Proceedings of the

ELISA workshop

Evolution of Large-scale Industrial Software Evolution

Tuesday, 23 September 2003

Royal Netherlands Academy of Arts and Sciences
Amsterdam, The Netherlands
co-located with ICSM 2003

Organised by: Tom Mens, Juan F. Ramil, Michael W. Godfrey, Brian Down

An official activity of the ESF RELEASE research network

TABLE OF CONTENTS

Full papers

Short papers

Rationale Support for Maintenance of Large Scale Systems

Janet E. Burge and David C. Brown
Worcester Polytechnic Institute
Computer Science Department
Worcester, Massachusetts, 01609, USA
jburge@cs.wpi.edu, dcb@cs.wpi.edu

August 5, 2003

Abstract

Software maintenance has long been one of the most difficult and expensive phases of the software life-cycle. Maintenance is especially difficult for large-scale systems. The more code involved, the larger the chance that there may be unexpected interactions that may cause problems when updates and corrections are made during maintenance. The large number of developers who were probably involved at various points in the system's creation means that it is likely to be difficult to answer questions about the *intent* behind the design and implementation decisions. The designer's, or developer's, intent can be captured as their Design Rationale. Unlike standard design documentation, which is a description of the final design, Design Rationale (DR) offers more: not only the decisions, but also the reasons behind each decision, including its justification, other alternatives considered, and argumentation leading to the decision.

To drive and evaluate our research into using rationale for software maintenance, we are developing the SEURAT (Software Engineering Using RATionale) system to support the software maintainer. This system will present the relevant DR when required and allow entry of new rationale for the modifications. The new DR will then be verified against the existing DR to check for inconsistencies. There are several types of inferences that should be made: structural inferences to ensure that the rationale is complete, evaluation, to ensure that it is based on well-founded arguments, and comparison to rationale collected previously for similar modifications to see if the same reasoning was used.

1 Introduction

1.1 Problem and Motivation

Software maintenance has long been one of the most difficult and expensive phases of the software life-cycle. Maintenance costs can be more than 40 percent of the cost of developing the software in the first place [18]. One reason for this is that the software lifecycle is a long one. Large projects may take years to complete and spend even more time out in the field being used (and maintained). The panic over the "Y2K bug" highlighted the fact that software systems often live on much longer than the original developers intended. Compared to hardware, software is "easy" to modify during maintenance—software maintenance changes can often be more extensive and more frequent than maintenance performed on less mutable systems. The likelihood that these changes may add defects to the system is increased by the fact that the combination of a long life-cycle and the typically high personnel turnover in the software industry increases the probability that the original designer is unlikely to be available for consultation when problems arise.

Maintenance is especially difficult for large-scale systems. The more code involved, the larger the chance that there may be unexpected interactions that may cause problems when updates and corrections are made during maintenance. The large number of developers who were probably involved at various points in the system's creation means that it is even more likely to be difficult to answer questions about the intent behind the design and implementation decisions. This increases the probability that new decisions made during maintenance will conflict with this original intent with adverse consequences.

All these reasons argue for as much support as can be provided during maintenance. Semi-automatic systems, such as Reiss's constraint-based system [27], working on the code, abstracted code, design artifacts, or meta-data, can already provide a lot of support. Design Rationale, however, provides an additional opportunity for assistance during maintenance.

1.2 Approach

The designer's, or developer's, intent can be captured as their Design Rationale. Unlike standard design documentation, which is a description of the final design, Design Rationale (DR) offers more: not only the decisions, but also the reasons behind each decision, including its justification, other alternatives considered, and argumentation leading to the decision [22]. This additional information offers a richer view of both the product and the decision making process by providing the designer's intent behind the decision [28]. This DR would then be available for use by the software maintainer to determine where software changes should be performed and to determine the impact of these changes.

To drive and evaluate our research into using rationale for software maintenance, we are developing the SEURAT (Software Engineering Using RATionale) system to support the software maintainer. This system will present the relevant DR when required and allow entry of new rationale for the modifications. The new DR will then be verified against the existing DR to check for inconsistencies. There are several types of inferences that should be made: structural inferences to ensure that the rationale is complete, evaluation, to ensure that it is based on well-founded arguments, and comparison to rationale collected previously for similar modifications to see if the same reasoning was used. In the latter, the previous rationale could be used as a guide in determining the rationale for the new modification. The system will also propagate, or assist in propagating, any necessary changes to the existing DR as well as alerting the maintainer if the code modifications are the same as those made earlier and then rejected. We have developed the requirements for the SEURAT system based on earlier work on the InfoRat (Inferencing over Rationale) system [5] as well as a preliminary study on using rationale for software maintenance [6].

Our work focuses on the *use* of DR, in order to explore and evaluate its potential. It is *not* intended to be a fully-fledged programming and maintenance support system.

In this paper, we first describe related work in this area (Section 2), the representation for our software design rationale (Section 3), and the argument ontology developed to support semantic inferencing (Section 4). This is then followed by a discussion of the inferencing supported by the system (Section 5), and the proposed system (Section 6). We then conclude with a summary of our approach (Section 7).

2 Related Work

Design rationale research has typically focused on three aspects of rationale: DR representation, DR capture, and DR use. Design Rationale representations vary from informal representations such as audio or video tapes, or transcripts, to formal representations such as rules embedded in an expert system [11]. A compromise is to store information in a semi-formal representation that provides some computation power but is still understandable by the human providing or using the information.

Semi-formal representations are often used to represent argumentation. Argumentation notations provide a structure to indicate what decisions were made (or not made) and the reasons for and against them. Argumentation formats date back to Toulmin's representation [29] of datums, claims, warrants, backings and rebuttals. This is the origin of most argumentation representations. More recent argumentation formats include Questions, Options, and Criteria (QOC) [23], Issue Based Information System (IBIS) [11], and Decision Representation Language (DRL) [21]. Each argumentation format has its own set of terms but the basic goal is to represent the decisions made, the possible alternatives for each decision, and the arguments for and against each alternative.

Argumentation has been used in rationale representations that were created specifically for software design. Potts and Bruns [26] created a model of generic elements in software design rationale that was then extended by Lee [21] in creating his Decision Representation Language (DRL), the language used in SIBYL [20]. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale [25]. This system is used to select design patterns based on the designers intent and other constraints.

There are also many different ways to capture DR. One approach is to build the rationale capture into a system used for the design task. Active Design Documents (ADD), a system that does routine, parametric design [15], uses rationale already built into a knowledge base and associates it with the user's decisions. Some systems capture DR by integrating the system into an existing design tool. This is done by M-LAP (Machine-Learning Apprentice System) [3]. In M-LAP, user actions are recorded at a low level and formed into useful sequences using machine-learning techniques. This is also done in the RCF (Rationale Construction Framework) [24]. RCF uses its theory of design metaphors to interpret actions recorded in a CAD tool and convert them into a history of the design process.

Capturing rationale is often expensive and time consuming and can only be justified if there are compelling uses for the rationale. Systems such as JANUS (Fischer, et. al., 1995), critique the design and provide the designers with rationale to support the criticism. Others, such as SIBYL [20], verify the design by checking that the rationale behind the decisions is complete. C-Re-CS [19] performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. InfoRat [5] performs inferencing over the rationale to verify that the rationale is complete and consistent, and to also evaluate that decisions made were well supported.

There has also been work on using design rationale in software design. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale [25]. Co-MoKit [12] uses a software process model to obtain design decisions and causal dependencies between them. WinWin [1] aims at coordinating decision-making activities made by various "stakeholders" in the software development process. Bose [2] defined an ontology for the decision rationale needed to maintain the decision structure. The goal was to model the decision rationale in order to support decision maintenance by allowing the system to determine the impact of a change and propagate modification effects. Chung, etc. al. [8] developed an NFR Framework which uses non-functional requirements to drive the software design process, producing the design and its rationale.

3 Rationale Representation

We have generated an initial rationale representation, RATSpeak. We have chosen to represent our rationale in a semi-structured argumentation format because we feel that argumentation is the best means for expressing the advantages and disadvantages of the different design options considered.

We chose to base RATSpeak on DRL [21] because DRL is the most comprehensive of the rationale languages. Even so, it was necessary to make some change because DRL did not provide a sufficiently

explicit representation of some types of argumentation (such as indicating if an argument was for or against an alternative).

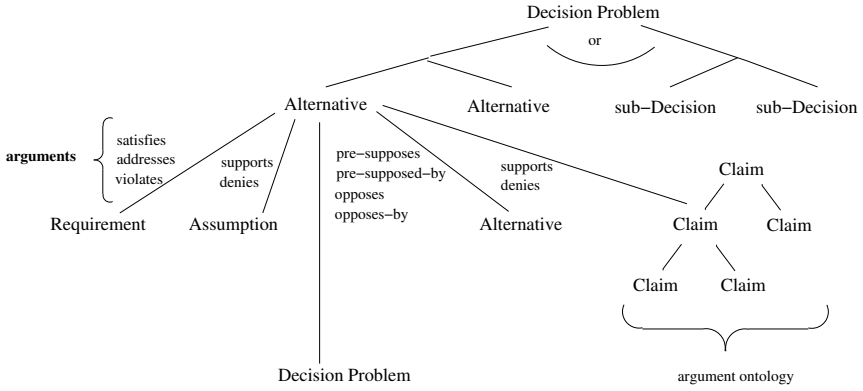


Figure 1: RATSpeak Argumentation Structure

RATSpeak uses the following elements as part of the rationale:

- *Requirements* - these are the requirements, both functional and non-functional. These can either be represented explicitly in the rationale or stored as pointers to requirements stored in a requirements document or database. For the purposes of our examples, we will show them as part of the rationale. Requirements serve two purposes in RATSpeak, one is as the basis of arguments for and against alternatives. This allows RATSpeak to capture cases where an alternative supports or violates a requirement. The other purpose is so that the rationale for the requirements themselves can be captured.

- *Decision Problems* - these are the decisions that must be made as part of the development process. They tend to be expressed in the form of questions.

- *Questions* - these are questions that need to be answered before the answer to the decision problem can be defined. These can be procedures or programs that need to be run or simple requests for information. While questions are not a standard argumentation concept, they can augment the argumentation by specifying the source of the information used to make the decisions, which would be very useful during software maintenance.

- *Alternatives* - these are alternative solutions to the decision problems. Each alternative will have a status that indicates if it is accepted, rejected, or pending.

- *Arguments* - these are the arguments for and against the proposed alternatives. They can either be requirements (i.e., an alternative is good or bad because of its relationship to a requirement), claims about the alternative, assumptions that are reasons for or against choosing an alternative, or relationships between alternatives (indicating dependencies or conflicts). Each argument is given an *amount* (how much the argument applies to the alternative, i.e., how flexible, how expensive) and an *importance* (how important the argument is to the overall system or the specific decision).

- *Claims* - these are reasons why an alternative is good or bad. Each claim maps to an entry in an *Argument Ontology* of common arguments for and against software design decisions. Each claim also indicates what direction it is in for that argument. For example, a claim may state that a choice is NOT safe or that an alternative IS flexible. This allows claims to be stated as either positive or negative assertions.

- *Assumptions* - these are similar to claims except that their truth is in doubt. Assumptions do not map to items in the Argument Ontology.
- *Argument Ontology* - this is a hierarchy of common argument types that serve as types of claims that can be used in the system. These are used to provide the common vocabulary required for semantic inferencing.
- *Background Knowledge* - this contains Tradeoffs and Co-Occurrence Relationships that give relationships between different arguments in the Argument Ontology. This is not considered part of the argumentation but is used to check the rationale for any violations of these relationships.

Figure 2 shows the relationships between the different rationale entities.

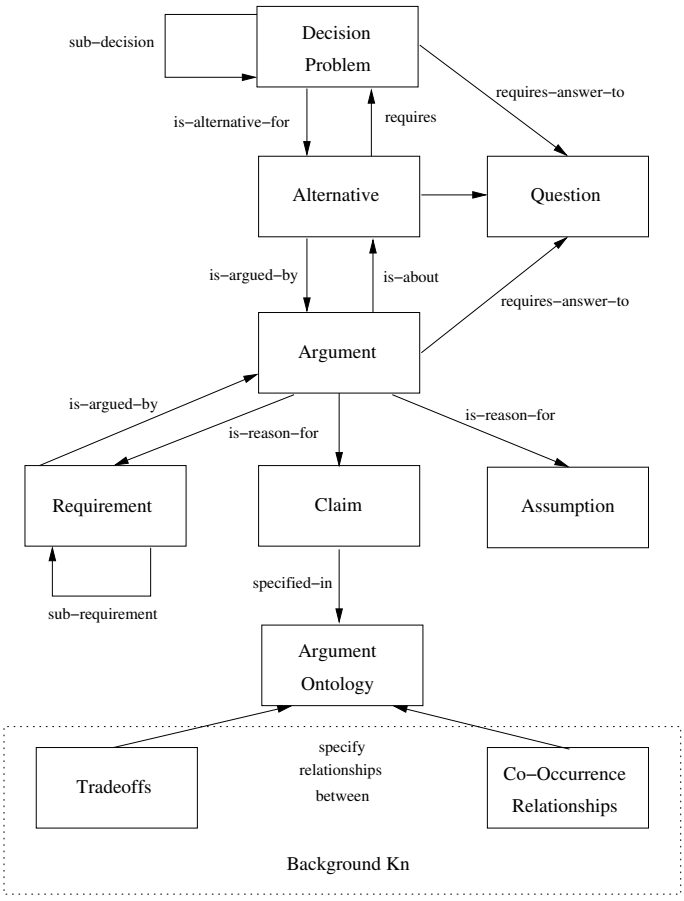


Figure 2: Relationship Between Rationale Entities

4 Argument Ontology

One key element in the RATSpeak representation is the Argument Ontology. Our work on InfoRat showed the importance of providing a common vocabulary to support inferencing over the content of the rationale as well as its structure. To support this, we have developed an ontology of reasons for choosing one design alternative over another. This ontology forms a hierarchy of terms with abstract reasons at the root and increasingly detailed reasons out towards the leaves.

RATSpeak provides the ability to express several different types of arguments for and against alternatives. One type of argument is whether an alternative satisfies or violates a requirement. Other arguments refer to assumptions made or dependencies between alternatives. A fourth type of argument involves claims that an alternative supports or denies a Non-Functional Requirement (NFR). These NFRs, also known as "ilities" [13] or quality requirements, refer to overall qualities of the resulting system, as opposed to functional requirements, which refer to specific functionality. As we describe in [7], the distinction between functional and non-functional is often a matter of context. RATSpeak also allows NFRs to be represented as explicit requirements.

There have been many ways that NFRs have been organized. CMU's Quality Measures Taxonomy [10] organizes quality measures into Needs Satisfaction Measures, Performance Measures, Maintenance Measures, Adaptive Measures, and Organizational Measures. Bruegge and Dutoit [4] break a set of design goals into five groups: performance, dependability, cost, maintenance, and end user criteria. Chung, et. al. [8] provide an un-ordered list of NFRs as well as specific criteria for performance and auditing.

