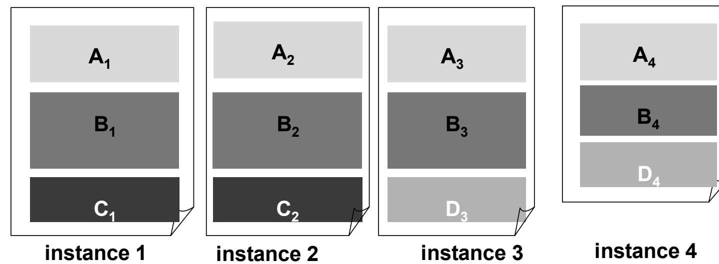


Figure 88. An example of an alternative entity

Solution: We use the XVCL `<select>` tag to handle this variation. As shown in Figure 89, we mark the two alternatives using `<option>` tags and select the appropriate option using the control variable of the `<select>` tag, at the point of adaptation.

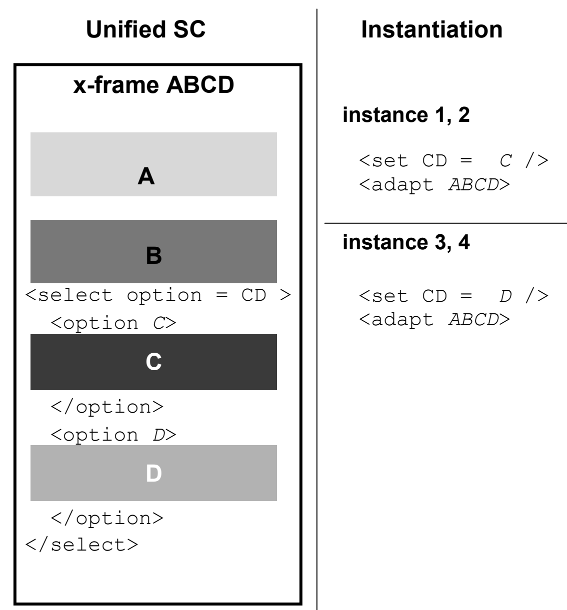


Figure 89. Solution to the alternative entity

8.7.5. Repetitive entity

Definition: A repetitive entity repeats (possibly with slight variations) an instance-specific number of times for each variant. A repetitive entity is an integral part of the structural clone.

Example: In Figure 90, the entity `B` is repeated varying number of times, including zero times.

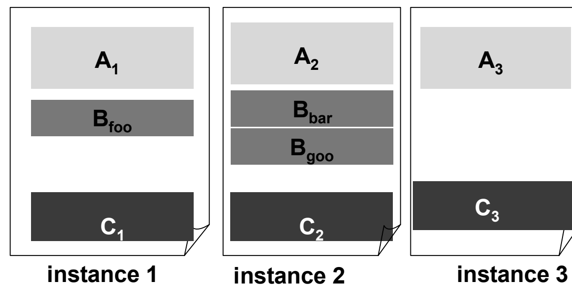


Figure 90. An example of a repetitive entity

Solution: The number of repetitions can be controlled by a multi-valued variable and a `<while>` loop, as shown in Figure 91.

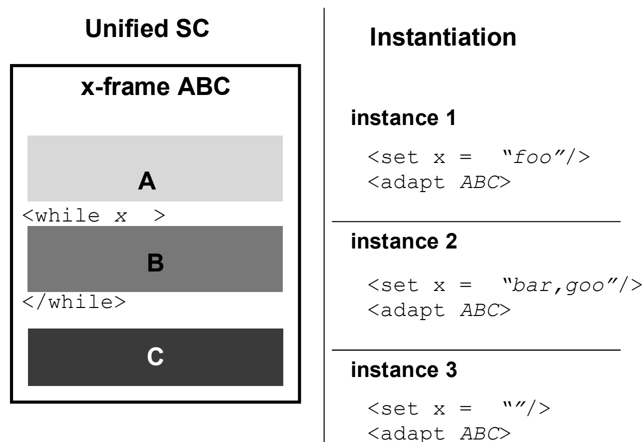


Figure 91. Solution for repetitive entity

8.7.6. Replaceable entity

Definition: A replaceable entity is an integral part of the structural clone, which may be replaced by another external entity for some variants.

Example: Figure 92 shows an SCC of file structures, where the entity `B` is replaced with `R` in `instance 2`.

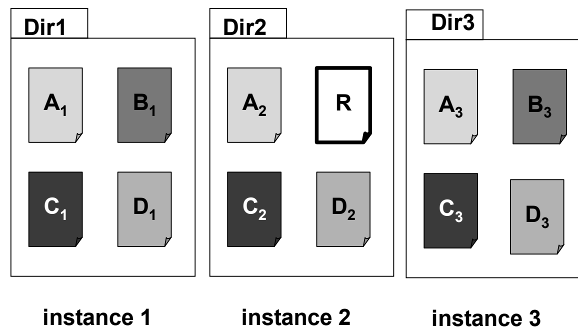


Figure 92. An example of a replaceable entity

Solution: We <adapt> the unification roots of all common/replaceable entities within one x-frame (e.g., x-frame ABCD in Figure 93). Replaceable entity is marked using <break> commands (see line 2) so that it can be replaced at the point of adaptation, using the <insert> command (see line 1).

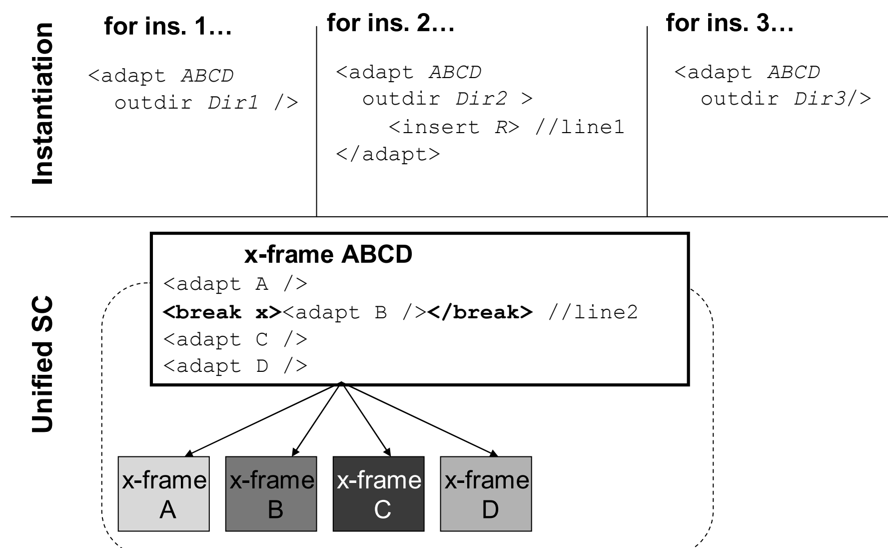


Figure 93. Solution for replaceable entity

Comments: Replaceable entity is very similar to optional entity. The difference is that it is mandatory to fill the slot in the case of replaceable entity. One can think of a hybrid of the two where an entity is both optional and replaceable. The solution for such a situation follows the same pattern as given above.

8.7.7. Reordered entity

Definition: When there is an order among the entities, reordered entity is a variation in the order of entities.

Example: Figure 94 shows two code fragment structures where the order of the entities differs.

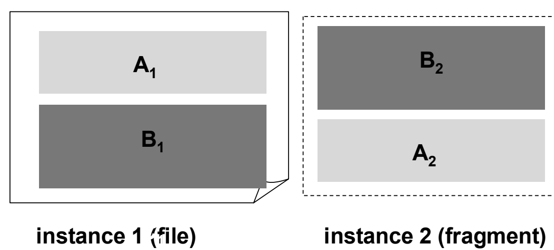


Figure 94. Examples of a reordered entity

Solution: As shown in Figure 95, we unify all fragments inside one x-frame (e.g., x-frame `AB_base`). Write another x-frame that reorders the fragments based on a control variable (e.g., x-frame `AB`).

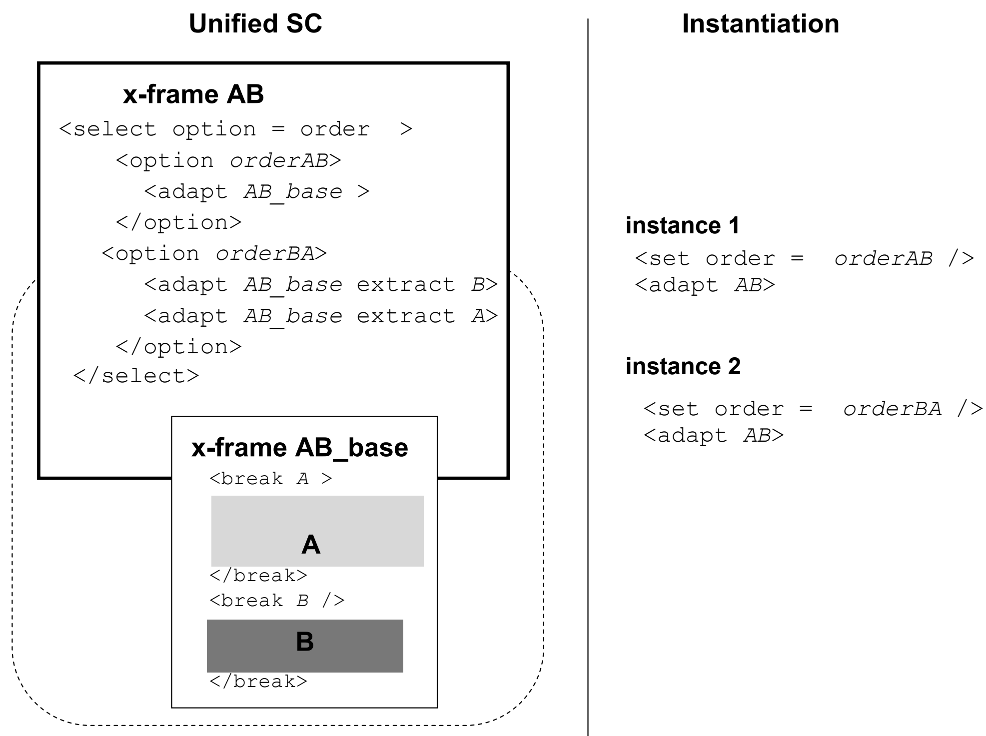


Figure 95. Solution for the reordered entity

Comments: This variation is only applicable to fragment structures. If the fragments are dispersed instead of occurring together as in the above example, then we can use a technique similar to that given for crosscutting fragment structures, in section 8.6.3.

8.7.8. Using basic SuM schemes

Although XVCL has only a small number of language constructs, usually there are multiple ways to unify a given set of clones. Therefore, one is always faced with the difficult task of choosing the best solution for a given clone situation. In SuM we break this task into two steps:

Step 1: Decide what the master is and what the variations are

In this step we extract what the master is by matching the variations (found in existing clone instances and anticipated variants) to the basic structural variations described in the basic SuM schemes. This step is aided by the information contained in basic SuM schemes, but the final decision is based on the knowledge of the application domain. For example, Figure 96 shows three instances of a structural clone (consisting of code fragments) where `manage_fuel` function varies from instance to instance, showing the characteristics of a parametric entity. However, if the concept of vehicles captured by this clone can have only three types of `manage_fuel` functions (i.e., `manage_fuelX`, `manage_fuelY`, and `manage_fuelZ`), then it becomes an alternative entity. Whether it is a parametric entity or an alternative entity directly affects the shape of master (parametric entities become empty slots in master, while all alternative entities are considered as a part of the master), and such a decision needs to be based on the knowledge of the concept being cloned.

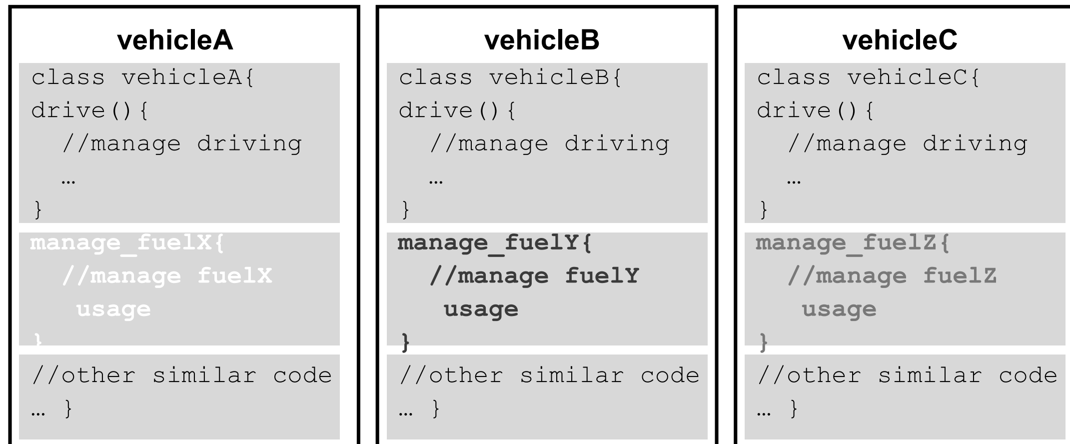


Figure 96. Alternative entities or parametric entities?

Step 2: Decide how to unify the clone

Once the master and the variations are known, we can easily unify the clone using techniques recommended by basic SuM schemes.

Other points to note when using basic SuM schemes

Unifying lower level variations: It should be noted that corresponding entities between instances in the examples (given in sections 8.7.1 to 8.7.7) are not exact clones. This is the general case for any structural clone. We should treat each one of them as lower-level structural clones (which also may be simple clones).

Overlapping variations: It is quite possible for these basic variations to overlap. An example is replaceable *and* optional entities (given under the comments in section 8.7.6). Such overlaps can be handled by a combination of solutions recommended for the individual variation types concerned.

8.7.9. Benefits of Basic SuM schemes

Following are the main benefits of having SuM schemes such as the basic SuM schemes described above.

- **They provide a standard vocabulary** – Terms such as 'alternative entity' provide a standard and succinct way to describe things we frequently encounter when applying SuM. This helps in communication and documentation of mixed-strategy solutions.
- **They help to standardize mixed-strategy solutions** – By guiding the CMP towards a recommended best-fit SuM solution, these schemes promotes standardization of mixed-strategy solutions. This in turn adds predictability to SuM solutions, an important quality that eases the maintenance of any software.
- **They may provide the basis for automation** – Once the variation type is identified using the domain knowledge, applying the appropriate XVCL syntax is fairly well defined, creating a possibility for it to be automated in the future.
- **They play the role of best practices** – SuM schemes play the role of SuM design best practices, similar to role played by analysis patterns[Fow96] and design patterns[GHJ97] in promoting good design in conventional software design.
- **They help to link mixed-strategy solutions with domain concepts** – As explained in the previous section, SuM schemes allow the domain knowledge to shape the SuM solution, so that the unified SC closely matches the domain concept that is being cloned. This promotes the future maintainability and reusability of the solution.

8.7.10. Basic SuM schemes in Adventure Builder

Now let us look at some examples of basic SuM schemes in action, used in applying SuM to Adventure Builder. In the structural clone `ws` (an entity of `SC5:EJBs, see` Figure 73), `[S]Details.java` appears in only one instance; let us assume it to be an extra entity. The entity `[S]order.java` on the other hand appears in all the three instances, but its content is totally different from one instance to the other; let us assume it to be a parametric entity. This situation is illustrated in the top half of Figure 97. Unification of `ws` (see bottom half of

Figure 97) illustrates how the above mentioned extra entities and parametric entities are handled in AB.