For the RATSpeak argument ontology, we took a bottom-up approach by looking at what characteristics a system could have that would support the different types of software qualities. This involved reviewing literature on the various quality categories to look for how a software system might choose to address these qualities. The aim was to go beyond the idea of design goals or quality measures to look at how these qualities might be achieved by a software system. In maintenance, the maintainers are more likely to be looking at the lower-level decisions and will need specific reasons why these decisions contribute to a desired quality of the overall system. It is probable that decisions made at the implementation level are likely to correspond to detailed reasons in the ontology, while higher level decisions are more likely to use reasons at the more abstract levels.

After determining a list of detailed reasons for choosing one alternative over another, an Affinity Diagram [17] was used to cluster similar reasons into categories. These categories were then combined again. The more abstract levels of the hierarchy were based on a combination of the NFR organization schemes listed earlier (the CMU taxonomy, and Bruegge and Dutoit's design goals). Also, NFRs from the Chung list were used to fill in gaps in the ontology.

Figure 3 shows the first two levels of the Argument Ontology.

```
Affordability Criteria
        Development Cost
        Deployment Cost
        Operating Cost
        Maintenance Cost
        Upgrade Cost
        Administration Cost

Adaptability Criteria
        Extensibility
        Modifiability
        Adaptability
        Portability
        Scalability
        Reusability
        Interoperability

Maintainability Criteria
        Readability
        Supportability

Dependability Criteria
        Security
        Robustness
        Fault Tolerance
        Reliability
        Safety
        Availability

End User Criteria
        Usability
        Integrity

Needs Satisfaction Criteria
        Verifiability
        Traceability

Performance Criteria
        Response Time and Throughput
        Memory Efficiency
        Resource Utilization
```

Figure 3: Top Levels of the Argument Ontology

Each of these criteria then have sub-criteria at increasingly more detailed levels. As an example, Figure 4 shows the sub-criteria for Usability. The ontology terms are worded to be part of an argument: i.e., "<*alternative*> is a good choice because it <*ontology entry*>" where <*ontology entry*> includes the appropriate verb (supports, provides, etc.) for the argument, e.g., "reduces development time". The SEURAT system will be designed so that this ontology will be easily extensible by the user to incorporate additional arguments that may be missing from the ontology. With use, the ontology will continue to be augmented and will become more complete over time. It is possible to add deeper levels to the hierarchy but that will make it more time consuming for the developer to find the appropriate item when adding rationale. Hence ontology depth is a tradeoff that must be made.

The ontology is also intended to be easily extended to incorporate domain-specific arguments that will apply to the system under development. The arguments in SEURAT, including those given here, are only a starting point.

```
Increases Physical Ease of Use       Increases Recoverability
   {provides | supports} effective use  supports undo of user actions
      of screen real-estate             corrects user errors
   minimizes keystrokes
   {provides | supports} increased   Increases Acceptability
     visual contrast                    increases aesthetic value
   is easy to read                    avoids offensiveness

Increases Cognitive Ease of Use      Provides User Customization
   provides reasonable default values   {provides | supports} customization
   provides user guidance               supports different levels of user
   {encourages | supports} direct          expertise
     manipulation
   minimizes memory load on the user Supports Internationalization
   provides feedback                    reduces cultural dependencies
   {conforms to | utilizes} user        supports internationalization
     experience
   increases visibility of function  Increases Accessibility
     to users                           supports visual accessibility
   uses predictable sequences of        supports auditory accessibility
      actions                           supports mobility accessibility
   increases intuitiveness              supports cognitive accessibility
   {provides | supports} an appropriate
     metaphor
```

Figure 4: Usability Arguments

Similar hierarchies have been developed for the remainder of the categories in Figure 3. One thing to note is that it is not a strict hierarchy—there are many cases where items contributing toward one quality also apply to another. One example of this is the strong relationship between scalability and performance—throughput and memory use, while primarily thought of as performance aspects, also impact the scalability of the system. In this case, and others that are similar, items will belong to more than one category.

The argument ontology also includes a user-modifiable default importance for each item. These are present so that SEURAT users can specify this information once if the importance value should hold for an entire system. The importance is used in weighing the different arguments during inferencing. The importance can be overridden for each claim or argument but is stored with the ontology to allow this information to be global if desired.

Other relationships that need to be captured are tradeoffs and co-occurrences. These are cases where two items in the ontology often either oppose each other in arguments or support each other in arguments.

RATSpeak captures these as background knowledge stored as part of the rationale. This background knowledge refers to the items in the argument ontology and stores the relationships between them.

5 Inferencing

Design Rationale is very useful even if it is only used in the traditional way as a form of documentation that provides extra insight into the designer's decision-making process. DR can provide even more useful information about the design and modifications made to the design if there is a way to perform inferences over it. Due to the nature of DR, the results may be in the form of warnings or questions (as opposed to conclusions) that help the maintainer act carefully and consistently. In the following sections we describe a number of different inferences that could be performed over rationale that was structured using the RATSpeak representation.

There are two types of inferences that can be performed. Syntactic inferences are those that are concerned mostly with the *structure* of the rationale. They look for information that is missing. Semantic inferences require looking into the *content* of the rationale. The SEURAT system will include the following syntactic inferences:

- Checking for selected alternatives with no supporting arguments;
- Checking for selected alternatives with more arguments against than for;
- Checking for decisions where no alternatives were selected;
- Checking for decisions where one alternative has more arguments than others (may indicate bias or missing information).

Many of these inferences have been implemented as CLIPS [9] rules. Figure 5 shows a set of rules that work together to check for selected alternatives with no supporting arguments.

Semantic inference was explored by the InfoRat [5] system, which used a common vocabulary of reasons so that the content of the arguments could be compared. SEURAT will support the following semantic inferences:

- Checking if the best supported alternative was not selected (based on the importance of the arguments given);
- Checking if contradictory arguments were used (the same criteria used for and against an alternative);
- Checking consistency of argument abstraction (were some alternatives argued with more or less detailed criteria);
- Checking abstraction levels;
- Checking for selected alternatives that violate requirements;
- Checking for requirements that were not satisfied or addressed;
- Checking for violations of tradeoffs and co-occurrences captured in the background knowledge;
- Reporting statistical information of frequency of specific arguments.

```
; favorable arguments
(defrule checkFavorable
   (alternative (name ?a))
   (argument (name ?r))
   (argues (argument ?r) (alternative ?a))
   ( or (argument (name ?r) (argtype supports))
        (or (argument (name ?r) (argtype satisfies))
            (argument (name ?r) (argtype addresses))))
=> (assert (favorable ?r ?a)))

; check for presupposed arguments
(defrule checkPresupposed
   (argument (name ?r))
   (alternative (name ?a))
   (argues (argument ?r) (alternative ?a))
   (argument (name ?r) (argtype presupposed)(alt ?a2))
   (alternative (name ?a2) (status selected))
=> (assert (presupposed ?a)))

; if *any* arguments are favorable we want this to return false
(defrule checkNotSupported
   (alternative (name ?a) (status selected))
   (not (presupposed ?a))
   (not (favorable ?r ?a))
=> (assert (notsupported ?a)))
```

Figure 5: Rules Checking for Unsupported Alternatives

```
;requirements traceability - violates
(defrule checkViolatedRequirements
   (requirement (name ?q))
   (decision (name ?d))
   (alternativeFor (alternative ?a) (decision ?d))
   (argues (argument ?r) (alternative ?a))
   (argument (name ?r) (req ?q) (argtype violates))
   (alternative (name ?a) (status selected))
=> (assert (violatesReq ?q ?d ?a)))
```

Figure 6: Rule Checking for Violated Requirements

Most of these inferences have been implemented as CLIPS rules. Figure 6 shows a rule that looks for requirements that were violated.

In addition to the inferencing, SEURAT will also support querying information about the rationale. This will let the maintainer see what portions of the design and/or implementation affect which requirements, functional or otherwise.

6 SEURAT: Software Engineering Using RATionale

We are in the process of building the SEURAT system in order to demonstrate how rationale can be used in software maintenance. SEURAT will use commercial and open source development tools to store the design and implementation. These tools will then be augmented with the ability to store, view, and inference over the rationale. Figure 7 shows a diagram of the proposed SEURAT system.

This diagram shows that the main user interaction will be through two off-the-shelf tools: an Interactive Development Environment (IDE) that is used for writing and building source code and a UML design tool used in capturing the software design. SEURAT will be initially aimed at Java development and will use

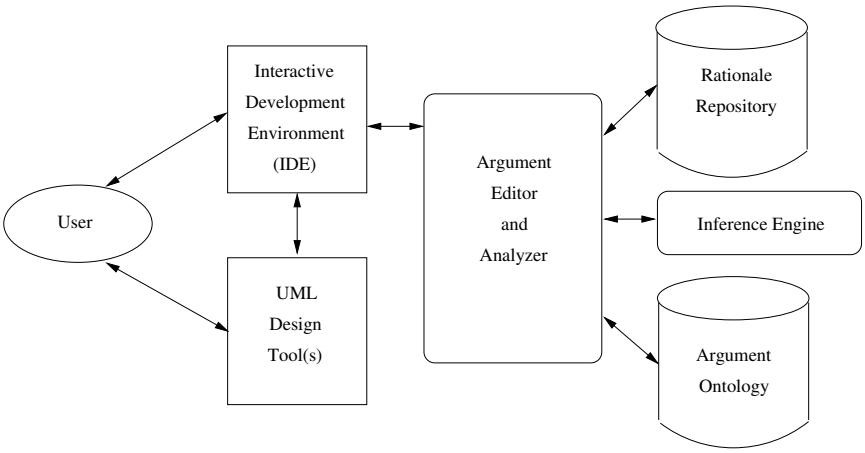


Figure 7: Proposed SEURAT System

the Eclipse IDE [16]. Eclipse is Open Source and can be extended to integrate with a context sensitive Argument Editor and Analyzer. There are a number of UML design tools that interface with Eclipse. One of those will be selected so that it is possible to enter and display rationale as notations to UML diagrams.

The inferencing will take place upon user command and will be performed by the Inference Engine. This will be developed using the Java Expert System Shell (JESS) [14], which is a CLIPS-based expert system shell. The CLIPS inference rules already developed will be used here, as well as any additional ones needed.

The rationale will be stored in a Rationale Repository. This will be done using a relational database. The Argument Ontology will be stored in a similar database. SEURAT will start with a general ontology that can be extended for the needs of a particular software system.

The SEURAT system will support a variety of uses for the rationale. For example, assume a software maintainer is working on a deployed system that was initially developed to support a relatively small number of users. This information appears in the rationale in two ways: an assumption that there would be only a few users at a time and the importance given to the scalability criteria in the argument ontology. SEURAT would be used to decrease the plausibility of the assumption and increase the importance of scalability as an argument. SEURAT would then re-evaluate the decisions made and inform the user if this results in selected alternatives that are no longer the best-supported. In addition, the maintainer could use SEURAT to look for where the scalability argument and assumption appear in the rationale as clues to where the software design and implementation need to be changed to allow additional users.

7 Summary and Conclusions

Several steps have been taken toward the development of SEURAT, the Software Engineering Using RATionale system. These include the development of a rationale representation, an argument ontology, and a preliminary set of inferences that can be performed over the rationale in order to support software maintenance. The SEURAT system will integrate the capture and use of rationale with development tools that could be used by the software maintainer to make modifications to the software.

Such a tool would be invaluable during the maintenance of large-scale software systems and will complement other programming and maintenance environments. One of the chief difficulties in maintaining a large system is knowing the reasons behind the choices made by the developers during design and implementation. The lack of this knowledge makes it difficult for the maintainer to do their job and increases

the risk that defects are introduced during maintenance. The presence of rationale would serve as a form of "corporate memory" by capturing design information that would be lost if the developers left the company or if they were inaccessible to the maintainers [25]. Further support will be provided by giving the maintainer the ability to inference over the rationale to both view the reasons for the choices, evaluate the choices made, and determine the potential impact of new decisions.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful feedback on the paper and Mikhail Mikhailov, Tom Mens, and Juan Ramil for Latex assistance.

References

[1] B. Boehm and P. Bose. A Collaborative Spiral Software Process Model Based on Theory W. In *Proc. 3rd International Conf. on the Software Process*, pages 59–68, Reston, VA, 1994.

[2] P. Bose. A Model for Decision Maintenance in the WinWin Collaboration Framework. In *Proc. of the Conf. on Knowledge-based Software Engineering*, pages 105–113, Boston, MA, 1995.

[3] M. Brandish, M. Hague, and A. Taleb-Bendiab. M-LAP: A Machine Learning Apprentice Agent for Computer Supported Design. In *Artificial Intelligence in Design Workshop Notes on Machine Learning in Design*, Stanford, CA, 1996.

[4] D. Bruegge and A. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice Hall, 2000.

[5] J.E. Burge and D.C. Brown. Inferencing Over Design Rationale. In J. Gero, editor, *Artificial Intelligence in Design '00*, pages 611–629. Kluwer Academic Publishers, 2000.

[6] J.E. Burge and D.C. Brown. Discovering a Research Agenda for Using Design Rationale in Software Maintenance. Technical Report WPI-CS-TR-02-03, WPI, 2002.

[7] J.E. Burge and D.C. Brown. NFRs: Fact or Fiction? Technical Report WPI-CS-TR-02-01, WPI, 2002.

[8] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[9] CLIPS. *CLIPS Reference Manual Volume I: Basic Programming Guide, Version 6.10*. 1998. http://www.ghgcorp.com/clips/download/documentation.

[10] CMU. Quality measures taxonomy. Technical report, CMU, 2002. http://www.sei.cmu.edu/str/taxonomies/view_qm.html.

[11] J. Conklin and K. Burgess-Yakemovic. A Process-oriented Approach to Design Rationale. In T. Moran and J. Carroll, editors, *Design Rationale Concepts, Techniques, and Use*, pages 293–328. Lawrence Erlbaum Associates, 1995.

[12] B. Dellen, K. Kohler, and F. Maurer. Integrating Software Process Models and Design Rationales. In *Proc. of the Conf. on Knowledge-based Software Engineering*, pages 84–93, Syracuse, NY, 1996.

[13] R.E. Filman. Achieving Ilities. In *Proc. of the Workshop on Compositional Software Architectures*, Monterey, CA, USA, 1998.

[14] E.J. Friedman-Hill. *Jess, The Java Expert System Shell*. Sandia National Laboratories, Livermore, CA, 1998.

[15] A. Garcia, H. Howard, and M. Stefik. Active Design Documents: A New Approach for Supporting Documentation in Preliminary Routine Design. Technical Report 82, Stanford University Center for Integrated Facility Engineering, 1993.

[16] IBM. *Eclipse Technical Platfom Overview*. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[17] K. Jiro. *KJ Method: A Scientific Approach to Problem Solving*. Tokyo: Kawakita Research Institute, 2000.

[18] F. P. Brooks Jr. *The Mythical Man-Month*. Addison Wesley, 1995.

[19] M. Klein. An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture. *Concurrent Engineering Research and Applications*, pages 37–46, 1997.

[20] J. Lee. SIBYL: A Qualitative Design Management System. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 104–133. MIT Press, 1990.

[21] J. Lee. Extending the Potts and Bruns Model for Recording Design Rationale. In *Proc. of the 13th International Conf. on Software Engineering*, pages 114–125, Austin, TX, 1991.

[22] J. Lee. Design Rationale Systems: Understanding the Issues. *IEEE Expert*, 12(3):78–85, 1997.

[23] A. MacLean, R.M. Young, V. Bellotti, and T.P. Moran. Questions, Options and Criteria: Elements of Design Space Analysis. In T. Moran and J. Carroll, editors, *Design Rationale Concepts, Techniques, and Use*, pages 201–251. Lawrence Erlbaum Associates, 1995.

[24] K. Myers, N. Zumel, and P. Garcia. Automated Capture of Rationale for the Detailed Design Process. In *Proc. of the 11th National Conf. on Innovative Applications of Artificial Intelligence*, pages 876–883, Menlo Park, CA, 1999.

[25] F. Pena-Mora and S. Vadhavkar. Augmenting Design Patterns with Design Rationale. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, pages 93–108, 1996.

[26] C. Potts and G. Bruns. Recording the Reasons for Design Decisions. In *Proc. of the International Conf. on Software Engineering*, pages 418–427, Singapore, 1988.

[27] S.P. Reiss. Constraining Software Evolution. In *Proc. of the International Conference on Software Maintenance*, pages 162–171, Montreal, Quebec, Canada, 2002.

[28] S. Sim and A. Duffy. A New Perspective to Design Intent and Design Rationale. In *Artificial Intelligence in Design Workshop Notes for Representing and Using Design Rationale*, pages 4–12, Lausanne, Switzerland, 1994.

[29] S. Toumlin. *The Uses of Argument*. Cambridge University Press, 1958.

Evolutionary Product Line Modelling

Serguei Roubtsov*
VTT Electronics
VTT Electronics, Kaitovayla 1,
P.O.Box 1100
FIN-90571 Oulu, Finland
ext-Serguei.Roubtsov@vtt.fi

Ella Roubtsova
Eindhoven University of Technology
Den Dolech 2, P.O.Box 513
5600 MB The Netherlands
E.Roubtsova@tue.nl

Pekka Abrahamsson†
VTT Electronics
VTT Electronics, Kaitovayla 1,
P.O.Box 1100
FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

ABSTRACT

A traditional product line approach struggles with complexity and weak evolution support. We propose an evolutionary software product line modelling approach based on controllable inheritance of product line members specifications. Instead of a predefined product line architecture we use hierarchies of implemented product specifications accompanied by correctness control of product model transformations. An industrial case study from the embedded systems domain demonstrating a modelling technique is provided. The approach is supported by an appropriate tool prototype.

1. INTRODUCTION

The product line approach is an approach to software reuse. In large-scale industrial systems it is used, for example, in embedded systems domain. Embedded software product lines such as consumers electronics applications are usually characterized by a huge variety of slightly different product line members [18].

The mainstream of approaches to software product line (SPL) development [8, 4] applies different diversity management techniques to a generic SPL architecture. This allows a designer to produce new products reusing common SPL assets [10] within the boundaries of such a generic architecture. This approach is robust but also complicated and not flexible enough in terms of evolution support.

On the other hand, a component-based development approach has its own worth in the SPL area [17, 5]. This approach employs composition of reusable components as a basis for product population [19] development. However, in the absence of a reference SPL architecture, the main advantage of the product line approach, i.e. controlled variability, may be damaged.

We propose an SPL modelling method that provides inheritance of implemented product line members model specifications accompanied by correctness control of model transformations. The method considers inheritance of product behaviour specifications as inheritance of processes [2, 22]. The method combines the flexibility of component-based approaches with the rigorous correctness of architecture-based techniques. As a result, a designer obtains an instrument that allows him to model new product line members quickly introducing new required functionality and avoiding design bags.

The rest of the paper is organized as follows. Section 2 provides a brief discussion about existing SPL approaches and raises the relevant problems. Section 3 describes a case study from the domain of embedded systems. Section 4 explains our method and provides corresponding illustrations using the case study. Section 5 describes the tool prototype, which has been developed to support our method. The paper is concluded in Section 6.

2. SOFTWARE PRODUCT LINES: STATE-OF-THE-ART APPROACHES AND PROBLEMS

Software product lines traditionally employ a top-down architecture -based methodology of software system development [8, 10, 4, 14, 9]. It starts by choosing a set of products comprising a product line and then proceeds by identifying what requirements are common to all products (commonalities) and what product features make them different (variabilities). On the basis of requirements analysis a common product line architecture and a set of reusable components are designed and implemented. Finally, actual products are derived from these shared assets [4]. Commonalities between SPL members are captured by a generic architecture. Variabilities are usually introduced into this architecture by means of so-called variation points [6], which imply unresolved diversity in the generic and component architectures that should be explicitly introduced and bound into a concrete product during possibly latest phases of product line

*The work of S.A. Roubtsov is supported by The European Economic Interest Grouping ERCIM (European Research Consortium for Informatics and Mathematics.
†The research is carried on within VTT Electronics Agile Software Technologies project: http://agile.vtt.fi.

members development [6] (Figure 1).

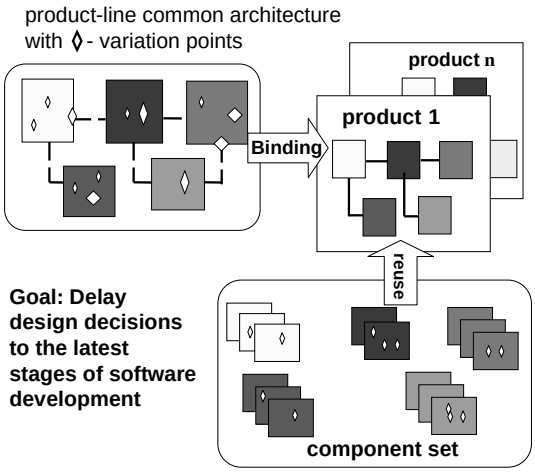


Figure 1: Traditional SPL modelling process.

So, a common SPL architecture with variability management fulfils a double role. Firstly, it provides the *reference of integrity* for SPL components reuse. Secondly, the diversity of all product line members, existent or future, should correspond to the variability already implicit in such a generic architecture. The SPL architecture should provide *correctness* of product modifications.

However, there are some disadvantages of such an architecture -driven [19] approach.

The first problem is complexity. The entire development process is divided into two concurrent parts - domain engineering for reusable SPL assets and application engineering for product line members [14]. SPL development and maintenance give rise to a lot of related tasks, which have to be solved coherently [8, 4]. Among others design of a reusable architecture is an especially complicated problem. How much commonality and variability should be introduced into a common SPL architecture? It has to be somewhat between minimal reuse (common requirements only) and maximal reuse (all requirements, both common and different). The more variability is introduced into the architecture, the more benefits of reuse should be expected. However, design of such a flexible architecture meets a truly challenge [10, 4, 3].

The second problem is evolution support [25]. Requirements are changed, technology is improved. How can we predict the features and, therefore, the architectures of future product line members? Even architecture itself suffers from erosion during a software product evolution process. Research [12] shows how seemingly robust design decisions taken early in the evolution of a single product may conflict with requirements that need to be implemented later in the evolution. For product lines the problem increases immensely (e.g., [27]).

The impact of above mentioned problems is high cost of wrong architectural design decisions.

The alternative software reuse approach is an evolutionary component-based software development process [26]. In the SPL domain it is a product population approach [17, 19, 18, 5]. That approach uses lightweight [17] common architecture and implements software component modifications and component compositions instead of architecture-based variability management (e.q., [18]).

The benefits of evolutionary approaches are explicit. An SPL grows when new product line members appear. A design process is flexible and incremental. Similar already implemented products are reused to introduce the extensions, which are required by a new product. However, in the absence of a fixed common architecture the problems of SPL integrity and product line members design correctness rise sharply. Component modification and composition rules are static, they do not guarantee that the entire system behaviour comprises the behaviour of composition parts in a correct manner. The evolutionary approach needs a design methodology that can help designers collect useful features of already implemented SPL members and avoid incorrect design decisions while they introduce new product functionality. In addition, SPLs are rather long-lived software projects and need to be supported not only by a reusable component set but also by some joint model to be a reference of integrity.

In order to overcome outlined challenges we propose an evolutionary software product line modelling method based on the inheritance of product line members design specifications and correctness control of model transformations. Each implemented specification can become a predecessor of a new product specification. At the same time, correctness of behavioural inheritance with new extensions should be proved (Figure 2).

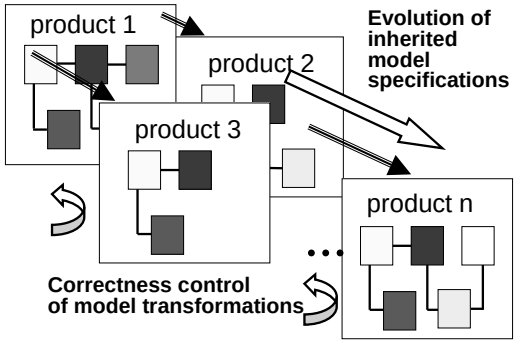


Figure 2: Evolutionary SPL modelling approach.

In our approach design specifications are implemented using UML (Unified Modeling Language) profile with defined inheritance relations on specifications [23]. The profile defines a special type of UML class diagrams, interface-role diagrams, similar to CATALYSIS approach [11]. Component system behaviour is specified in the profile using UML sequence diagrams as it was first introduced in [7]. Process semantics is used as a basis for inheritance relations on

component behavioural specifications [2, 22].

Correctness control is provided by product model transformation checks using inheritance of processes. Applying of backward derivation rules to produce parent product process specifications from inheritor's ones allows a designer to prove correctness of inheritance or to find the points of wrong design decisions.

In [21] the evolutionary SPL modelling technique is used within the traditional architecture-centric SPL development process. Now we advocate our modelling method as a self-sufficient and robust alternative to the traditional one. The previous theoretical results are extended by the notion of a product process graph. The notion of inheritance of product line members specifications is defined on the basis of a process graph definition. In this paper we also discuss the application of our method.

3. CASE STUDY: SCIENTIFIC SILICON ARRAY X-RAY SPECTROMETER

We intend to emphasize applicability of our method. Our case study is a product line representation of Scientific Silicon Array X-Ray Spectrometer (SIXA) Control Software [13, 9][1]. This is an onboard satellite system that provides scientific data in two measurement modes [13]: Energy Spectra (EGY) and Single Event Characterization (SEC).

Despite some differences between EGY and SEC measurement realizations there are also a lot of common requirements that makes it possible to regard this case study as an example of an SPL. Following [9] we intend to model three members of SIXA software product line:

∘ stand alone EGY Controller
∘ stand alone SEC Controller
∘ combined EGY and SEC Controller

The key aspects of SPL modelling have to be found in the requirements, both functional and behavioural, to product line members. Let us consider them subsequently.

3.1 Product line members functionality

The SIXA Controller fulfils the following functional requirements [13]:
- it receives measurement programmes from the ground via a satellite computer,
- provides data measurement,
- collects and sends data back.

These requirements to the product line software can be described in terms of four interconnected subsystems [13] realizing main product features:

- *Measurement Control* subsystem. This subsystem provides *Controller Commands* interface with an onboard satellite computer. External control commands and measurement programmes come via this interface.
- *Data Acquisition* subsystem. It executes measurement programmes received via its interface *Control Data Acquisition* from *Measurement Control* subsystem.
- *Data Management* subsystem. It
  - fills its internal buffer with data received from *Data Acquisition* subsystem via interface *Save Data*.
  - sends scientific data back to the ground via *Satellite Computer* interface *Controller Data Response* following commands from *Measurement Control* subsystem via interface *Control File Management*.
- *Satellite Computer* that is regarded as an external system. It uses Spectrometer interface *Controller Commands* and receives scientific data via its own interface *Controller Data Response*.

The described above SIXA spectrometer functionality is common for the entire SPL.

The variability is defined by the different measurement modes that have to be implemented. EGY and SEC modes are realized by different specific *Data Acquisition* subsystems and corresponding interfaces *Control Data Acquisition* and *Save Data*. There is also slightly different organization of a data exchange process with the satellite computer: EGY Controller *Data Management* subsystem sends data to the satellite computer after measurement programme has been fulfilled completely, whereas SEC Controller *Data Management* subsystem can initialize data exchange when its internal buffer is full. So, this subsystem should be able to send such a request to *Satellite Computer*.

EGY and SEC Controller has to provide functionality of each stand alone mode whatever has been chosen by the ground measurement programme.

3.2 Product line members behaviour

The behavioural requirements to the SIXA Spectrometer software are defined by two data observation processes, one process for each observation mode [13]. Both processes comprise two sequential sub-processes: data measurement and data exchange. Using usual algorithmic notation the processes can be described as it is shown in Fig. 3. (We omit a few not significant technical details in order to draw a more clear picture.) Each block in Fig. 3 corresponds to an operation call that is performed by interacting SIXA Controller software subsystems and supported by hardware signals. The blocks above the dashed line (Fig. 3) perform the data measurement sub-processes, the blocks below this line correspond to the data exchange sub-process.

The data exchange sub-process is common for EGY and SEC modes: after sending to the ground the number of blocks with scientific data to be transmitted it performs a cycle of data blocks transmission.

The data measurement sub-processes are partially different. The dark blocks in Fig. 3 depict the steps of the measurement sub-processes which are different for EGY and SEC modes. The EGY measurement sub-process is performed subsequently for each of the predefined observation targets.

[1]We thank Prof. Eila Niemela and Tuomas Ihme from VTT Electronics for sharing the insights into this case study

This corresponds to the external cycle of the algorithm on the left hand side in Fig. 3. The algorithm on the right hand side does not contain this cycle because in SEC measurement mode a single target is observed continuously. For both modes a single target observation cycle lasts until an observation time is expired. However, in SEC mode the observation process can be interrupted when *Buffer Full* message is raised in the system.

The real SIXA spectrometer has more features to be modelled [9], support of a hard disk in SEC mode, for example. However, additional features can become part of future SPL members generations. The case study is enough to give a demonstration of how our method works.

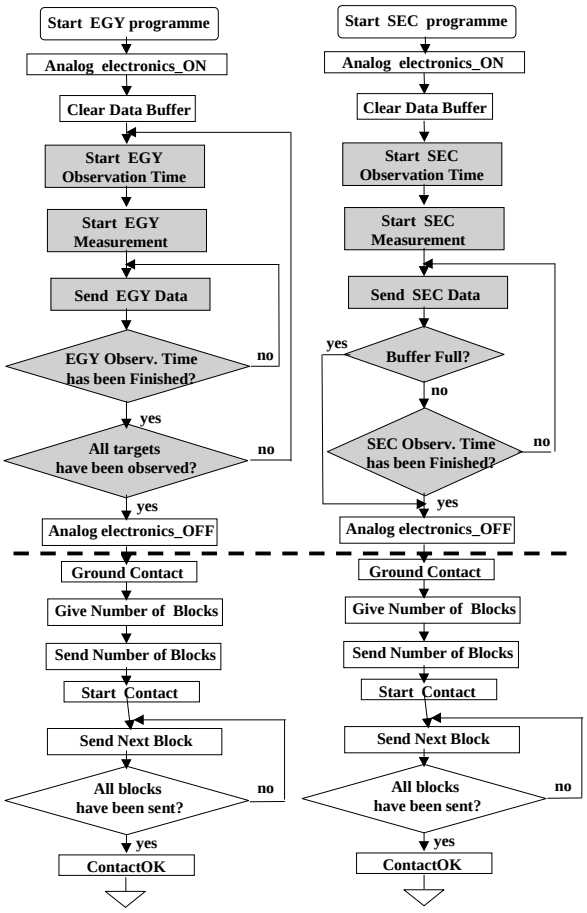


Figure 3: Observation algorithms for SIXA Spectrometer. On the left hand side: EGY mode; on the right hand side: SEC mode; measurement subprocess is above – – – – line; data exchange subprocess is below.