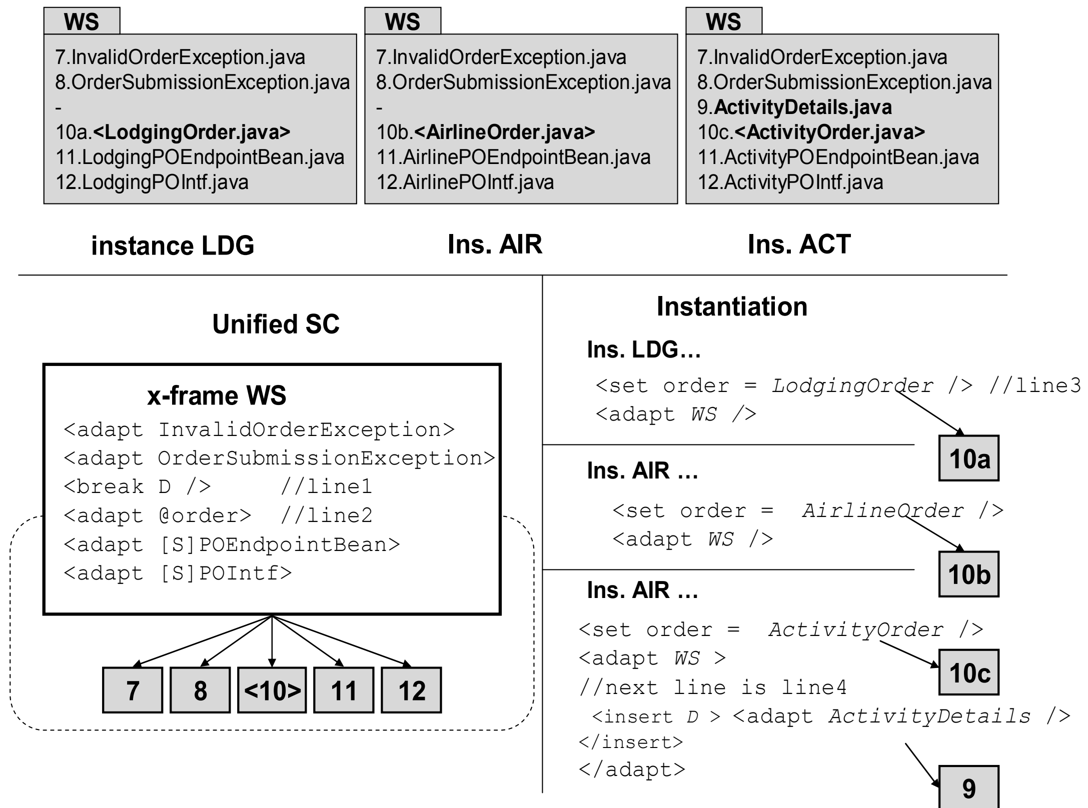


Figure 97. Handling extra entities and parametric entities in AB

Let us now assume that `[S]Details.java` appears in two instances instead of one. Let us also assume file 10 is `OrderA.java` in two instances and `OrderB.java` in the other instances. In such a situation (shown in Figure 98) we may consider `[S]Details.java` as an optional entity, and `OrderA.java` and `OrderB.java` as alternative entities.

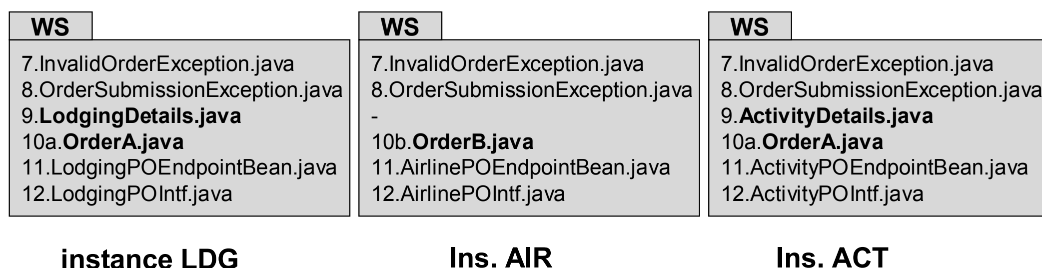


Figure 98. Optional entities and alternative entities in AB

Finally, we show how an example of a repetitive entity, using the example of SC6:ejb-jar.xml. Figure 98 shows a simplified version of ejb-jar.xml and its unification. Fragments A and B are repeated for each bean configured in ejb-jar.xml file. Instance LDG configures one bean, while instance ACT configures two beans. Figure 99 shows the XVCL version of this variation.

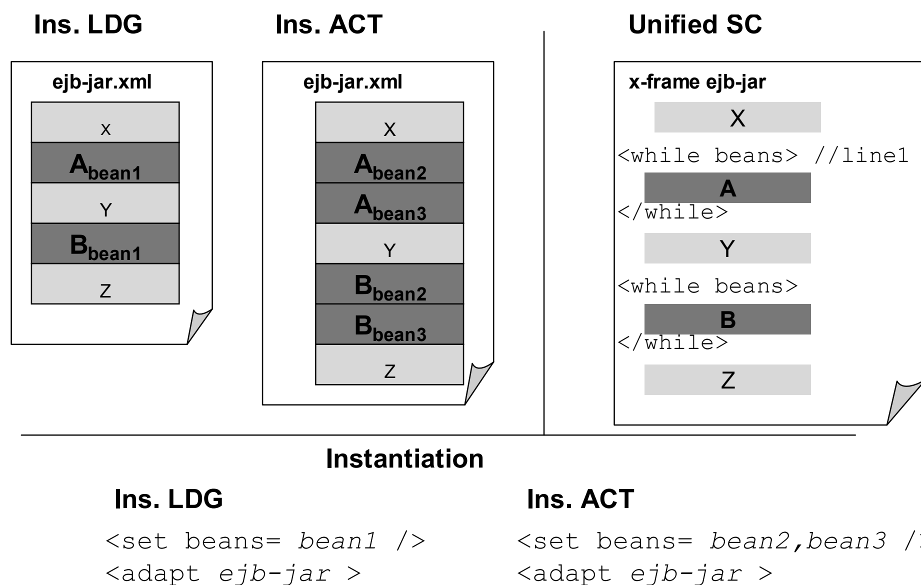


Figure 99. Handling repetitive entities in AB

8.8. Chapter conclusions

SuM is the combination of the mixed-strategy and the structural clone (SC) concept. With the alignment of the XVCL solution along SC boundaries, as advocated by SuM, we hope to achieve better structured XVCL solutions.

XVCL's independency from the semantics of the clone contents greatly reduces the size of the SC unification problem space.

The techniques for representing all basic entity types and basic structure types have been described. SuM unification schemes for identifying and handling all basic structural variations have been presented.

Defining basic SuM schemes promotes a common SuM vocabulary, and could serves as the basis for standardization of SuM solutions, developing SuM best practices, and creating tool support for SuM.

Chapter 9.

Conclusions and Future Work

Good judgement is the result of experience.
Experience is the result of bad judgement.

- Fred Brooks

During the initial stages of our research we defined, and used, a need-oriented framework for organizing web technologies. We also provided concrete evidence of the cloning problem in the web domain, and showed that cloning is more substantial in web applications as compared to traditional applications. This work also identified similarity metrics useful for evaluating the clone level of software. Based on this initial work, we decided to address two challenges in effective clone management: tenacious clones, and clone fragmentation.

Some tenacious clones are simply not unifiable using available conventional clone unification mechanisms, while others persists due to trade-offs incurred by the unification. As further evidence of such tenacious clones, we described two case studies in which generics in Java and C++ failed to unify certain clones. Based on those two case studies, and in other studies done by our research group, we accepted the 'mixed-strategy' (a mix of conventional techniques and meta-programming) as an acceptable method to unify non-unifiable clones. Taking the success of mixed-strategy one step further, we showed how mixed-strategy can also avoid most unification trade-offs incurred by conventional clone unification techniques.

Then, we illustrated the concept of structural clones (higher-level clones, typically cloned structures consisting of multiple program entities) using various examples we found in

software. We showed how clone fragmentation can be compensated using Basit's definition of structural clones, which defines a structural clone as "a configuration of lower level clones".

As the culmination point of our research, we present SuM (Structural clone management using Mixed-strategy) - a systematic and holistic approach to unify and reuse fragmented, possibly tenacious, structural clones, without compromising other desirable qualities in the software. We presented the basic activities involved in applying SuM to an existing system or a system under development. We further supported the SuM approach by presenting the basic SuM unification schemes.

Conclusions

The main conclusions of our thesis are as follows.

- Cloning is high in the web domain; higher than that of traditional applications. Given the increasing proliferation of web applications, it is timely and worthwhile to address the cloning problem in the context of web applications.
- Clones are generally bad for maintenance; majority of negative effect of clones are directly related to maintenance. However, clones have both positive and negative aspects. Therefore, clone management needs to be a balanced approach that combats negative effects of clones without losing their benefits.
- There are many reasons behind creating clones. Only some of the clones so created can be removed later. Even less can be prevented altogether. Therefore, we need to complement preventive and corrective measures with compensatory measures. However, prevailing compensatory techniques are not up to this task.