4. EVOLUTIONARY PRODUCT LINE MODELLING METHOD

The method includes two parts: a product model specification and the definition of inheritance of product line members specifications with the derivations rules providing correctness of model transformations.

4.1 Product Model Specification

The *product line member specification* is a pair

$$PrSp = (IR, BS)$$

where IR is an interface-role specification and BS is a behavioural specification.

4.1.1 Interface-role specification

The interface-role specification describes static aspects of product functionality. Roles can *provide* interfaces, which the other roles can *require* [11]. Each such a pair of roles interacting via the interface can model a piece of product functionality, i.e. a product feature [4]. So, product functional requirements can be mapped directly to interface-role specifications.

On the other hand, roles with interfaces are quite similar in nature to product components. Components interact by playing roles. A designer is free to abstract from a concrete component implementation during role modelling [28]. However, one or several interacting roles can be mapped to a product component architecture in such a way that component boundaries should come across the interfaces provided by roles [28, 21].

Interface-role specification is a tuple

$$IR = (R, I, PI, RI, RR), where:$$

- R is a finite set of roles. $R = R_p \cup R_d$, R_p is a subset of roles that provide interfaces; R_d is a subset of roles that require interfaces. The same role can belong to both subsets R_p and R_d.
- I is a finite set of interfaces provided by roles from R_p. Each interface $i \in I$ has finite set of operations OP_i. Each operation $op \in OP_i$ has finite set of result values Res_{op}.
- $PI \subseteq \{(r, i) |\ r \in R_p, i \in I\}$ defines provided relations between roles and interfaces.
- $RI \subseteq \{(r', pi) |\ r' \in R_d, pi \in PI\}$ defines required relations between roles and interfaces. Each role requires a finite set of provided interfaces.
- $RR \subseteq \{(r, r') |\ r, r' \in R\}$ is a set of inheritance relations on the set of roles. These relations are part of inheritance relations between product line members specifications and will be considered later (see section 4.2.1).

The interface-role specification of EGY Controller is shown in Fig. 4. In all specification parts, where EGY Controller specifics has to be introduced, the names have prefix "EGY".

Four roles-providers correspond to four subsystems in the product requirements specification as well as five provided interfaces represent specified earlier (section 3.1) system interfaces.

Provided relations are presented by pairs (*role-provider*, *interface*), for example, (*Satellite Computer*, *IController Data*

Responce). For each such a pair each possible triple (*role-requirer*, *role-provider*, *interface*) represents a required relation, for example, (*EGYData Acquisition*, *EGYData Management*, *ISaved EGYData*) (Fig. 4).

Operation names in Fig. 4 are the same as the names of operations presented by blocks in Fig. 3. We only use a few abbreviations.

We have chosen EGY Controller to be the first product in the product line; hence its specification does not contain inheritance relations.

Roles-requirers (Rd)	Roles-providers (Rp)	Interfaces (I)		
		Names of interfaces	Operations (Op_i)	Result values (Res_{op})
EGY Data Management	Satellite Computer	IController Data Responce	SendNoOf Blocks(integer)	void
			SendNext Block(structure)	void
Satellite Computer	EGY Measurement Control	IController Commands	Analog_ON	void
			Start EGY Observation Time	void
			Finish EGY Observation Time	void
			Analog_OFF	true
			GroundContact	void
			ContactOK	void
EGY Measurement Control	EGY Data Acquisition	IControl EGYData Acquisition	StartEGY Measurement	true
EGY Measurement Control	EGY Data Management	IControl File Management	ClearData	void
			GiveNoOf Blocks	void
			StartContact	void
EGYData Acquisition		ISaved EGYData	SendEGYData (structure)	void

Figure 4: Interface-role specification IR_{EGY} of EGY Controller

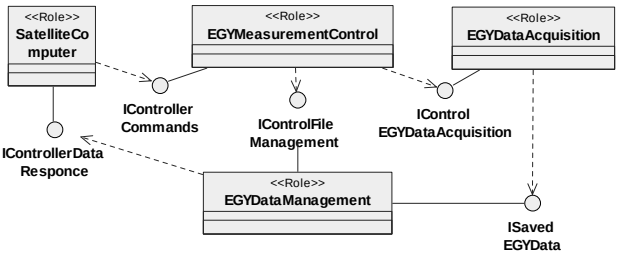


Figure 5: Interface-role diagram for EGY Controller

The interface-role specification is realized in the UML profile [23] and presented by a UML class diagram [16], where roles are UML classes with stereotype ≪Role≫ and interfaces are classes with stereotype ≪Interface≫. Interfaces are depicted by cycles. Provided relations are presented by UML realize-relations between roles and provided interfaces and depicted by solid lines [16]. Required relations are the same as UML dependency relations between roles and required interfaces. A required relation is depicted by a dashed arrow directed from a role to a required interface [16].

The interface-role diagram of EGY Controller is shown in Fig. 5.

4.1.2 Behavioural specification

The behavioural specification describes dynamic aspects of product functionality, i.e. product behaviour. A grain of product behaviour is presented by a *pair of actions* [22]. The first action of the pair is an *operation call*, the second one is an *operation return*. It has to be noticed here that operation calls and returns in the model specification are not the same as ones in the implementation phase: each modelled call and/or return can be implemented by one or several methods (procedures and functions).

An action name for the operation call is $a = r'.r.i.op$, which means "role r' calls operation op of interface i provided by role r ".

An action name for the operation return is $a = r'.r.i.op : res_{op}$, which means "role r returns result res_{op} responding to operation call $a = r'.r.i.op$".

As a result of product IR specification, action set A_{PrSp} is introduced for the entire product specification. To refer to the concrete actions of this set we apply on it a numeric order relation giving natural numbers to all actions:

$$A_{PrSp} = \{a_1, a_2, ...\}$$

The quantity of actions $a_i \in A_{PrSp}$ is defined completely by the quantity of operation calls and returns via required relations $ri \in RI$ between roles $r' \in R_d$ and $r \in R_p$.

Fig. 6 shows the action set for the EGY Controller specification. We omit interface names in action names for convenience. This is possible if operation names are unique for each pair of interacting roles. There are thirteen operation calls and same number of operation returns in this set.

Using action set A_{PrSp} we construct behavioural specification BS of a product line member as a finite *set of sequences* representing product behavioural patterns [22]:

$$BS = \{S_1, S_2, ..., S_n\},$$

where $S_i, \forall i = 1, 2, ..., n$ is a *sequence of actions* $a_j, a_k \in A_{PrSp}, \forall j, k = 1, ..., |A_{PrSp}|$:

$$S_i = \{a_j, a_k, ...\}$$

The last definition means that we can construct behavioural pattern S_i using any action from action set A_{PrSp} any number of times. We apply the restriction that one and only one action representing operation return must appear after (but not necessarily just after) the action that represents the corresponding operation call.

Any sequence S_i can contain any number nested in any depth repeated subsequences or *cycles* [21]. For example,

A_{EGY}={a1,...a26}

a1 - SatelliteComputer.EGYMeasurementControl.Analog_ON
a2 - EGYMeasurementControl.EGYDataManagement.ClearData
a3 - EGYMeasurementControl.EGYDataManagement.ClearData:void
a4 - SatelliteComputer.EGYMeasurementControl.Analog_ON:void
a5 - SatelliteComputer.EGYMeasurementControl.StartEGYObservationTime
a6 - SatelliteComputer.EGYMeasurementControl.StartEGYObservationTime:void
a7 - EGYMeasurementControl.EGYDataAcquisition.StartEGYMeasurement
a8 - EGYDataAcquisition.EGYDataManagement.SendEGYData(structure)
a9 - EGYDataAcquisition.EGYDataManagement.SendEGYData:void
a10 - EGYMeasurementControl.EGYDataAcquisition.StartEGYMeasurement:true
a11 - SatelliteComputer.EGYMeasurementControl.FinishEGYObservationTime
a12 - SatelliteComputer.EGYMeasurementControl.FinishEGYObservationTime:void
a13 - SatelliteComputer.EGYMeasurementControl.Analog_OFF
a14 - SatelliteComputer.EGYMeasurementControl.Analog_OFF:void
a15 - SatelliteComputer.EGYMeasurementControl.GroundContact
a16 - SatelliteComputer.EGYMeasurementControl.GroundContact:void
a17 - EGYMeasurementControl.EGYDataManagement.GiveNoOfBlocks
a18 - EGYDataManagement.SatelliteComputer.SendNoOfBlocks(integer)
a19 - EGYDataManagement.SatelliteComputer.SendNoOfBlocks:void
a20 - EGYMeasurementControl.EGYDataManagement.GiveNoOfBlocks:void
a21 - EGYMeasurementControl.EGYDataManagement.StartContact
a22 - EGYMeasurementControl.EGYDataManagement.StartContact:void
a23 - EGYDataManagement.SatelliteComputer.SendNextBlock(structure)
a24 - EGYDataManagement.SatelliteComputer.SendNextBlock:void
a25 - SatelliteComputer.EGYMeasurementControl.ContactOK
a26 - SatelliteComputer.EGYMeasurementControl.CoontactOK:void

Figure 6: Set of actions A_{EGY} for EGY Controller

sequence:

$$S_i = \{st_1, a_j, ...f_1, a_k, ...st_2, a_m, ...st_3, a_p, ...f_3, a_q, ...f_2, a_n\}$$

contains three cycles, the first cycle goes form a_j to a_k, the second one lasts from a_m to a_n. The third cycle $a_p, ...a_q$ is nested in the second one. Prefix "st," with the number of a cycle denotes the action starting repetition and prefix "f," with the same number denotes the action finishing repetition.

{Si}	Sequence of actions $a_j \in A_{EGY}$
EGYObservation	a1, a2, a3, a4, st_1,a5, a6, st_2,a7, a8, a9, f_2,a10, a11, f_1,a12, a13, a14, a15, a16, a17, a18, a19, a20, a21, a22, st_3,a23, f_3,a24, a25, a26

Figure 7: Behavioural specification BS_{EGY} of EGY Controller

Behaviour of EGY Controller is specified by requirements to the EGY observation process which is described in section 3.2. Using this specification we have designed behavioural specification

$$BS_{EGY} = \{EGYObservation\}$$

containing single sequence $EGYObservation$ (Fig. 7).

The behavioural specification is realized in the UML profile [22] and presented by a set of UML sequence diagrams [16], one diagram for each sequence S_i. The precise definition of a sequence diagram for this UML profile is given in [21].

The sequence diagram for EGY Controller is shown in Fig. 8. This diagram corresponds to the algorithm on the left hand side in Fig. 3.

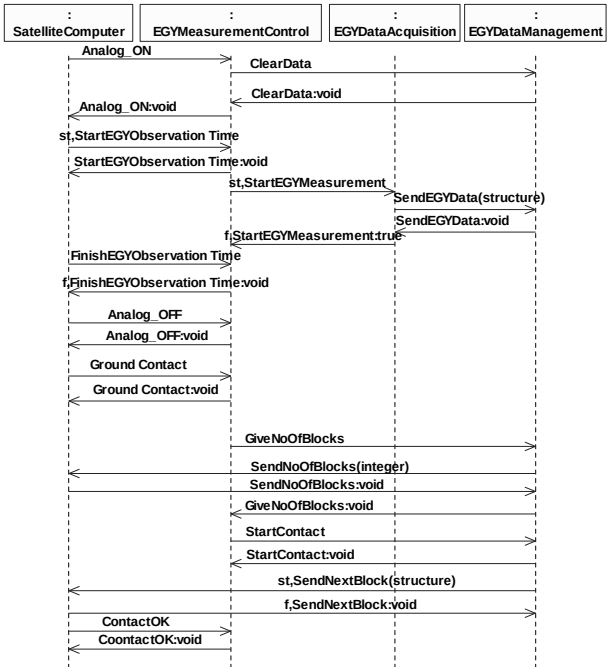


Figure 8: Sequence diagram EGYObservation for EGY Controller

4.2 Inheritance of Product Specifications

We regard inheritance of product line members as inheritance of product behaviour. If, for example, product EGY and SEC Controller inherits product EGY Controller, then it inherits the possibility to observe energy spectra and extends it by the SEC spectra observation facility.

Let us use notation $PrSp_q -\triangleright PrSp_p$ to depict *inheritance* of product $PrSp_q$ from product $PrSp_p$.

In our approach behaviour is presented by product BS specification. So, *product specification $PrSp_q$ inherits product specification $PrSp_p$ if behavioural specification BS_q inherits behavioural specification BS_p.*

Behaviour specification $BS_q = \{S_{1_q}, S_{2_q}, ..., S_{n_q}\}$ *completely inherits* $BS_p = \{S_{1_p}, S_{2_p}, ..., S_{m_p}\}$ *if* $n \geq m$ *and each sequence* S_{i_q} *inherits corresponding sequence* S_{i_p}.

If BS_q inherits a subset of sequences of BS_p we have the case of *partial inheritance.*

Hence, to define the inheritance of product specifications we need to define the inheritance of sequences presenting product behaviour patterns.

Each sequence S_i is defined by set of actions A_{PrSp} and

this set is defined by set RI of required relations on product interface-role specification IR. So, first we need to define inheritance at the level of interface-role specifications.

4.2.1 Inheritance of interface-role specifications

Interface-role specification

$$IR_q = (R^q, I^q, PI^q, RI^q, RR^q)$$

inherits interface-role specification

$$IR_p = (R^p, I^p, PI^p, RI^p, RR^p)$$

if $\exists(r', r) \in RR^q | r' \in R^q, r \in R^p$ and $\neg\exists(r, r') \in RR^p | r' \in R^q, r \in R^p$

In other words, at least one role from IR_q inherits at least one role from IR_p and none of the roles from IR_p inherit roles from IR_q.

If role r' inherits role r: $r' -\triangleright r$, then [22]:

- role-parent r is included in specification IR^q;
- role-child r' inherits all interfaces, provided by role-parent and, hence, all its provided relations;
- role-child r' inherits required relation of role-parent r

  $ri = (r, pi) \in RI^p | pi = (r'', i) \in PI^p, r, r'' \in R^p, i \in I^p$,

  if role-provider r'' is also inherited by specification IR^q.

Inheritance of roles is defined in the UML profile [22] and corresponds to the specialize-relation between UML classes [16]. The relation is shown on the interface-role diagram by a solid line with the triangle end $-\triangleright$ directed from role-child to role-parent [16].

As a result of inheritance, the child interface-role specification comprises two parts:

$$IR_q = (IR_q^{Inh}, IR_q^{New}), where$$

IR_q^{Inh} contains inherited roles, their provided interfaces and provided relations, and, possibly, required relations; IR_q^{New} is a new part, which contains new roles, interacting via new interfaces; it realizes new product functionality and inherits the functionality of a parent product. The only possibility to utilize IR_q^{Inh} specification is to use its roles as parents in inheritance relations with roles from IR_q^{New} specification.