- Two fundamental challenges in effective clone management are tenacious clones, and clone fragmentation. Clones can be tenacious because they are simply non-unifiable, or because of their unification trade-offs.
- Complementing conventional techniques with a powerful meta-programming technique, as advocated by the 'mixed-strategy' enables us to unify clones that are non-unifiable using conventional clone unification techniques.
- At times it is technically feasible to use conventional programming techniques to unify most of the clones, yet such unification forces trade-offs (i.e., unification trade-offs) in many important web application properties. Similar clone unification levels could be achieved using the mixed-strategy, more importantly, without incurring such trade-offs. Therefore, mixed-strategy offers a viable solution to avoid unification trade-offs.
- Structural clones are coarse-grained higher level clones. Managing structural clones can bring in more benefits when compared to managing cloned code fragments. However, structural clones get fragmented into patterns of finer-grained clones due to decomposition and variations introduced during implementation/maintenance. Basit's definition of a structural clone as "a configuration of lower-level clones" provides an excellent abstraction mechanism to explain the fragmented coarse-grained clones in terms of the resultant clone fragments.
- SuM is the combination of the mixed-strategy and the structural clone (SC) concept. With the alignment of the XVCL solution along SC boundaries, as advocated by SuM, we hope to achieve better structured XVCL solutions.
- XVCL's independency from the semantics of the clone contents greatly reduces the size of the SC problem space. SuM unification schemes for identifying and handling all basic structural variations promotes a common SuM vocabulary, and could serve

as the basis for standardization of SuM solutions, developing SuM best practices, and creating tool support for SuM.

Future directions

Following future extensions are anticipated on the work reported in this thesis.

- Process aspect of SuM – It is still too early to define a rigorous process for SuM, but we need to continually collect experiences in applying SuM, and work towards defining a SuM process. Such a process should address issues such as:
  - Integration with currently followed process
  - Points of the SDLC at which SuM may be adopted
  - Various possible scopes of adoption (at individual level, at team level, company-wide etc.),
- Develop taxonomies – Developing SC taxonomies is a natural step that follows from our work in this thesis. The foundation has been set by defining the basic entity/structure/variation types, but a more extensive taxonomy needs to be built upon this foundation.
- Tool integration – Mixed-strategy (and XVCL) already has a suite of tools such as XVCL workbench, metric tool, and back propagation tool. But the integration of this tools into the SuM is yet to be done, and an important prerequisite to the success of SuM.
- More case studies – There is always room for more case studies to gather more evidence of tenacious clones, and clone fragmentation, and the areas in which SuM could help to overcome these challenges.

Bibliography

[ABB+04] Acerbis, R., Bongio, A., Butti, S., Ceri, S., Ciapessoni, F., Conserva, C. Fraternali, P., and Carughi, G. T., "WebRatio, an Innovative Technology for Web Application Development," *Lecture Notes in Computer Science*, vol. 3140 (Web Engineering: 4th International Conference), ICWE 2004, pp. 613 – 614.

[ACDG01] Aversano, L., Canfora, G., De Lucia, A., and Gallucci, P., "Web site reuse: cloning and adapting," *Proc. 3rd Intl. Workshop on Web Site Evolution*, (WSE'01), pp.107 - 111.

[ACM03] Alur, D., Crupi, J., and Malks, D., *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2003.

[ACV+05] Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., and Ramalho, G. "Comparative Analysis of Porting Strategies in J2ME Games,". *Proc. 21st IEEE International Conference on Software Maintenance* (ICSM'05), Budapest, Hungary, pp. 123-132, 2005.

[AH02] Aragones, A., and Hart-Davidson, W., "Why, when and how do users customize Web portals?," *Proc. IEEE International Professional Communication Conference* (IPCC 2002), pp.375 - 388.

[AHAD05] April, A., Hayes, J. H., Abran, A., and Dumke, R., "Software Maintenance Maturity Model^{mm} : the software maintenance process model," *Journal of Software Maintenance and Evolution: Research and Practice*, vol 17, issue 3, pp 197-223, 2005.

[AKHG05] Al-Ekram, R., Kapser, C., Holt, R., and Godfrey, M. "Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems," *Proc. 2005 Intl. Symposium on Empirical Software Engineering* (ISESE-05), Noosa Heads, Australia, pp. 376 – 385, 2005.

[AVMD02] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta - "Analyzing Cloning Evolution in the Linux Kernel," *Journal of Information and Software Technology,* vol.44 issue 13, October 2002, pp. 755-765.

[Bak95] Baker, B. S., "On finding duplication and near-duplication in large software systems," *Proc. 2nd Working Conference on Reverse Engineering*, 1995, pages 86-95.

[Bas97] Bassett, P., *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997.

[BB02] Burd, E., and Bailey, J., "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *2nd IEEE Intl. Workshop on Source Code Analysis and Manipulation* (SCAM'02) pp. 36-43.

[BBD00] Burd, E., Bradley, S., and Davey, J. "Studying the Process of Software Change: An Analysis of Software Evolution," *Proc. Working Conference on Reverse Engineering* (WCRE 2000), pp 232-239.

[BBH98] Brereton, P., Budgen, D., and Hamilton, G. "Hypertext: the next maintenance mountain," *Computer* , vol. 31 , no. 12 , Dec. 1998, pp. 49 – 55.

[BBL01] Boldyreff, C., Burd, E., and Lavery, J., "Towards the Engineering of Commercial Web-Based Applications," *Intl. Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 2001.

[BVVT04] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwé. T., An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. *Proc. Intl.*

Conference on Software Maintenance (ICSM04). IEEE Computer Society, 2004, pp. 200-209.

[BVVT05] Bruntink, M., van Deursen, A., van Engelen R., and Tourwé, T., “On the Use of Clone Detection for Identifying Cross Cutting Concern Code,” *IEEE Transactions on Software Engineering*, vol 31 issue 10, Oct 2005, pp. 804 - 818.

[Big94] Biggerstaff, T. “The library scaling problem and the limits of concrete component reuse,” *Proc. 3rd Int. Conf. on Software Reuse* (ICSR'94), 1994, pp. 102-109.

[BJ05] Basit, A.H. and Jarzabek, S. “Detecting Higher-level Similarity Patterns in Programs,” *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC-FSE'05), pp. 156-165.

[BK01] Boldyreff, C., and Kewish, R., “Reverse engineering to achieve maintainable WWW sites,” *Proc. 8th Working Conf. on Reverse Eng.* (WCRE 01), pp. 249 – 257.

[BM97] Burd, E., and Munro, M., “Investigating the Maintenance Implications of the Replication of Code,” *Proc. IEEE Intl. Conference on Software Maintenance* (ICSM ’97), pp. 322-329.

[BMD+00] Balazinska, M., Merlo, E., Dagenais M., Lague, B., and Kontogiannis, K., “Advanced Clone-analysis to Support Object-oriented System Refactoring,” *Proc. Working Conference on Reverse Engineering* (WCRE 200), pp. 98 – 107.

[BMDL99] Balazinska, M., Merlo, E., Dagenais, and M., Lagüe, B., “Measuring clone based reengineering opportunities,” *Proc. 6th International Software Metrics Symposium*, 1999, pp. 292-303.

[BMD+99] Balazinska, M., Merlo, E., Dagenais, M., Lagüe, and B., and Kontogiannis, K.A., “Partial redesign of Java software systems based on clone analysis,” *Proc. 6th IEEE Working Conference on Reverse Eng.*, 1999, pp. 326-336.

[BRJ05a] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "Beyond Templates: a Study of Clones in the STL and Some General Implications," *Proc. 28th Intl. Conf. on Software Engineering* (ICSE'05), pp. 451-459.

[BRJ05b] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "An Empirical Study on Limits of Clone Unification Using Generics," *Proc. 17th Intl. Conference on Software Engineering and Knowledge Engineering* (SEKE'05), 2005, pp. 109-114.

[BRJ06] Basit, A.H., Rajapakse, D.C. and Jarzabek, S. "Structural Clones – Higher Level Similarity Patterns in Programs" Draft available from http://www.comp.nus.edu.sg/~damithch/files/StructuralClones2006.pdf

[BYM+98] Baxter, I., Yahin, A., Moura, L., Anna, M. S., and Bier, L. "Clone detection using abstract syntax trees," *Proc. Intl. Conference on Software Maintenance* (ICSM '98), pp. 368-377.

[CAV+01] Casazza, G., Antoniol, G., Villano, U., Merlo, E., and Di Penta, M., "Identifying clones in the Linux kernel," *Proc. 1st IEEE Intl. Workshop on Source Code Analysis and Manipulation*, 2001, pp. 90 – 97.

[CD03] Capilla, R., Duenas, J.C., "Light-weight product-lines for evolution and maintenance of Web sites," *Proc. Seventh European Conference on Software Maintenance and Reengineering,* (CSMR' 2003), pp. 53 – 62.

[CDS04] Cordy, J. R., Dean, T. R., and Synytskyy, "Practical Language-Independent Detection of Near-Miss Clones," *Proc. 14th IBM Center for Advanced Studies Conference* (CASCON'04), pp. 29-40.

[CE00] Czarnecki, K. and Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[CFB00] Ceri, S., Fraternali, P., and Bongio, A., "Web Modeling Language (WebML): a modeling language for designing Web sites," Computer Networks, vol 33, issues 1-6, 1999, pp. 137-157.

[CFM02] Ceri, S., Fraternali, P., and Matera, M., "Conceptual modeling of data-intensive Web applications". *IEEE Internet Computing* vol 6, issue 1, 2002, pp 20 – 30.

[CLM04] F. Calefato, F. Lanubile, and T. Mallardo, "Function Clone Detection in Web Applications: A Semiautomated Approach", *Journal of Web Engineering*, vol.3, no.1, May 2004, pp. 3-21.

[CKGN05] Centeno, V. L., Kloos, C. D., Gaedke, M., and Nussbaumer, M., "Web composition with WCAG in mind," *Proc. 2005 International Cross-Disciplinary Workshop on Web Accessibility* (W4A 2005), pp. 38-45.

[Con00] Conallen, J, *Building Web Applications with UML*, Addison Wesley, 2000, p 25.

[Cor03] Cordy, J. R., "Comprehending Reality: Practical Challenges to Software Maintenance Automation," *Proc. 11th IEEE Intl. Workshop on Program Comprehension* (IWPC 2003), pp. 196-206.

[CP04] Christodoulou, S. P., and Papatheodorou, T. S., "WEP: A Reference Model and the Portal of Web Engineering Resources", *Proc. Intl Workshop on Web Engineering*, 2004.

[CSP98] Christodoulou, S., Styliaras, G., and Papatheodourou, T. S., "Evaluation of Hypermedia Application Development and Management Systems", *Proc. of the 9th ACM Conference on Hypertext and Hypermedia, Pittsburgh*, 1998.

[CTP03] Christodoulou, S. P., Tzimou D. G., and Papatheodorou, T. S., "An Evaluation Support Framework for Internet Technologies and Tools", *Proc. IASTED conference on Communications, Internet and Information Technology*, 2003.

[CZP01] Christodoulou, S. P., Zafiris, P. A., and. Papatheodorou, T. S., “Web Engineering: The Developers’ View and a Practitioner’s Approach”, *LNCS (Web Engineering: Managing Diversity and Complexity in Web Application Development)* ed. San Murugesan and Yogesh Deshpande, Vol. 2016, Springer-Verlag, April 2001.

[DDAC01] Di Lucca, G. A., Di Penta, M., Antoniol, G., and Gerardo Casazza, G., “An Approach for Reverse Engineering of Web-Based Application,” *Proc. 8th Working Conference on Reverse Engineering* (WCRE'01), pp. 231-240.

[DFST04] De Lucia, A., Francese, R., Scanniello, G., Tortora, G., “Reengineering Web Applications Based on Cloned Pattern Analysis”. *Proc.12th IEEE International Workshop on Program Comprehension* (IWPC'04), pp. 132-141.

[DFST05] De Lucia, A., Francese, R., Scanniello, G., and Tortora, G., “Understanding Cloned Patterns in Web Applications”, *Proc. 13th International Workshop on Program Comprehension* (IWPC05), pp 333 – 336.

[DFTV04] Di Lucca, G. A., Fasolino, A. R., Tramontana, P., and Visaggio, C. A., “Towards the Definition of a Maintainability Model for Web Applications,” *Proc. Conference on Software Maintenance and Reengineering* (CSMR04), pp. 279-287.

[DST04] De Lucia, A., Scanniello, G., and Tortora, G., “Identifying Clones in Dynamic Web Sites Using Similarity Thresholds,” (ICEIS04), pp. 391-396.

[DDFG01] Di Lucca, G. A., Di Penta, M., Fasilio, A. R., and Granato, P., “Clone analysis in the web era: An approach to identify cloned web pages,” *Proc. 7th IEEE Workshop on Empirical Studies of Software Maintenance* (WESS 2001), pp. 107 - 113.

[DDF02] Di Lucca, G.A., Di Penta, M., Fasolino, A.R., “An approach to identify duplicated web pages,” *Proc. 26th Annual Intl. Computer Software and Applications Conference* (COMPSAC02), pp. 481 - 486.

[DFTD02] Di Lucca, G. A., Fasolino, A. R., Pace, F., Tramontana, P., de Carlini, U., "WARE: A Tool for the Reverse Engineering of Web Applications," *Proc. 6th European Conference on Software Maintenance and Reengineering* (CSMR02), pp. 241-250.

[DNAM05] Di Penta, M., Neteler, M., Antoniol, G., and Merlo, G., "A language-independent software renovation framework," *Journal of Systems and Software* vol. 77, no. 3, 2005,pp. 225-240.

[DMG+02] Deshpande, Y., Murugesan, S., Ginige, A., Hansen, S., Schwbe, D., Gaedke, M. and White, B., "Web Engineering," *Journal of Web Engineering,* vol. 1, 2002, pp. 3 – 17.

[DRD99] Ducasse, S, Rieger, M., and Demeyer, S., "A language independent approach for detecting duplicated code," *Proc. Intl. Conference on Software Maintenance* (ICSM '99), pp. 109-118.

[Fow03] Fowler, M., *Patterns of Enterprise Application Architectur*e, Addison-Wesley, 2003

[Fow96] Fowler, M., *Analysis Patterns*, Addison-Wesley, 1996.

[Fow99] Fowler, M., *Refactoring - Improving the Design of Existing Code*, Addison Wesley, 1999.

[FP98] Fraternalli, P., and Paolini, P. "A Conceptual Model and a Tool Environment for Developing More Scalable, Dynamic, and Customizable Web Applications," *LNCS (Proc. of the 6th International Conference on Extending Database Technology: Advances in Database Technology)*, vol. 1377, pp. 421 – 435.

[FR99] Fanta, R., and Rajlich, V, "Removing Clones from the Code" *Journal of Software Maintenance,* vol. 11 , no. 4 July/Aug. 1999, pp. 223 – 243.

[GBGM89] Blair, G. S., Gallagher, J. J., and Malik, J., "Genericity vs inheritance vs delegation vs conformance," *Journal of Object-Oriented Programming*, vol. 2, No. 3, Sept./Oct. 1989, pp. 11-17.

[GDG05] Ginige, J. A., De Silva, B., Ginige, A. "Towards End User Development of Web Applications for SMEs: A Component Based Approach," *Proc. Internation Conference on Web Engineering* (ICWE05), pp. 489-499.

[GDKZ04] Godfrey, M., Dong, X., Kapser, C., and Zou, L.,. "Four Interesting Ways in Which History Can Teach Us About Software", *Position paper, International Workshop on Mining Software Repositories* (MSR04), pp. 58 – 62.

[GG00] Gaedke M., and Graef. G., "Development and evolution of web-applications using the webcomposition process model,". *Proc. International Workshop on Web Engineering at the 9th International WorldWide Web Conference (WWW9)*, 2000.

[GGS+99] Gaedke, M., Gellersen, H., Schmidt, A., Stegemüller, U., and Kurr, W., "Object-oriented Web Engineering for Large-scale Web Service Management," *Proc. 32nd Annual Hawaii International Conference on System Sciences* (HICSS99), vol. 5, p. 5029

[GHJ97] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design patterns: Elements of reusable object-oriented software*, Addison-wesley, 1997.

[GL03] Gallagher, K. and Layman, L. "Are decomposition slices clones?" *Proc. 11th IEEE International Workshop on Program Comprehension* (IWPC 03), 2003, pp 251-256.

[GM01a] Ginige, A., and Murugesan, S, "Guest Editors' Introduction: The Essence of Web Engineering-Managing the Diversity and Complexity of Web Application Development," *IEEE MultiMedia* vol. 8 no. 2, 2001, pp. 22-25.

[GM01b] Ginige, A., and Murugesan, S, "Guest Editors' Introduction: Web Engineering - An Introduction," *IEEE MultiMedia* vol. 8, no. 1, 2001, pp. 14-18.

[GNM04a] Gaedke, M., Nussbaumer, M., and Meinecke J., "WSLS: A Service-Based System for Reuse-Oriented Web Engineering," *Proc. 4th International Workshop on Web-oriented Software Technology* (IWWOST 2004), pp. 11-26.