Dealing with our case study a designer should first decide how to order the chain of inheritance:

$$PrSp_{EGYandSEC} -\triangleright PrSp_{SEC} -\triangleright PrSp_{EGY}$$

or

$$PrSp_{SEC} -\triangleright PrSp_{EGYandSEC} -\triangleright PrSp_{EGY}.$$

In other words, what product should inherit EGY Controller first, SEC Controller or EGY and SEC Controller? Despite the fact that a usual composition way dictates the first variant, the second one is the right answer. If the first variant had been chosen, then role *EGYData Acquisition* from IR_{EGY} specification should have been replaced by a new role that fulfils another observation process and EGY data acquisition functionality would have been lost for further utilization.

The first inheritor EGY and SEC Controller has to utilize functionality of EGY Controller and extend it by new SEC Controller functionality. Fig. 9 a) shows inheritance relations between roles from IR_{EGY} and $IR_{EGYandSEC}$. Each role from parent specification IR_{EGY} has a child role. So, all provided interfaces and required relations are inherited by product EGY and SEC Controller. The part IR^{New} of interface-role specification $IR_{EGYandSEC}$ is shown in Fig. 9 b). New functionality is realized by three new interfaces of the child roles.

Child roles (R^q)	Parent roles (R^p)	Inherited interfaces I^p
EGY&SEC SatelComputer	Satellite Computer	IController Data Responce
EGY&SEC MeasureControl	EGY Measurement Control	IController Commands
EGY&SEC Data Acquisition	EGY Data Acquisition	IControl EGYData Acquisition
EGY&SEC Data Management	EGY Data Management	IControl File Management
		ISaved EGYData

a)

Roles-requirers (Rd)	Roles-providers (Rp)	Interfaces (I)		
		Names of interfaces	Operations (Op_i)	Result values (Res_{op})
EGY&SEC Data Manag.	EGY&SEC SatelComputer	IBufferFull	BufferFull	void
EGY&SEC Measurement Control	EGY&SEC Data Acquisition	IControl SECData Acquisition	StartSEC Measurement	true
EGY&SEC MeasureControl	EGY&SEC Data Manag.	ISaved SECData	SendSECData (structure)	void

b)

Figure 9: a) Inheritance of roles and b) IR^{New} part of EGY and SEC Controller specification

The interface-role diagram of EGY and SEC Controller is shown in Fig. 10.

Third product SEC Controller inherits the second one. The interface-role specification of EGY and SEC Controller already contains the functionality required for the third product. A designer is free not to utilized by SEC Controller part of this functionality dealing with EGY data acquisition.

Products-inheritors keep functionality of their predecessors within inherited required relations. However, how can a designer be aware that parent behaviour is not damaged by new design decisions widening or narrowing parent functionality? Such decisions should be supported by product behaviour inheritance modelling, which we consider next.

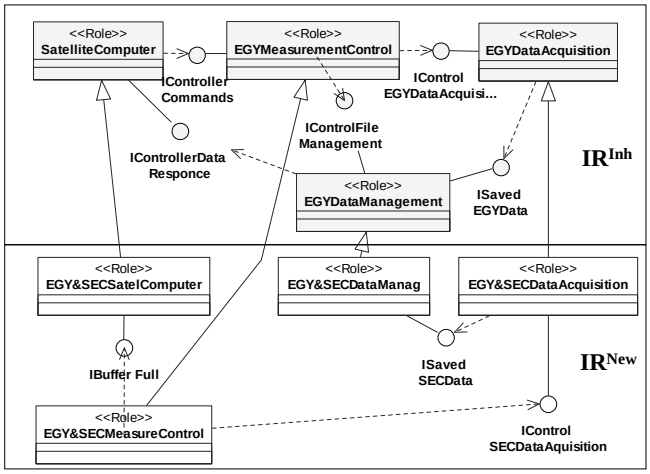


Figure 10: Interface-role diagram of EGY and SEC Controller

4.2.2 *Inheritance of product behaviour*

To define inheritance of product behaviour we apply process semantics on behaviour specifications BS. We use a process semantics of type

$$P = (A, \mathcal{P}, T)\ [2], where:$$

- A is a finite set of actions.

- $\mathcal{P} = \{p, p_1, p_2, ..., p_F\}$ is a finite set of abstract states from initial state p to final state p_F.

- T is a set of transitions. Transition $t \in T$ defines a pair of states (p', p''), such that p'' is reachable from p' as a result of action $a \in A$: $p' \stackrel{a}{\Longrightarrow} p''$.

Considering set of actions A as set A_{PrSp} from a product line member specification, we construct a single *process graph* for the entire product behaviour specification.

Process graph $G_p = (N, E)$ is a directed (cyclic or acyclic) graph [1] in which

- each node n∈N corresponds to the state from $\mathcal{P}$; all nodes, except the root and the final nodes, are unnamed;
- each edge e∈E corresponds to the action from A_{PrSp} and is named as this action;
- the edges may carry the termination label ↓ to one **final** node. This node corresponds to states p_F.
- The process graph has one common root in **start** node that corresponds to initial states p. Each initial state p is considered as a result of *start* action that creates instances of interacting roles [22]. Action *start* is implicit but not shown in the process graph.

Process graph (Fig. 11) keeps parallel branches containing alternatives of sequential, probably cyclic, paths between

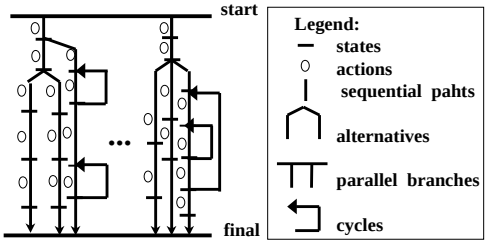


Figure 11: Process graph type

start and **final** nodes. Each such a finite sequential path corresponds to sequence S_i from product behaviour specification BS. Two or several sequences beginning from the same action and containing the same subsequence of actions correspond to a single sequential sub-path in the process graph beginning from **start** node. First two actions that become different for two sequences running the same subpath produce alternative edges in the process graph. Parallel branches model parallel processes. These branches are the alternatives, which begin from **start** node and, in addition, each pair of them corresponds to the subsets of sequences from BS, which have disjoint sets of actions and are not started by same roles [21].

For process graph construction we apply our own algorithm. The algorithm provides control of crosscutting cycles which may be designed by mistake for a single sequence or produced during the process graph construction. The early alternative exit from a cycle body is not prohibited for the process of type P.

The process graph for EGY Controller is shown in Fig. 13 a). It contains the only sequential path that corresponds to single sequence $EGYObservation$ from BS_{EGY} specification.

Behaviour specification $BS_{EGYandSEC}$ for EGY and SEC Controller

$BS_{EGYandSEC} = \{EGYObservation, SECObservation, SECObservationBufferFull\}$

contains three sequences realizing the requirements to the behaviour of second product. These requirements have been described in section 3.2.

Sequence $EGYObservation$ fulfils the same behaviour pattern as the sequence from BS_{EGY} specification. However, inherited required relations are realized by new roles and, therefore, actions from the second product behaviour specification (Fig. 12) have different names, for example,
b1 = EGY&SECSatelComputer.EGY&SECMeasureControl.Analog_ON
instead of
a1 = SatelliteComputer.EGYMeasurementControl.Analog_ON
and so on to actions $b26$ and $a26$ correspondingly (compare Fig.7 and Fig.12).

Sequence $SECObservation$ models the conventional SEC mode measurement process, whereas sequence

SECObservation BufferFull corresponds to Buffer Full event in the system (section 3.2).

{Si}	$BS_{EGY\&SEC}$ $b_j \in A_{EGY\ and\ SEC}$	BS_{SEC} $c_j \in A_{SEC}$
EGY Observation	b1, b2, b3, b4, st_1,b5, b6, st_2,b7, b8, b9, f_2,b10, b11, f_1,b12, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_3, b23, f_3,b24, b25, b26	not inherited
SEC Observation	b1, b2, b3, b4, b27, b28, st_1,b29, b30, b31, f_1,b32, b33, b34, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_2, b23, f_2,b24, b25, b26	c1, c2, c3, c4, c5, c6, st_1,c7, c8, c9, f_1,c10, c11, c12, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, st_2, c23, f_2,c24, c25, c26
SEC Observation BufferFull	b1, b2, b3, b4, b27, b28, b29, b30, b35, b36, b37, b38, b13, b14, b15, b16, b17, b18, b19, b20, b21, b22, st_1, b23, f_1,b24, b25, b26	c1, c2, c3, c4, c5, c6, c7, c8, c27, c29, c29, c30, c13, c14, c15, c16, c17, c18, c19, c20, c21, c22, st_1, c23, f_1,c24, c25, c26

Figure 12: Behavioural specifications $BS_{EGYandSEC}$ for EGY and SEC Controller and BS_{SEC} for SEC Controller

The corresponding process graph for EGY and SEC Controller is shown in Fig. 13 b). It contains three possible sequential paths from **start** to **final** node. These three paths correspond to three sequences in $BS_{EGYandSEC}$ specification (Fig. 12).

Behaviour specification BS_{SEC} for SEC Controller

$$BS_{SEC} = \{SECObservation, SECObservationBufferFull\}$$

contains two sequences, which comprise exactly the same operations as ones for EGY and SEC Controller (Fig. 12). However, corresponding actions have different names. The process graph for SEC Controller is shown in Fig. 13 c). It contains two sequential paths corresponding two sequences from BS_{SEC}. Sequence *EGYObservation* is not utilized.

As a result of inheritance of interface-role specifications action set A_{PrSp_q} of the inheritor contains two subsets:

$$A_{PrSp_q} = A^{New}_{PrSp_q} \cup A^{Old}_{PrSp_q}; A^{New}_{PrSp_q} \cap A^{Old}_{PrSp_q} = \emptyset, where$$

- $A^{Old}_{PrSp_q}$ is a subset of actions, which are realized by *inherited required relations from* IR^{Inh}_q;
- $A^{New}_{PrSp_q}$ is a subset of actions, which are realized by *newly designed required relations from* IR^{New}_q.

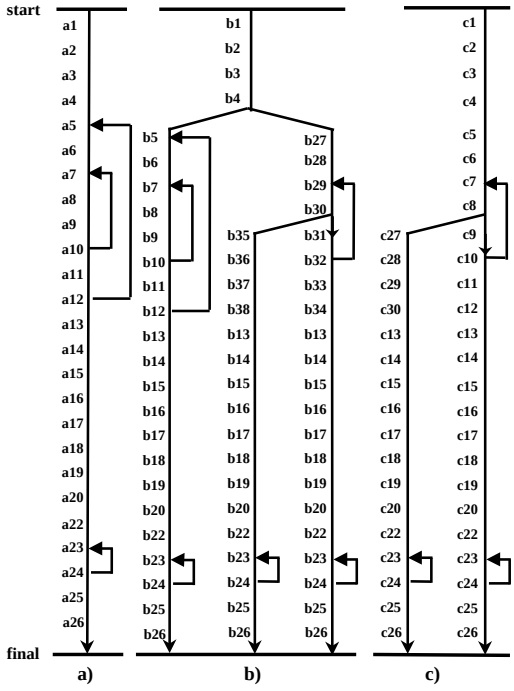


Figure 13: Process graphs for a) EGY Controller; b) EGY and SEC Controller; c) SEC Controller

For example, EGY and SEC Controller has subset $A^{Old}_{EGY\&SEC} = \{b1, b2, ...b26\}$ and subset $A^{New}_{EGY\&SEC}$ of new actions presented in Fig. 14.

$A^{New}_{EGY\&SEC}$={b27,...b38}

b27 - EGY&SECSatelComputer.EGY&SECMeasureControl.StartSECObservationTime
b28 - EGY&SECSatelComputer.EGY&SECMeasureControl.StartSECObservationTime:void
b29 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement
b30 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData(structure)
b31 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData:true
b32 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement:true
b33 - EGY&SECSatelComputer.EGY&SECMeasureControl.FinishSECObservationTime
b34 - EGY&SECSatelComputer.EGY&SECMeasureControl.FinishSECObservationTime:void
b35 - EGY&SECDataAquisition.EGY&SECDataManag.SendSECData:false
b36 - EGY&SECMeasureControl.EGY&SECDataAquisition.StartSECMeasurement:false
b37 - EGY&SECMeasureControl.EGY&SECSatelComputer.BufferFull
b38 - EGY&SECMeasureControl.EGY&SECSatelComputer.BufferFull:void

Figure 14: Subset of new actions for EGY and SEC Controller

Now let us give the definition of correct product behaviour inheritance.

Firstly, we define *renaming function* RN, which we apply on parent set of actions A_{PrSp_p} producing subsets of inherited $A^{Inh}_{PrSp_p}$ and not inherited $A^{not\_Inh}_{PrSp_p}$ parent actions:

$$A^{Inh}_{PrSp_p} \cup A^{not\_Inh}_{PrSp_p} = RN(A_{PrSp_p}); A^{Inh}_{PrSp_p} \cap A^{not\_Inh}_{PrSp_p} = \emptyset$$

such that $A^{Inh}_{PrSp_p} = A^{Old}_{PrSp_q}$.

For example, $RN(A_{EGY}) = A^{Inh}_{EGY} = A^{Old}_{EGY\&SEC} =$

$\{b1, b2, ..., b26\}$; $A^{not\_Inh}_{EGY} = \emptyset$.

SEC Controller does not inherit from EGY and SEC Controller subset of actions $A^{not\_Inh}_{EGY\&SEC}$= $\{b5, b6, b7, b8, b9, b10, b11, b12\}$, which corresponds to the specific EGY measurement subsequence from *EGYObservation* sequence (Fig. 12).

Secondly, let us define on graph of type G_p a pair of *graph transformation rules* $\delta(G_p)$ and $\tau(G_p)$.

- *Blocking rule* $\delta(G_p)$. If subset $B \in A_{PrSp}$ is defined and action $x \in B$, action $a \notin B$ and δ is *blocking* action, then process graph G_p is transformed as it follows from Fig. 15 a). This rule allows cutting down alternative branches starting from actions $x \in B$. Applied to a sequential path this rule cuts it down starting from action x but blocking action is not removed [2].
- *Hiding rule* $\tau(G_p)$. If subset $H \in A_{PrSp}$ is defined and action $y \in H$, action $a \notin H$ and τ is *silent* action, then process graph G_p is transformed as it follows from Fig. 15 b). This rule allows shortening sequential branches by means of deleting actions $y \in H$ [2].

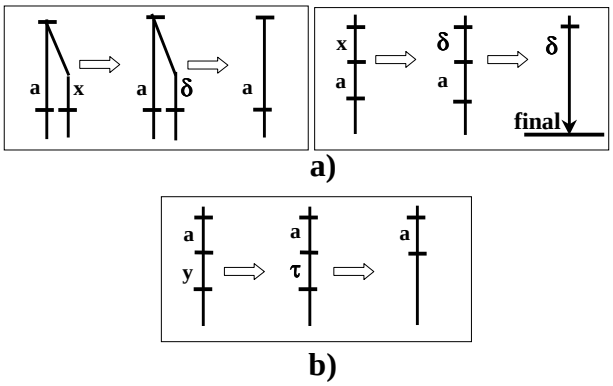


Figure 15: a) $\delta(G_p)$ and b) $\tau(G_p)$ graph transformation rules

Applying process algebra for process of type P [2] on process graph representation we define conditions of *complete* and *partial* inheritance of product specifications.