[GNM04b] Gaedke, M., Nussbaumer, M., and Meinecke, J., "WSLS: An Agile System Facilitating the Production of Service-Oriented Web Applications," *Proc. 4th International Workshop on Web-Oriented and Software Technologies* (IWWOST 2004), pp. 26-37.

[GNT03] Gaedke, M., Nussbaumer, M., and Tonkin E., "WebComposition Service Linking System: Supporting development, federation and evolution of service-oriented Web applications," *Proc. 3rd Int. Workshop on Web-oriented Software Technology* (IWWOST 2003).

[GSG00] Gaedke, M., Segor, C., and Gellersen, H., "WCML: Paving the Way for Reuse in Object-Oriented Web Engineering," *Proc. ACM Symposium on Applied Computing* (SAC 2000), pp. 748-755.

[GT99] Gaedke, M. and Turowski, K. "Generic Web-Based Federation of Business Application Systems for E-Commerce Applications," *Proc. Engineering Federated Information Systems* (EFIS99), 1999, pp. 25-42.

[HKK+04] Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K., "ARIES: Refactoring Support Environment Based on Code Clone Analysis," *Proc. 8th IASTED International Conference on Software Engineering and Applications* (SEA 2004), 2004, pp.222-229.

[HROS01] Heberle, A., Rehse, J., Onasch, B., and Sieling, B., "Utilizing Abstract WebEngineering Concepts: An Architecture," *Proc. 34th Annual Hawaii International Conference on System Sciences* (HICSS-34), 2001, pp. 70-79,

[HT02] Hazra, T. K., "Building enterprise portals: principles to practices," *Proc. 24th International Conference in Software Engineering* (ICSE02) pp. 623-633.

[HUK+02] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, "On software maintenance process improvement based on code clone analysis," *Proc. of International Conference on Product Focused Software Process Improvement* (Profes02), pp. 185-197

[JBZZ03] Jarzabek, S., Basset, P., Zhang, H. and Zhang, W. “XVCL: XML-based Variant Configuration Language,” *Proc. Int. Conf. on Software Engineering* (ICSE’03), May 2003, Portland , pp. 810-811

[JL03] Jarzabek, S. and Li S., "Eliminating Redundancies with a “Composition with Adaptation” Meta-programming Technique," *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC-FSE'03), ACM Press, September 2003, Helsinki, pp. 237-246.

[JS00] Jarzabek, S. and Seviora, R. “Engineering components for ease of customization and evolution,” *IEE Proceedings - Software (a special issue on Component-based Software Engineering)*, vol. 147, no. 6, December 2000, pp. 237-248,

[JZ01] Jarzabek, S. and Zhang, H. “XML-based Method and Tool for Handling Variant Requirements in Domain Models”, *Proc. 5th International Symposium on Requirements Engineering* (RE’01), August 2001, Toronto, Canada, pp. 166-173

[Joh93] Johnson, J. H., “Identifying redundancy in source code using fingerprints,” *Proc. 1993 Conf. of the Centre for Advanced Studies on Collaborative research: software engineering* (CASCON ’93), pp 171-183.

[Joh94] Johnson, J. H., “Substring Matching for Clone Detection and Change Tracking,” *Proc. Intl. Conference on Software Maintenance* (ICSM ‘94), pp. 120–126.

[Joh96] Johnson, H., "Navigating the Textual Redundancy Web in Legacy Source," *Proc. Conference of the Centre for Advanced Studies on Collaborative research* (CASCON '96), pp. 7-16.

[Joh00] Johnson, J. H., Mackay, S. A., “Witan web and the software engineering of web-based applications,” *Proc. of the 2000 conference of the Centre for Advanced Studies on Collaborative research* (CASCON’2000), pp. 5-20.

[Kan90] Kang, K et al. 1990. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh, Nov. 1990.

[KBLN04] Kim, M., Bergman, L., Lau, T., and Notkin, D., "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *Proc. International Symposium on Empirical Software Engineering, August* (ISESE'04), 2004, pp. 83 – 92.

[KG03a] Kapser, C., and Godfrey, M. W., "A Taxonomy of Clones in Source Code: The Re-Engineers Most Wanted List", 2nd International Workshop on Detection of Software Clones (IWDSC-03), Victoria BC, November 2003.

[KG03b] Kapser, C., and Godfrey, M. W., "Toward a Taxonomy for Source Code Cloning: A Case Study", *Proc. of the 2003 Intl. Workshop on Evolution of Large-scale Industrial Software Applications* (ELISA).

[KG05] Kapser, C., and Godfrey, M. W., "Improved Tool Support for the Investigation of Duplication in Software", *Proc. of the 2005 Intl. Conference on Software Maintenance* (ICSM05), Budapest, Hungary, pp. 305 – 314.

[KG06] Kapser. C., and Godfrey. M. W. "'Cloning Considered Harmful' Considered Harmful", *Proc. of the 2006 Working Conference on Reverse Engineering* (WCRE'06)

[KH01a] Komondoor, R., and Horwitz, S., "Tool Demonstration: Finding Duplicated Code Using Program Dependences," *Proc. European Symposium on Programming Languages* (ESOP '01), pp. 383-386.

[KH01b] Komondoor, R., and Horwitz, S., "Using slicing to identify duplication in source code," *Proc. 8th International Symposium on Static Analysis,* 2001, pp. 40-56.

[KKI02] Kamiya, T., Kusumoto, S., and Inoue, K., "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. Software Engineering*, vol. 28, no. 7, pp. 654-670.

[KKJK01] Kerer, C., Kirda, E., Jazayeri, M., and Kurmanowytsch, R., "Building and Managing XML/XSL-powered Web Sites: an Experience Report," *Proc. International Computer Software and Applications Conference* (COMPSAC 2001) pp. 547-554.

[KMP+04] Kappel, G., Michlmayr, E., Pröll, B., Reich, S., and Retschitzegger, W., "Web Engineering - old wine in new bottles?," *Proc. 4th International Conference on Web Engineering* (ICWE2004), pp. 6-12.

[KMW03] Kienle, H. M., Müller, H. A., and Weber, A., "In the Web of Generated "Clones"," *Proc. 2nd Intl. Workshop on detection of clones*, (IWDSC'2003).

[KN05] Kim M., and Notkin, D., "Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones" *Proc. 2nd International Workshop on Mining Software Repositories*(MSR), co-located with ICSE 2005, pp. 1-5.

[KSBM99] Kwon, O., Shin, G., Boldyreff, C., and Munro, M., "Maintenance with Reuse: An Integrated Approach Based on Software Configuration Management," *Proc. 6th Asia-Pacific Software Engineering Conference* (APSEC'99), p. 507-515.

[KSNM05] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. C., "An Empirical Study of Code Clone Genealogies," Proc. *10th European Software Engineering Conference and the 13th Foundations of Software Engineering* (ESEC/FSE 2005), pp. 187 – 196.

[Kos04] Koskinen, J., Software Maintenance Costs, URL: http://www.cs.jyu.fi/~koskinen/smcosts.htm, last updated 2004.

[Kru92] Krueger, C.W., Software Reuse. *ACM Computing Surveys*, vol. 24, no. 2, 1992, pp. 131-183.

[LJ05] Lee, M., Jefferson, and T. L., “An Empirical Study of Software Maintenance of a Web-Based Java Application,” *Proc. Inernational Conference on Software Maintenance* (ICSM05), pp. 571-576.

[LM03] Lanubile, F., and Mallardo, T., “Finding Function Clones in Web Applications,” *Proc. Seventh European Conference on Software Maintenance and Reengineering* (CSMR’03), pp. 379.

[LPM+97] Lague B., Proulx D., Mayrand J., Merlo E., and Hudepohl J., “Assessing the benefits of incorporating function clone detection in a development process,” *Proc. Intl. Conference on Software Maintenance* (ICSM ’97), pp. 314-321.

[LR04] Loh, A., and Robey, M., “Generating Web Applications from Use Case Scenarios,” *Proc. 2004 Australian Software Engineering Conference* (ASWEC'04), 2004, pp. 320.

[MDHG99] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige, “Web Engineering: A New Discipline for Development of Web-based Systems,” *Proc. First ICSE Workshop on Web Engineering,* Los Angeles, CA, May 16-17, 1999.