- Child $PrSp_q$ *completely inherits* parent $PrSp_p$ if and only if $RN(A_{PrSp_p})$= $A^{Inh}_{PrSp_p}$ and $A^{not\_Inh}_{PrSp_p} = \emptyset$ and

  $$\tau(\delta(G_p^{PrSp_q})) = G_p^{PrSp_p}$$

  on condition that
  - the action set of $G_p^{PrSp_p}$ is renamed using function $RN(A_{PrSp_p})$;
  - for the transformation of the child process graph subset B=$A^{New\_Alt}_{PrSp_q}$ and subset H=$A^{New\_Seq}_{PrSp_q}$, where $A^{New\_Alt}_{PrSp_q}$ is a subset of $A^{New}_{PrSp_q}$ containing actions, which *start alternative branches* and $A^{New\_Seq}_{PrSp_q}$ is the rest of $A^{New}_{PrSp_q}$.

In other words, if the parent action set contains only inherited actions we apply the renaming function on the parent set of actions and using the blocking rule eliminate from the child process graph all alternative branches that are started by new actions. Next, we apply the hiding rule and eliminate the rest of new child actions. If the resulting transformed graph is equal to the parent graph with renamed actions, then the child specification is a correct inheritor of the parent specification.

In spite of seemingly tricky notation this definition has clear rationale: alternatives started by new actions will run their own branches to the **final** state (Fig. 11); they will never return to parent behaviour and, therefore, have to be eliminated during parent process graph derivation. New actions running a sequential branch may be hidden to return to parent behaviour within the same branch (sequence).

- Child $PrSp_q$ *partially inherits* parent $PrSp_q$ if and only if $A^{Inh}_{PrSp_p} \neq \emptyset$ and $A^{not\_Inh}_{PrSp_p} \neq \emptyset$ and

  $$\tau(\delta(G_p^{PrSp_q})) = \delta(G_p^{PrSp_p})$$

  on condition that
  - action set from $PrSp_p$ is renamed using function $RN(A_{PrSp_p})$;
  - for the transformation of the child process graph subset B=$A^{New\_Alt}_{PrSp_q}$ and subset H=$A^{New\_Seq}_{PrSp_q}$, where $A^{New\_Alt}_{PrSp_q}$ is a subset of $A^{New}_{PrSp_q}$ containing actions, which *start alternative branches* and $A^{New\_Seq}_{PrSp_q}$ is the rest of $A^{New}_{PrSp_q}$;
  - for the transformation of the parent process graph subset B=$A^{not\_Inh}_{PrSp_p}$.

In other words, child process graph transformation is the same as that in the case of complete inheritance, but before comparing, the parent process graph is transformed using the blocking rule to eliminate not inherited parent actions and, therefore, corresponding sequences. The hiding rule is not applicable to the parent process graph because hiding means shortening sequences from parent specification BS_p each of those must be inherited completely or not inherited at all.

In our case study EGY and SEC Controller is a correct complete inheritor of EGY Controller. Indeed, if we rename parent actions $\{a1, a2, ..., a26\}$ to $\{b1, b2, ..., b26\}$ and hide and block the new actions from the child set, the child process graph is transformed to the parent one (actually, for such transformation blocking of action b27 in Fig. 13 b) is enough).

SEC Controller is a correct partial inheritor of EGY and SEC Controller. To prove this we need to block not inherited action b5 in Fig. 13 b) and rename the parent inherited actions: b1 to c1 , b2 to c2 and so on (compare graphs in Fig. 13 b) and c)). Graph transformation of the child graph is not required because the specification of the inheritor does not contain new actions.

If a child specification is not a correct inheritor of a parent specification, then transformed child or/and parent process graphs contain not eliminated τ and δ actions. The rest of a sequence (or sequences) starting by such an action becomes unreachable [2]. All these sequences are easily transformed back from the process graph and the positions of τ or/and δ actions show the points of design errors. These errors are actions, which cannot be realized within a given specification. So, the roles performing such impossible actions can be indicated. As a result, the method allows a designer not only to prove correctness of inherited specifications but also to find design bags.

5. TOOL SUPPORT

The described method comprises several formal techniques and algorithms to be used during a modelling process. The successful usage of the method requires appropriate tool support. We have developed a tool that provides an environment for design and reuse of component specifications in the UML [24]. The tool is implemented as a Rational Rose Add-In [20].

A familiar with Rational Rose designer performs with the help of the tool the following sequential steps:

1. He/she chooses a parent product to inherit from. The interface-role diagram of this product is drawn by the tool in a Rational Rose class diagram window.
2. The designer extends the parent interface-role diagram by new roles and interfaces using dialogs provided by the tool. The interface-role diagram of the new product is produced.
3. The designer draws a set of sequence diagrams using the set of actions derived by the tool from the interface-role diagram of the new product.
4. The tool constructs the process graph corresponding to the UML specification of the new product.
5. The tool defines action sets that have to be hidden and blocked in the process graph of the new product to derive the parent process graph, hides and blocks those actions and compares the parent process graph with the process graph-result of hiding and blocking.
6. If the process graph-result is not equal to the parent process graph, then the sequence diagrams that represent unreachable behaviour patterns are indicated by the tool. The designer should correct the design of the new product.
7. If the process graph-result is equal to the parent process graph, then the new product specification is correct and it can be used in further product development phases.

The screen shot of a derivation dialog for EGY and SEC Controller is shown in Fig 16. More details about the tool are contained in [24].

6. CONCLUSION AND FUTURE WORK

The presented method provides evolutionary incremental modelling of software product line members using inheritance of their behaviour specifications. Correctness of model transformations is proved by using a derivation technique that allows a designer to produce the process graph of a product-predecessor from the inheritor's one or to find the points of incorrect design.

An appropriate tool prototype has been developed to support the modelling. The tool applies techniques and algorithms which accompany the method. Robustness of the method and the tool is proved by the modelling of an industrial case study.

In future work we intend to find out how our method applicable to large-scale industrial systems. In this context the problem of product requirements mapping to our specifications needs to be investigated. In large-scale applications such successful direct mapping that we have shown in our case study is not so apparent. A kind of a specifications mapping technique is required. Recent researches (e.g., see in [14]) apply UML use case and scenario diagrams to SPL requirements engineering. In such a case, requirements can be mapped to interface-role specifications directly: actors iterating via use cases can be mapped to roles; use cases itself can be realized as sets of required relations between roles; scenario diagrams can be considered as prototypes of sequence diagrams.

Mapping between our specifications and product component architectures is also a significant problem. Component systems are usually described in Architecture Description Languages (ADLs) (see good overview [15]). Most of them allow representing roles and interfaces as components and connectors. Among others, ADLs with strong component evolution support, such as Koala [18], are more close to our approach. Moreover, Koala is a good practical example of an ADL for component-based product population development. Our specifications can be mapped to Koala's configurations in such a manner that roles would correspond to components. Provided and required relations can be presented by Koala's provides and requires interfaces. Compositional capacity of a Koala component (combinations of components are components again [18]) provides appropriate support for inheritance of roles. Inheritance of interface-role specifications is supported by the ability of Koala's configurations to comprise other configurations.

7. REFERENCES

[1] Baeten J.C.M.,W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[2] Basten T., W.M.P. van der Aalst. Inheritance of behaviour. *The Journal of Logic and Algebraic Programming*, 46:47–145, 2001.

[3] Becker M. Towards a General Model of Variability in Product Families. *Workshop on Software Variability Management. Editors Jilles van Gurp and Jan Bosch. Groningen, The Netherlands. http://www.cs.rug.nl/Research/SE /svm/proceedingsSVM2003Groningen.pdf*, pages 19–27, 2003.

[4] Bosch J. *Design&Reuse of Software Architectures - Adopting and Evolving a Product Line Approach.* Addison-Wesley, 2000.

[5] Bosch J. Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In *Second Conference Software Product Line Conference, SPLC2*, August 2002.

[6] Bosch J., M. Svahnberg and J. van Gurp. On the notion of variability in software product lines. In *Software Architecture. Working IEEE/IFIP Conference*, pages 45–54, 2001.

[7] Cheesman J., J. Daniels. *UML Components. A simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

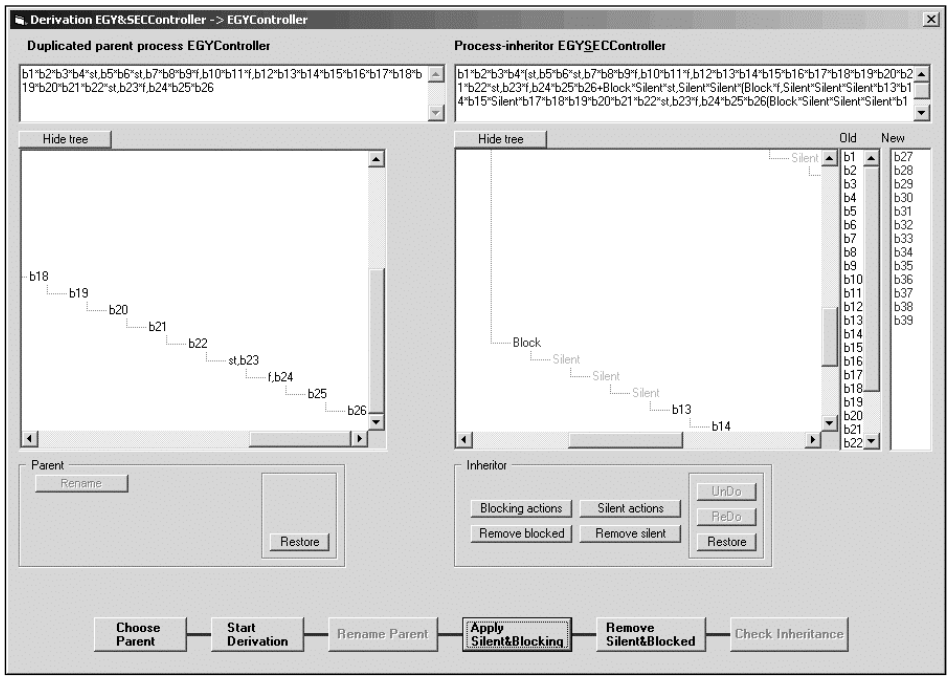


Figure 16: Parent process derivation dialog in the tool

[8] P. Clements and R. Northrop. *Software Product Lines - Practices and Patterns*. Pearson Education (Addison-Wesley), ISBN 0-201-30977-7, 2000.

[9] Dobrica L.,E.Niemela. *A strategy for analysis product line software architectures*. VTT Technical Research Centre of Finland, ISBN 951-38-5599-6, 2000.

[10] P. Donohoe, editor. *Software Product Lines - Experience and Research Directions*. Kluwer Academic Publishers, 2000.

[11] D'Souza D.F., A.C.Wills. *Objects, Components and Frameworks with UML. The CATALYSIS Approach*. Addison-Wesley , 1999.

[12] Gurp J. van , J.Bosch. Design Erosion: Problems and Causes. *Journal of Systems and Software, 61(2), Elsevier*, 61:105–119, 2002.

[13] Ihme T. A ROOM Framework for the Spectrometer Controller Product Line. *Workshop on Object Technology for Product Line Architecture*, pages 119–128, ESI–199–TR–034, 1999.

[14] MacGregor J. Requirements Engineering in Industrial Product Lines. In *International Workshop on Requirements Engineering for Product Lines, REPL'02*, pages 5–11, Essen, Germany, 2002.

[15] N. Medvidovic and R.Taylor. A classification and comparison framework for software architecture description languages. Technical report, USC Center for Software Engineering http://sunset.usc.edu/ neno/papers/TSE-ADL.pdf.

[16] OMG. *Unified Modeling Language Specification v.1.3, ad/99-06-10 http://www.rational.com/ uml/resources/documentation/index.jsp*, June 1999.

[17] R. van Ommering. Roadmapping a Product Population Architecture. *Workshop on Product Family Engineering, Bilbao, Spane*, 2001.

[18] R. van Ommering, F. van der Linden, J. Kramer, J.Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, pages p78–85, March 2000.

[19] R. van Ommering, J.Bosch. Widening the Scope of Software Product Lines - From Variation to Composition. In *Second Conference Software Product Line Conference, SPLC2*, pages 328–347, August 2002.

[20] Rational Rose 98i. *Rose Extensibility Reference 2000*. http:// www.rational. comwww.se. fh-heilbronn.de/ usefulstuff/ Rational Rose 98i Documentation.

[21] Roubtsov S.A., E.E.Roubtsova. Modeling Evolution and Variability of Software Product Lines Using Interface Suites. *Workshop on Software Variability Management. Editors Jilles van Gurp and Jan Bosch. Groningen, The Netherlands. http://www.cs.rug.nl/Research/SE /svm/proceedingsSVM2003Groningen.pdf*, pages 62–71, 2003.

[22] Roubtsova E. and R. Kuiper. Process Semantics for UML Component Specifications to Assess Inheritance. *Electronic Notes in Theoretical Computer Science, 72,3 Elsevier Science Publishers, Paolo Bottoni and Mark Minas, http://www.elsevier.nl/gej- ng/31/29/23/127/48/show/Products/notes/index.htt*, 2003.

[23] Roubtsova E.E , L.C.M. van Gool, R. Kuiper, H.B.M. Jonkers. A Specification Model For Interface Suites. *UML'01, LNCS 2185*, pages 457–471, 2001.

[24] Roubtsova E.E., S.A.Roubtsov. UML-based Tool for Constructing Component Systems via Component Behaviour Inheritance. *Proceedings of the Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03) To appear in Elsevier Electronic Notes in Theoretical Computer Science, 80 (2003) http://www.elsevier.nl/locate/entcs/volume80.html* , pages 139–154, 2003.

[25] Svahnberg M., Bosch J. Evolution in Software Product Lines: Two Cases. *Journal of Software Maintenance: Research and Practice, Vol. 11, No. 6*, 1999.

[26] Szyperski C. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, New-York, 1998.

[27] W.Eixelsberger, M.Ogris, H.Gall, and B.Bellay. Software recovery of a program family. *International conference on Software Engineering, Kyoto, Japan*, 1998.

[28] Zhao L., Kendall E. Role Modelling for Component Design. *The 33rd Hawaii International Conference on System Science*, 2000.

Evaluating Clone Detection Techniques

Filip Van Rysselberghe
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1, B 2020 Antwerpen
Filip.VanRysselberghe@ua.ac.be

Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1, B 2020 Antwerpen
Serge.Demeyer@ua.ac.be

Abstract

In the last decade, several researchers have investigated techniques to detect duplicated code in programs exceeding hundreds of thousands lines of code. All of these techniques have known merits and deficiencies, but as of today, little is known on where to fit these techniques into the software maintenance process. This paper compares three representative detection techniques (simple line matching, parameterized matching, and metric fingerprints) by means of five small to medium cases and analyses the differences between the reported matches. Based on this experiment, we conclude that (1) simple line matching is best suited for a first crude overview of the duplicated code; (2) metric fingerprints work best in combination with a refactoring tool that is able to remove duplicated subroutines; (3) parameterized matching works best in combination with more fine-grained refactoring tools that work on the statement level.

1. Introduction

Code cloning or the act of copying code fragments and making minor, non–functional alterations, is a well-known problem for evolving software systems leading to duplicated code fragments or code clones. Of course, the normal functioning of the system is not affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive [7, 18]. Fortunately, the problem has been studied intensively and several techniques to both detect and remove duplicated code have been proposed in the literature.