[MM01] Miller R. C., and. Myers, B. A., “Interactive simultaneous editing of multiple text regions,” *USENIX Annual Technical Conference*, General Track, 2001, pp. 161-174,

[MNK+02] Monden, A., Nakae, D., Kamiya, T., Sato, S., and Matsumoto, K., “Software quality analysis by code clones in industrial legacy software,” *Proc. of the 8th IEEE Symposium on Software Metrics* (METRICS2002), 2002, pp. 87-94.

[MW01] McDonald A. and Welland R., “Web Engineering in Practice”, *Proc.4th WWW10 Workshop on Web Engineering*, pp. 21-30.

[MLM96] Mayrand, J., Leblanc, C., and Merlo, E, “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics,” *Proc. International Conference on Software Maintenance* (ICSM’96), pp. 244-253.

[MM01] Marcus, A., and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code," *Proc. Automated Software Engineering*, 2001, pp. 107-114.

[MW01] McDonald A. and Welland R., "Agile Web Engineering (AWE) Process", Department of Computing Science Technical Report TR-2001-98, University of Glasgow, Scotland, 2 December 2001, http://www.dcs.gla.ac.uk/~andrew/TR-2001-98.pdf

[MW04] McDonald A. and Welland R., "Evaluation of Commercial Web Engineering Processes", Koch N., Fraternali P. & Wirsing M. (Eds.): *Fourth International Conference on Web Engineering,* ICWE 2004, LNCS 3140, Page(s): 166-170, July 2004. ISBN: 3-540-22511-0.

[Mye95] Myers N. C., "Traits: a new and useful template technique," *C++ Report*, June 1995

[NMT05] Nguyen, T. N., Munson, E. V., and Thao, C., "Managing the Evolution of Web-Based Applications with WebSCM," *Proc. International Conference on Software Maintenance* (ICSM'05), pp. 577-586.

[NS03] Nickell, E. and Smith, I. E., "Extreme programming and software clones," *Proc. 2nd International Workshop on the Detection Of Software Clones* (IWDSC 2003), 2003.

[NWG00] Ng, E. H., Wade, S., and Ghaoui, C., "Web page reuse techniques: a dynamic referential navigational guide," *Proc. 26th Euromicro Conference,* vol. 2, pp. 72 - 77.

[Opd92] Opdyke, W. F., Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science, 1992, http://citeseer.nj.nec.com/article/opdyke92refactoring.html.

[PAF01] Pastor, O., Abrahão, S. M., Fons, J., "An Object-Oriented Approach to Automate Web Applications Development", *Proc. 2nd International Conference on Electronic Commerce and Web Technologies* (EC-Web'01), pp. 16-28.

[Par94] Parnas, D., "Software aging," *Proc. 16th International Conference on Software Engineering* (ICSE 1994), pages 279 -287.

[PJ05] Pettersson, U., and Jarzabek, S. "Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach," *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC-FSE'05),, ACM Press, September 2005, Lisbon, pp. 326-335.

[Pig97] Pigoski, T. M., *Practical software maintenance*, Wiley computer publishing, 1997.

[PK04] Ping, Y, and Kontogiannis, K., "Refactoring Web sites to the Controller-Centric Architecture," *Proc. 8th Euromicro Working Conference on Software Maintenance and Reengineering* (CSMR'04), pp.204-213.

[PKL04] Ping, Y, Kontogiannis, K., and Lau, T. C., "Transforming Legacy Web Applications to the MVC Architecture," Proc. 11th Annual International Workshop on Software Technology and Engineering Practice (STEP'04) pp. 133 – 142.

[PMP02] Prechelt, L., Malpohl, G., and Philippsen, M., "JPlag: Finding plagiarisms among a set of programs," J*ournal of universal Computer Sc.*, vol 8, issue 11, 2002.

[Pre98] Pressman, R.S., et al. "Can Internet-Based Applications Be Engineered?," *IEEE Software*, vol. 15, no. 5, Sept 1998, pp. 104 – 110.

[Pre00] Pressman, R.S., "What a tangled Web we weave [Web engineering]," *IEEE Software*, vol. 17 , no. 1 , Jan.-Feb. 2000, pp.18 – 21.

[RJ05a] Rajapakse, D.C and Jarzabek, S. "A Need-Oriented Assessment of Technological Trends in Web Engineering," *Proc. Int. Conf. on Web Engineering* (ICWE'05), July 2005, Sydney, pp. 30-35.

[RJ05b] Rajapakse, D. C., and Jarzabek, S., "An Investigation of Cloning in Web Applications," *Proc. 5th Intl Conference on Web Engineering* (ICWE'05), Sydney, Australia, 2005, pp. 252-262.

[Rie05] Rieger, M. "Effective clone detection without language barriers", PhD thesis

[RSG97] Rossi, G., Schwabe, D., and Garrido, A., "Design Reuse in Hypermedia Applications Development," *Proc. 8th ACM Conference on Hypertext and Hypermedia*, 1997, pp. 57-66.

[RSL00] Rossi, G., Schwabe, D., and Lyardet, F., "Abstraction and Reuse Mechanisms in Web Application Models," *Proc. Workshops on Conceptual Modeling Approaches for E-Business and The World Wide Web and Conceptual Modeling: Conceptual Modeling for E-Business and the Web*, LNCS, Vol. 1921, pp. 76 – 88.

[RSL03] Rossi, G., Schmid, H. A., and Lyardet, F., "Customizing Business Processes in Web Applications," *Proc. Electronic Commerce and Web Technologies* (EC-Web 2003), pp. 359-368.

[RT03] Ricca, F., and Tonella, P., "Using clustering to support the migration from static to dynamic web pages," *Proc. 11th IEEE International Workshop on Program Comprehension* (IWPC' 2003), pp. 207 – 216.

[RT05] Ricca, F., and Tonella, P., "Anomaly Detection in Web Applications: A Review of Already Conducted Case Studies," *Proc. 9th European Conference on Software Maintenance and Reengineering* (CSMR'05), 2005, pp. 385-394.

[SCD03] Synytskyy, N. Cordy, J. R., and Dean, T., "Resolution of static clones in dynamic Web pages," *Proc. Fifth IEEE Intl. Workshop on Web Site Evolution* (IWSE'2003), pp. 49 – 56.

[SERL01] Schwabe, D.,Esmeraldo, L., Rossi, G., and Lyardet, F., “Engineering Web Applications for Reuse,” *IEEE MultiMedia*, vol. 8, no. 1, 2001, pp. 20-31.

[SVB05] Svahnberg, M. van Gurp, J and Bosch, J., “A taxonomy of variability realization techniques,” *Software - Practice & Experience*, vol. 35, Issue 8, July 2005, pp. 705 – 754.

[Som00] Sommerville, I. *Software Engineering (6th Edition),* Addison-Wesley, 2000.

[SR04] Schmid, H. A., and Rossi, G., “Modeling and Designing Processes in E-Commerce Applications,” *IEEE Internet Computing*, vol. 08, no. 1, Jan/Feb, 2004, pp. 19-27.

[SR98a] Schwabe, D. and Rossi, G., “Developing Hypermedia Applications using OOHDM,” Workshop on Hypermedia Development Processes: Methods and Models (Hypertext'98). 1998.

[SR98b] Schwabe, D. and Rossi, G., “An object-oriented approach to web-based application design,” *Theory and Practise of Object Systems* (TAPOS), Special Issue on the Internet, vol. 4, no. 4, October 1998, pp. 207-225.

[SRB96] Schwabe, D. and Rossi, G., and Barbosa, S. D. J., “Systematic Hypermedia Application Design with OOHDM,” *Proc. ACM Hypertext'96*, 1996, pp. 116-128.

[SREL01] Schwabe, D. and Rossi, G., Esmeraldo, L., and Lyardet, F., “Web Design Frameworks: An Approach to Improve Reuse in Web Applications,” *LNCS*, vol 2016, pp. 335-352.

[Sut95] Sutherland, J. (1995). “Business objects in corporate information systems,” *ACM Computing Surveys*, vol. 27, no. 2, pp. 274-276.

[TOHS99] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, *Proc. International Conference on Software Engineering* (ICSE’99), Los Angeles, 1999, pp. 107-119.

[TRPG02] Tonella, P., Ricca, F., Pianta, E., and Girardi, C., "Restructuring Multilingual Web Sites," Proc. International Conference on Software Maintenance (ICSM'02), 2002, pp. 290-299.

[TST03] Taguchi, M., Suzuki, T., and Tokuda, T., "A visual approach for generating server page type web applications based on template method," *Proc. IEEE Symposium on Human-Centric Computing Languages and Environments* (HCC 2003), pp. 248-250.