As far as removal of duplicated code is concerned, the state of the art proposes *refactoring* which is a technique to gradually improve the structure of (object-oriented) programs while preserving their external behaviour [17]. *Extract Method* which extracts portions of duplicated code in a separate method, is an example of a typical refactoring to remove duplicated code. However, quite often one must use a series of refactorings to actually remove duplicated code, as in *Transform Conditionals into Polymorphism* where duplicated conditional logic is refactored over the class hierarchy using polymorphism [7]. With refactoring tools like the refactoring browser [6] emerging from research laboratories into mainstream programming environments[1], refactoring is becoming a mature and widespread technique.

Concerning the detection of duplicated code, numerous techniques have been successfully applied on industrial systems. These techniques can be roughly classified into three categories. (i) *string-based*, i.e. the program is divided into a number of strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings [8, 12]; (ii) *token-based*, i.e. a lexer tool divides the program into a stream of

[1] See http://www.refactoring.com/ for an overview of IDE's supporting refactoring

tokens and then searches for series of similar tokens [2, 13]; (iii) *parse–tree based*, i.e., after building a complete parse-tree one performs pattern matching on the tree to search for similar sub–trees [14, 15, 4]. On the first International Workshop on Detection of Software Clones, a number of research groups recently participated in a clone detection contest[2] to compare the accuracy of different tools against a benchmark of programs containing known duplication. The results of this experiment are currently being analysed by the participants.

Despite all this progress, little is known about the most optimal application of a given clone detection technique during the maintenance process. For instance, which technique should one use in a problem assessment phase, when one suspects duplicated code but isn't sure how much and in which files? Or which technique works best in combination with a refactoring tool, which has to know the exact boundaries of the code segment to be refactored, including possible renaming of variables and parameters? To answer these questions, this paper compares three representative *clone detection techniques* —namely simple line matching, parameterized matching, and metric fingerprints— by means of five small to medium cases. The reported matches as well as the process are analysed with special interest in differences. Afterwards, our findings are interpreted in the context of a generic software maintenance process and some suggestions are made on the most optimal application of a given technique.

The paper is structured as a comparative study, however due to the multiple aspects involved in the issue studied a more extensive experiment is necessary in the near future. A brief overview of existing duplicated code detection techniques is given in section 2. The experimental set-up, including the questions and cases driving the experiment are discussed in section 3. The results of section 4 are interpreted in section 5 to evaluate where the given technique might fit into the software maintenance process. Finally, section 6 summarises our findings in a conclusion.

2. Detection Techniques

The detection of code clones is a two phase process which consists of a *transformation* and a *comparison* phase. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. During the succeeding comparison phase the actual matches are detected.

Due to its central role, it is reasonable to classify detection techniques according to their internal format. This section gives an overview of the different techniques available for each category while selecting a representative for each category.

2.1. String Based

String based techniques use basic string transformation and comparison algorithms which makes them independent of programming languages.

Techniques in this category differ in underlying string comparison algorithm. Comparing calculated signatures per line, is one possibility to identify for matching substrings [12]. Line matching, which comes in two variants, is an alternative which is selected as representative for this category because it uses general string manipulations.

Simple Line Matching is the first variant of line matching in which both detection phases are straightforward.

Only minor transformations using string manipulation operations, which can operate using no or very limited knowledge about possible language constructs, are applied. Typical transformations are the removal of empty lines and white spaces.

During comparison all lines are compared with each other using a string matching algorithm. This results in a large search space which is usually reduced using hashing buckets. Before comparing all the lines, they are hashed into one of *n* possible buckets. Afterwards all pairs in the same bucket are compared.

[2] http://www.informatik.uni-stuttgart.de/ifi/ps/clones/

Duploc is a Smalltalk tool which implements such a simple line matching technique [8], however also a Java version is available

Parameterized Line Matching is another variant of line matching which detects both identical as well as similar code fragments. The idea is that since identifier–names and literals are likely to change when cloning a code fragment, they can be considered as changeable *parameters*. Therefore, similar fragments which differ only in the naming of these parameters, are allowed.

To enable such parameterization, the set of transformations is extended with an additional transformation that replaces all identifiers and literals with one, common identifier symbol like ”$”. Due to this additional substitution, the comparison becomes independent of the parameters. Therefore no additional changes are necessary to the comparison algorithm itself.

Parameterized line matching is discussed in [9].

2.2. Token Based

Token based techniques use a more sophisticated transformation algorithm by constructing a token stream from the source code, hence require a lexer. The presence of such tokens makes it possible to use improved comparison algorithms.

Next to parameterized matching with suffix trees, which acts as representative, we include [13] in this category because it also transforms the source code in a token-structure which is afterwards matched. The latter tries to remove much more detail by summarising non interesting code fragments.

Parameterized Matching With Suffix Trees consists of three consecutive steps manipulating a suffix tree as internal representation.

In the first step, a lexical analyser passes over the source text transforming identifiers and literals in parameter symbols, while the typographical structure of each line is encoded in a non-parameter symbol. One symbol always refers to the same identifier, literal or structure. The result of this first step is a parameterized string or p-string.

Once the p-string is constructed, a criterion to decide whether two sequences in this p-string are a parameterized match or not is necessary. Two strings are a parameterized match if one can be transformed into the other by applying a one-to-one mapping renaming the parameter symbols. An additional encoding prev(S) of the parameter symbols helps us verifying this criterion. In this encoding, each first occurrence of a parameter symbol is replaced by a 0. All later occurrences are replaced by the distance since the previous occurrence of the same symbol. Thus, when two sequences have the same encoding, they are the same except for a systematic renaming of the parameter symbols.

After the lexical analysis, a data structure called a parameterized suffix tree (p-suffix tree) is built for the p-string. A p-suffix tree is a generalisation of the suffix tree data structure [16] which contains the prev()-encoding of every suffix of a P-string. Concatenating the labels of the arcs on the path from the root to the leaf yields the prev( )-encoding of one suffix. The use of a suffix tree allows a more efficient detection of maximal, parameterized matches.

All that is left for the last step, is to find maximal paths in the p-suffix tree that are longer than a predefined character length.

Parameterized matching using suffix trees was introduced in [2] with Dup as implementation example.

2.3. Parse-tree Based

Parse tree based techniques use a heavyweight transformation algorithm, i.e. the construction of a parse tree. Because of the richness of this structure, it is possible to try various comparison algorithms as well.

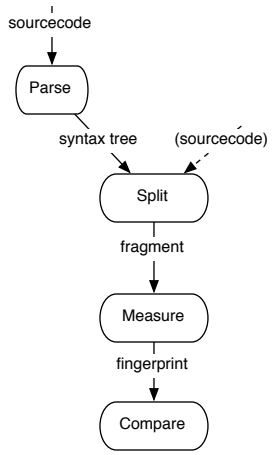


Figure 1. Detection steps for the metric fingerprint technique

The representing technique differs from [4] in that the latter uses sub–tree matching on the syntax tree.

Metric Fingerprints builds on the idea that you can characterise a code fragment using a set of numbers. These numbers are measurements which identify the functional structure of the fragment and sometimes the layout.

The metric fingerprint technique can be divided in five steps, each with a well-defined task. However the algorithm behind each task may differ between implementations. Figure 1 shows the basic steps in the detection process.

Before we can characterise the functional structure of a code fragment with numbers, it's wise to transform the source code into a representation that allows us to calculate such measurements efficiently. This transformation job is done using a *parser* which builds the syntax tree of the source code.

After parsing we end up with one large syntax tree. This tree is then *split* into interesting fragments. The choice of the type of fragments used is difficult because it affects the detection results. Most of the time, however, method and scope blocks are used as fragments since they are easily extracted from a syntax tree.

Afterwards the fragments are characterised through a set of measurements by *measuring* the values for a set of metrics, chosen in advance. This set of metrics can differ between various implementations, but most of the time it specifies functional properties. However there are implementations in which layout metrics are used as well. Cyclomatic complexity, function points, expression complexity (functional) and lines of code (layout) are examples of possible measures.

Finally, these sets of numbers are compared to each other. Depending on the implementation, algorithms with different levels of sophistication or power may be used. One possible approach calculates the Euclidean distance between each pair of fingerprints, considering fragments within zero distance as clones.

Both [14] and [15] describe a possible implementation of metric fingerprints. In the first, the metric set consists of 5 indirect metrics which are treated as a vector, while the latter uses 21 measures which are compared to each other using a system of hierarchical categories (an overview of both techniques can be found in [19]).

3. Research Approach

The research process used during our experiment is based on the Goal-Question-Metric paradigm which states that you should (1) outline a goal, (2) generate questions that verify whether the goal has been met and (3) select measures to answer them [3].

3.1. Goal

Identify which clone detection techniques are more appropriate for specific tasks of the maintenance process.

3.2. Questions

The questions we selected, were chosen because each highlights features that are of importance during the maintenance process. This way, these questions help us verifying whether our goal was accomplished.

Q1. How much configuration is needed to apply on another language? Before using a technique, you like to know how much configuration has to be done to adapt it to your particular programming context. Especially because it may limit the applicability of the technique, certainly in COBOL and C++ environments, where lots of dialects exist.

Q2. What kind of matches are found? Depending on the maintenance task at hand, you may be looking for specific kinds of duplication. For instance, during a problem assessment phase, maintainers want to obtain an overall report of the amount of duplication existing in all program files. On the other hand, during a restructuring phase, maintainers are interested in a duplication tool that detects only the programming constructs that one can restructure using a particular tool. Therefore a refactoring tool, moving methods in the class hierarchy, is interested only in duplicated method bodies.

Q3. How accurate are the results? For the clone detection problem, detection accuracy is difficult to define, but in the context of duplicated code detection it is characterised by three quality measures:

- number of false positives (to be minimised): that is, the number of matches the technique incorrectly identified as a piece of duplicated code.
- number of useless matches (to be minimised): that is, the number of matches which are not worth to be removed by means of refactoring. Typically depending on the length of a match.
- number of recognisable matches (to be maximised): that is, the number of matches that are easily recognised as interesting. For instance, in a program restructuring phase these are the matches that are easily removed by the refactoring tool at hand.

Q4. How does it perform? When using a detection technique one wishes to balance the amount of usable information that one can derive, with the time and memory one invested. Therefore you need to establish the performance of each technique and identify performance bottlenecks. This question addresses how the execution time of each technique relates to its input.

3.3. Experimental set-up

The next step after selecting research questions, consists of constructing an experiment that answers these questions. For the experiment reported in this paper following steps were conducted:

creation of reference implementations — Evaluating clone detection techniques differs from the evaluation of clone detection tools, in that it is the algorithm that is evaluated instead of the implementation. Differences in execution time between tools can for example be caused by the use of different programming languages or the application of techniques such as parallel computing. Unlike [5], which evaluates the results of various detection tools, this experiment focusses on the techniques themselves.

To evaluate each of these techniques, reference implementations of them were made in Java. Each of these implementations tried to adhere as closely as possible to the original technique's specification as given in [2]

for parameterized matching using suffix trees and [14, 15] for the metric fingerprint technique. For simple line matching such a reference implementation was already available and the original Duploc-tool[8] was used as an additional reference.

selection of cases — Five case were selected to evaluate the different techniques. These cases are representative for different degrees of duplication. Their limited size (under 10 000 LOC) allows an in-depth study of the duplication present as well as the reported matches. Section 3.4 describes each of the different cases.

application of the implementations — After selecting the cases, the different techniques were applied on each of them.

comparison and collection of results — At the end, the different matches were studied and compared with the different techniques. Data that was related to the execution of the different implementations like the execution time and its memory use, was studied as well.

3.4. Selected Cases

For the experiment, we selected five small to medium sized cases which are known to suffer from different kinds of duplication, although we did not know the exact locations of the duplicated code beforehand. Therefore, these cases are representative for various usage scenario's or different amounts of clones. Moreover, all cases are available on the web which allows replication of the experiment by other researchers studying duplicated code detection techniques. Following cases were used:

- ScoreMaster is a Java application automatically generated for the Enhydra web–server. Because most of the code has been generated automatically, it contains a high degree of duplication.
- TextEdit is an example project that is distributed with Borland's JBuilder to demonstrate GUI programming in Java. Due to its educational nature it contains little duplication[20].
- Brahms is music sequencing and notation software for linux written in C++ and was formerly known as KooBase. The small amount of duplication present is of a different nature because the code was written manually in an open source context[1].
- JMocha is a Java beans benchmark developed by IBM[11].
- JavaParser of JMetric is, as indicated by its name, a Java parser generated by Java for the JMetric project. It concerns a larger example of automatically generated code full of duplication[10].

4. Results

This section reports about the experiment by answering the questions listed under 3.2. A summary of these answers is given by table 1.

How much configuration is needed to apply on another language?

Simple line matching, as it only utilises basic string manipulations, is a truly language independent technique which is *very easy to configure*. As a language independent technique, no modification is required to be applicable on different languages.

All the remaining techniques on the other hand, do require configuration.

For parameterized matching the portability to another language is fair. Changing the lexer, which lies at the basis of both techniques, suffices to port it. Because more changes in the lexer are necessary for the parameterized

line matching technique, its portability is slightly lower than that of the suffix tree technique. Both parameterized techniques are *fairly portable*.

The metric fingerprint technique demands *much configuration effort as it is syntax dependent* due to the use of a parser. Even in our very first attempt to analyse a program, we were confronted with this syntax dependence because it failed due to a syntax error in the analysed code. The use of a parser limits the technique to syntactically correct sources of one language and makes changing to other languages difficult.

What kind of matches are found?

A rough classification of the clones found yields: *functional block duplication* and *general duplication*.

Functional block duplication characterises the duplication found by the metric fingerprint technique. Because this technique characterises functional blocks such as methods or code blocks by a fingerprint, only code fragments which share a functionally equivalent structure, are reported. The addition or removal of structures in a block violates this equivalence.

General duplication is found by the three other techniques. Everything that was duplicated, including pre–processor directives or comments, can be detected by them.

This last category can eventually be subdivided into the different fragments found by the corresponding techniques: *duplicated symbol blocks* for the suffix tree technique, *duplicated lines block* for parameterized line matching and *equal lines* for simple line matching. Duplicated in this context refers to the fact that parameter symbols may have changed.

How accurate are the results?

Number of false matches— *No false matches* are reported by both simple line matching and parameterized matching using suffix trees. Simple line matching reports only equal lines which makes it impossible to have false positives, while parameterized matching using suffix trees benefits from its P-string encoding that enforces a strict one–to–one parameterization. Only positive matches (parameterized or exact) are found by them

Parameterized line matching allows a non systematic renaming of the parameters which leads to *few false matches*. Such systematic renaming is necessary to ensure that two fragments share the same basis functionality which characterises duplication. Figure 2 shows an example, discovered in TextEdit. The problem especially seems to target GUI initialisation code. However reporting fragments consisting of a long sequence of matching lines instead of shorter ones, helps in keeping the number low. When we used this technique for ScoreMaster and Brahms we did not receive any false matches, while one false match was reported for TextEdit.