[Tro03] Trowbridge et al., *Enterprise Solution Patterns Using Microsoft .NET (Version 2.0)*, Microsoft Press, 2003

[UHK+02] Ueda, Y., Higo, Y., Kamiya, T., Kusumoto, S., and Inoue, K., "Gemini: Code Clone Analysis Tool," *Proc. of 2002 International Symposium on Empirical Software Engineering* (ISESE2002), vol.2, 2002, pp.31-32.

[UKKI02a] Ueda, Y., Y., Kamiya, T., Kusumoto, S., and Inoue, K., "On Detection of Gapped Code Clones using Gap Locations," *Proc. Asia Pacific Software Engineering Conference* (APSEC 02), pp. 327-336.

[UKKI02b] Ueda, Y., Y., Kamiya, T., Kusumoto, S., and Inoue, K., "Gemini: Maitenance Support Environment Based on Code Clone Analysis," *Proc. of the 8th IEEE Symposium on Software Metrics* (METRICS2002), pp. 67-76.

[WBM99a] Warren, P., Boldyreff, C., and Munro, M., "Characterising Evolution Web Sites: Some Case Studies," *First International Workshop on Web Site Evolution*, (WSE'99),

[WBM99b] Warren, P., Boldyreff, C., and Munro, M. "The evolution of Websites," *Proc. 7th Intl. Workshop on Program Comprehension* (IWPC'99), pp.178 – 185.

[Wei71] Weinberg, G. M., "The Psychology of Computer Programming," Van Nostrand Reinhold Ltd. New York, 1971.

[WLK04] Walenstein, A., Lakhotia, A., and Koschke R., "The Second International Workshop on Detection of Software Clones: Workshop Report", *SIGSOFT Software Eng. Notes*, vol. 29, no. 2, March 2004, pp. 1-5.

[Won99] Wong, K., "Toward Reusable and Evolvable Web Sites," *Proc. 1st Annual Workshop on Web Site Evolution* (WSE'99), pp. 49-52.

[XVCL] "XML-based Variant Configuration Language," XVCL Website, http://xvcl.comp.nus.edu.sg

[YJ05]Yang, J. and Jarzabek, S. "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," *Proc. 4th Int. Conf. on Generative Programming and Component Engineering* (*GPCE'05)*, 2005, Tallinn, Estonia, pp. 237-255.

[ZB03] Zhang, J., and Buy, U., "A Framework for the Efficient Production of Web Applications,". *Proc. of the Eighth IEEE International Symposium on Computers and Communications* (ISCC 2003), pp. 419-424.

[Zdu02] U. Zdun. "Dynamically generating web application fragments from page templates," *Proc. of Symposium of Applied Computing* (SAC 2002), Madrid, Spain, 2002, pp. 1113-1120.

[ZJ03a] Zhang, H. and Jarzabek, S. "An XVCL approach to handling variants: A KWIC product line example," *Proc. 10th Asia-Pacific Software Engineering Conference* (APSEC'03), Chiangmai, Thailand, pp. 116-125.

[ZJ03b] Zhang, H. and Jarzabek, S., "An XVCL-based Approach to Software Product Line Development", *Proc. 15th International Conference on Software Engineering and Knowledge Engineering* (SEKE'03), San Francisco, USA, 2003, pp. 267-275.

[ZJ05] Zhang, W. and Jarzabek, S. “Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices,” *Proc. 9th Int. Software Product Line Conf.* (SPLC’05), pp. 57-69.

[ZJLR03] Zhang, W., Jarzabek, S., Loughran, N and Rashid, A. “Reengineering a PC-based System into the Mobile Device Product Line,”, *Proc. 4th Int. Workshop on Principles of Software Evolution* (IWPSE’03), IEEE Comp. Soc., 2003, Helsinki, Finland, pp. 149-160.

Appendix A: Essential XVCL Syntax

Following summary of XVCL syntax was adopted from the XVCL website [XVCL].

command: x-frame	
Syntax	**<x-frame name=** ”*name*” **>** *x-frame body*: mixture of code and XVCL commands **</x-frame>**
Attributes	**name:** is the name of the x-frame being defined.
Description	The <x-frame> command denotes the start and end of the x-frame body. The x-frame body contains textual contents (e.g., program code), instrumented with XVCL commands for ease of adaptation
command: adapt	
Syntax	<**adapt x-frame**=”*name*”> *adapt-body* : mixture of <insert>, <insert-before>, <insert-after> commands </**adapt**> or: <**adapt x-frame**=”*name*”/>
Attributes	**x-frame:** defines the name of x-frame to be adapted.
Description	The <adapt> command instructs the processor to: • adapt the x-subframework rooted in the named x-frame by inserting x-frame texts, • emit/assemble the customized content of the adapted x-subframework into the output, • resume processing of the current x-frame after processing the x-subframework rooted in the named x-frame. The *adapt-body* may contain a mixture of <insert>, <insert-before> and <insert-after> commands.
command: break	
Syntax	<**break name** =”*break-name*”> *break-body* </**break**> or <**break name** =”*break-name*”/>
Attributes	**name**: defines the name of breakpoint in an x-frame.

Description	The <break> command marks a breakpoint (slot) at which changes can be made by ancestor x-frames via <insert>, <insert-before> and <insert-after> commands. The break-body defines the default code, if any, that may be replaced by <insert> or extended by <insert-before> and <insert-after> commands.
command: insert	
Syntax	<**insert break** = ”*break-name*”> *insert-body* </**insert**> <**insert-before break** = ”*break-name*”> *insert-body* </**insert-before** > <**insert-after break**=”*break-name*”> *insert-body* </**insert-after** >
Attributes	**break**: defines the name of the breakpoint.
Description	The <insert> command replaces the breakpoint “*break-name*” in the adapted x-subframework with the *insert-body*. The <insert-before> command inserts the *insert-body* **before** the breakpoint “*break-name*” in the adapted x-subframework. The <insert-after> command inserts the *insert-body* **after** the breakpoint “*break-name*” in the adapted x-subframework. The *insert-body* may contain a mixture of textual content and XVCL commands.
command: set-var	
Syntax	<**set var** = ”*var-name*” **value** = ”*value*” **/**>
Attributes	**var**: defines the name of single-value variable. **value**: defines the value to be assigned.
Description	The <set> command assigns a “*value*” defined in the “value” attribute to single-value variable “*var-name*” defined in the “var” attribute.
command: set-multi	
Syntax	<**set-multi var**=”*var-name*” **value**=”*value1, value2, ...*” />
Attributes	**var**: defines the name of multi-value variable. **value**: defines a list of values to be assigned to the variable.
Description	The <set-multi> command assigns multiple values (*value1, value2*,…) defined in the “value” attribute to a multi-value variable “*var-name”* defined in the “var” attribute.
command: value-of	
Syntax	<**value-of expr** = ”*expression*” />
Attributes	**expr**: defines an expression to be evaluated.
Description	The value of the “*expression*” is evaluated and the result replaces the <value-of> command.

command: select	
Syntax	<**select option** = ”*var-name*”> *select-body*: may contain options listed below </**select**> select-body: <**option-undefined**> (optional) option-body </**option-undefined**> <**option value** = ”*value*”> (0 or more) option-body </**option**> <**otherwise**> (optional) option-body </**otherwise**>
Attributes	**option**: The “option” attribute in <select> command defines the variable whose value will be matched in <option> commands. **value**: The “value” attribute in <option> command defines the value to be matched.
Description	In this command, we select from a set of options based on variable “*var-name*” as follows: • <**option-undefined**> is processed, if the variable “*var-name*” is undefined, • <**option**> is processed, if the value of “*var-name*” matches <option>’s “*value*”, • <**otherwise**> is processed, if none of the <option>’s “*value*” is matched. The *option-body* may contain a mixture of textual content and XVCL commands.
command: while	
Syntax	**<while using-items-in=**”*multi-var*”**>** while-body **</while>**
Attributes	**using-items-in**: defines the multi-value variable “*multi-var*” to be used inside while.
Description	The <while> command iterates over the *while-body* using the values of multi-value variable “*multi-var*” defined in the “using-items-in” attribute. The i’th iteration uses i’th value of the “*multi-var*”. Inside *while-body*, *multi-var* with the i’th value can be used as single-value variable. The *while-body* may contain a mixture of textual content and XVCL commands.
comments	
Syntax	<!-- comment -->
Description	Text enclosed between <!-- --> is considered a comment. Comments may spread over multiple lines.