Even *more false matches* are reported by the metric fingerprint technique. Applying metric fingerprints with block-fragments resulted in over 200 false matches (cf. with 0 for the other 3 techniques) while only two were found using methods as fragments. The characterisation of expressions which lacks accuracy (see figure 3 for an example in ScoreMaster), is responsible for this problem. However it is our opinion that adding better expression metrics, like “expression complexity”, reduces this problem’s impact. Furthermore, less false matches are found when the granularity or size of the selected fragments is bigger. The number of false matches for this technique thus depends on the way expressions are characterised and the length of the fragments.

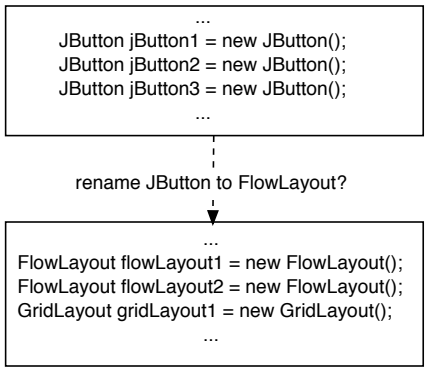


Figure 2. Example of a false match for parameterized line matching

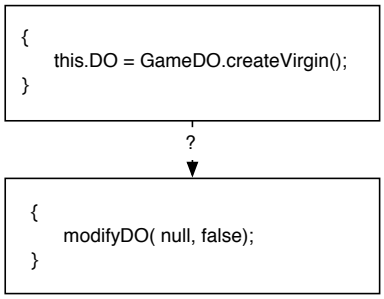


Figure 3. Example of a false match for metric fingerprints

Number of useless matches— The use of a threshold like in both parameterized matching techniques, keeps the number of useless matches *low*. Changing the threshold helped us in keeping the number of useless matches below 20.

For the metric fingerprint technique *more useless matches* are reported. Most of them are only one to four lines long and are caused because two method calls with the same number of arguments always match. For TextEdit for example, we found 133 useless matches on 138 reported matches (137 of them were valid matches) when we used method granularity. Using a threshold would reduce the amount of useless matches, especially in programs which contain many small methods or code blocks.

Simple line matching also reports *many useless matches*. For the same example as in the previous paragraph we got 229 useless matches. The problem here is that any program already contains some exactly matching lines by nature. As an example think of the “return;” statement you tend to write in your program. It is hard to estimate the exact number of useless matches in general but usually it is larger than the amount for metric fingerprints.

Number of recognisable matches— For the metric fingerprints technique the number is *high*. Each match that is returned is a *functional block* like e.g. scope blocks and method definitions.

Both parameterized matching techniques return a *lower* number of recognisable matches. It is difficult to decide which matches are important by just looking at the output because each match represents a chunk of duplicated lines or symbols, which lacks context.

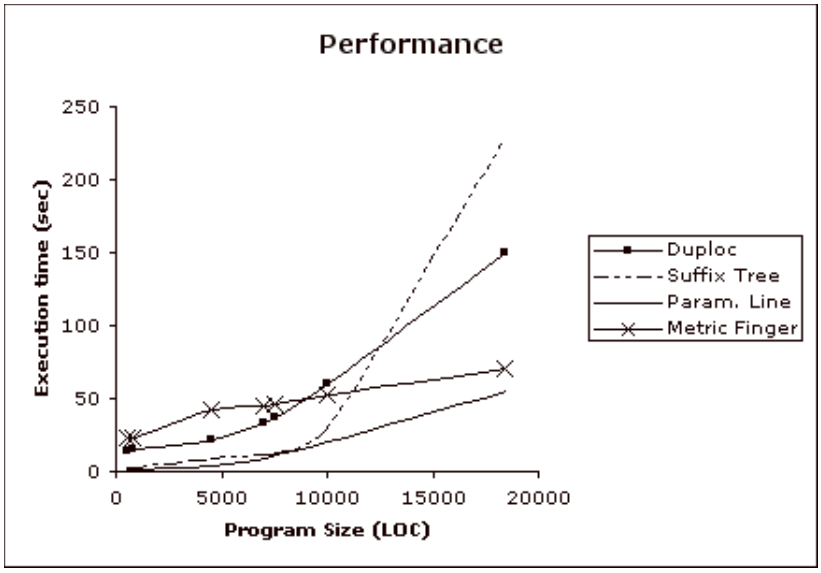


Figure 4. Performance of the different techniques

The number of recognisable matches for simple line matching is *even lower* (reduced from 4 with parameterized matching to 2). All exactly matching lines are reported. Visualisation can be used to detect the interesting duplicates. However the lack of parameterization makes it more difficult than the parameterized techniques to detect altered duplicates.

How does it perform?

Because the actual performance of a technique depends on many factors like implementation and testing platform, we started by calculating the theoretical time complexities. For both line matching techniques this results in a time complexity of O(n^2) because each line is compared with each other line resulting in an exponential complexity. Using Ω hash buckets as proposed in [8] reduces this complexity to O($\frac{n^2}{\Omega}$). Parameterized matching using suffix trees on the other hand, has a complexity of O($|\Pi| * n$) (with $|\Pi|$ the number of parameter symbols) as was formally proven by Baker in [2]. As a last technique we studied the time complexity of the metric fingerprint technique which shows a time complexity of O(m^2) when a simple comparison is used to compare the *m* fragments.

Afterwards we compared these complexity formulas with the execution times we measured[3] leading to a couple of rather interesting observations. A first observation was the problem of page swapping. From a certain point (in our experiment 10 000 LOC) the linearity of the suffix tree could no longer be maintained. The reason for this was the *page swapping* which was necessary to store the whole suffix tree in memory. Memory space is thus a constraining factor when analysing large projects.

A second observation was the unexpectedly high performance of the parameterized line matching technique. The execution time of this technique showed a very flat exponential tendency. Better memory use and shorter comparisons due to shorter strings, are reasons for that performance.

Figure 4 shows how the execution time for each technique relates to the input size. It clearly shows our two observations as well as an overview of each technique's performance.

[3]Testing platform was a pentium 200Mhz with 64MByte RAM

	Portability	Duplication	Matches: number of			Scalability
			False	Useless	Recognisable	
Simple Line	$+++++$	general lines		$----$	$+$	$+$
Param. Line	$+++$	general line block	$-$		$++++$	$+++$
Suffix Tree	$++$	general token block			$++++$	$++$
Metric. Fing.		functional entity	$--(--)$	$--(--)$	$+++++$	$+++$

Table 1. Summary of the relation between each technique and the properties studied. The number of symbols indicates the comparative degree of satisfaction of the property studied. Positive properties are marked with +, negative with -. The additional symbols placed between brackets, denote the additional impact when using block–granularity

5. Interpretation

A first observation we made, was the difference in scalability of the various techniques. By applying each technique on a common case, we were able to get in touch with the scalability of the different techniques, something we could not derive from the theoretical time complexities alone. Who could ever imagine that the relative execution time of parameterized line matching increases much slower than its simple counterpart while an additional transformation is applied?

During our experiment we were certainly puzzled by the major difference in execution time (2 minutes versus 8) for the suffix tree technique when advancing from 7500 LOC to 10710 LOC, certainly because a linear time complexity was formally proven for this technique. As analysis of the memory showed, page swapping was the reason for this behaviour. By experimenting we found that parameterized matching using suffix trees has problems sustaining its theoretical linearity due to memory restrictions which in turn limits its scalability.

For the comparison of the output of the techniques, we also used a visualisation tool. Quite often this visual comparison showed striking differences in the outputs. At one moment for example, we were really stunned by the large amount of matches reported when we used block–fragments instead of method–fragments in the metric fingerprint technique. However our amazement was of short notice because investigation of the various fragments revealed a large number of false and useless matches. Comparison with other techniques supported this idea immediately. Using block-granularity for metric fingerprints did not only cost much more time and memory, but also resulted in a large amount of useless information.

After this we immediately compared the method-granularity with the remaining techniques. The number of matches drew our immediate attention as metric fingerprints finds a number of very small (1 or 2 lines), yet useless matches. However, the remaining large matches were duplicated methods, which usually are easy to refactor. The limited amount of matches combined with their clear content makes the technique useful in a first, coarse refactoring phase.

At first sight, the parameterized techniques and simple line matching seemed to report different duplicates, while the difference in output between our two parameterized techniques was small. However, a second more in-depth look at the reports revealed that sometimes very small matches were found by simple line matching while the parameterized techniques found an entire fragment. A small amount of duplicates was not even found by simple line matching because in each line at least one parameter symbol was altered. This indicates that some very detailed duplication was missed. Applying parameterized matching resulted in more detailed and more recognisable matches.

6. Conclusion

In this paper we have studied three duplicated code detection techniques, which are representative for the techniques published in the literature. By means of five small to medium cases (some of them including generated code, hence having lots of duplication) we compared the results, focussing on those portions where the techniques performed differently. Based on this experiment, we make the following conclusions.

- *Simple line matching* (representative for the string-based techniques) gives a crude overview of the duplicated code that is quite easy to obtain, hence is most appropriate during problem detection and problem assessment.
- *Parameterized matching* (representative for the token-based approaches) provides a precise picture of a given piece of duplicated code and is robust against rename operations. Therefore it works best in combination with fine-grained refactoring tools that work on the level of statements (i.e. *Extract Method*, *Move Behaviour Close to Data*, and *Transform Conditionals into Polymorphism*.
- *Metric fingerprints* (representative for the parse-tree based techniques) are very good at revealing duplicated subroutines, irrespective of small differences, hence work best in combination with refactoring tools that work on the method level (i.e. *Remove Method* and *Pull up method*);

These results are preliminary in nature and should be confirmed by other experiments. First of all, future experiments should incorporate large and very-large (over a million lines of code) programs into the set of cases to see whether our results still hold. Secondly, the same experiment should be done with other techniques to see whether our findings indeed generalise across the given categories.

Despite these limitations, we have shown that the different clone detection techniques reported in the literature each have specific advantages compared to the others. As such, each technique is more appropriate for a certain maintenance task. In that sense, this paper laid the foundation for a more systematic way of detecting and removing duplicated code.

7. Acknowledgements

We would like to thank Gerd Van Den Heuvel, whose master's thesis provided the necessary means for conducting the experiments described in this paper. We also would like to thank Stéphane Ducasse, Bart Du Bois and Andy Zaidman for reviewing the paper. Matthias Rieger was helpful by providing us with an implementation of Duploc.

References

[1] Brahms. *http://brahms.sourceforge.net*. by Sourceforge.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering 1995*, 1995.

[3] V. R. Basili and H. D. Rombach. The tame project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758 – 773, 1988.

[4] I. Baxter, A. Yahin, L. Moura, and M. S. Anna. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, 1998.

[5] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation(SCAM '02)*, October 2002.

[6] J. B. D. Roberts and R. E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253 – 263, 1997.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.
[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, 1999.
[9] G. V. D. Heuvel. Parameterized matching: a technique for the detection of duplicated code. Master's thesis, University of Antwerp, 2002.
[10] Jmetric. *http://www.it.swin.edu.au/projects/jmetric/products/jmetric*. by School of Information Technologie at Swinburne University of Technology.
[11] Jmocha. *http://www–124.ibm.com/developerworks/opensource/jmocha/*. by IBM.
[12] J. Johnson. Identifying redundancy in source code using fingerprints. In *Cascon*, 1993.
[13] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654 – 670, 2002.
[14] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1), 1996.
[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, 1996.
[16] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.
[17] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[18] D. Parnas. Software aging. In *Proceedings of The 16th International Conference on Software Engineering*, 1994.
[19] F. V. Rysselberghe. Detecting duplicated code using metric fingerprints. Master's thesis, University of Antwerp, 2002.
[20] Textedit. *http://www.borland.com/jbuilder*. by Borland.

Describing the impact of refactoring on internal program quality

Bart Du Bois
Lab On ReEngineering
Universiteit Antwerpen
Middelheimlaan 1, B-2020 Antwerpen, Belgium
bart.dubois@ua.ac.be

Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be

Abstract

The technique of refactoring – restructuring the source-code of an object-oriented program without changing its external behavior – has been embraced by many object-oriented software developers as a way to accommodate changing requirements. The overall goal of refactoring is to improve the maintainability of software. Unfortunately, it is unclear how specific quality factors are affected. Therefore, this paper proposes a formalism to describe the impact of a representative number of refactorings on an AST representation of the source code, extended with cross-references. We elicitate how internal program quality metrics can be formally defined on top of this program structure representation, and demonstrate how to project the impact of refactorings on these internal program quality metric values in the form of potential drifts or improvements.

1 Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behavior [9, 16, 17]. The key idea is to redistribute attributes and methods across the class hierarchy in order to prepare the software for future extensions. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [9].

However, the impact of a particular refactoring on the software quality varies. Some refactorings raise the level of abstraction (at the expense of increasing the program complexity), others may reduce the complexity (at the expense of decreasing the performance, for example), etc. Therefore, our goal is to provide techniques and tools for software developers to help them maintain and improve program quality through refactorings. The use of metrics in achieving this goal is advocated in [6].

This paper takes a first step towards this goal, by proposing a formalism for describing the impact of refactorings on program structure. Our representation of the program structure is borrowed from [13], which uses an abstract syntax tree representation of the source-code, extended with cross-references to model type references, method calls, accesses, updates and inheritance links. This abstract syntax tree representation allows us to reason about program structure in terms of nodes interconnected with edges. The fact that dependencies between program entities are explicit in this representation makes it easier to reason about the impact of refactorings from a quality perspective.

Object-oriented program quality metrics are typically used as internal quality factors [3]. Defining these metrics in terms of the entities of the extended tree representation allows formal descriptions of structural changes on the tree (eg. refactorings) to be projected into impacts on the particular metrics. In this way, the integration of the formal description of refactorings and the formal definition of a representative set of object-oriented program quality metrics provides *a-priori* feedback on the impact of any application of a particular refactoring on any particular internal program quality metric.

The goal of this mechanism is to construct (once and only once) refactoring impact tables. Such information facilitates refactoring trade-offs, in that they make explicit which internal program quality metrics are affected when the refactoring

*Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)

would be applied. In other words, the contribution of this work is to make explicit the *quality drift* caused by the application of refactorings.

This paper is structured as follows. Section 2 proposes the refactorings and case study we have selected for our experiments. Section 3 introduces our extended tree representation of the program structure. Section 4 shows how we can describe the impact of refactorings on the program structure. Section 5 uses this to analyse the impact of refactorings on object-oriented program quality metrics, and discusses the current limitations and their solutions. Section 6 discusses related work, and section 7 concludes.

2 Preliminaries

2.1 Selected refactorings

Fowler's catalogue [9] lists seventy-two object-oriented refactorings and since then many others have been discovered. To demonstrate the possibility of reasoning about refactoring in terms of their impact on program structure, we apply our formalism to a number of selected refactorings: ExtractMethod, EncapsulateField and PullUpMethod. These refactorings are quite typical for the categories of refactoring strategies they belong to - respectively Composing Methods, Organizing Data and Dealing with Generalization - which are among the most popular refactoring strategies in todays refactoring tools [IntelliJ IDEA, Eclipse, Together]. Hence they may serve as representatives for the complete set of primitive refactorings.

ExtractMethod extracts part of a method and factors it out into a new method.
EncapsulateField encapsulates public attributes by making them private and providing accessors. In other words, for each public attribute a method is introduced for accessing (getting) and updating (setting) its value, and all direct references to the attribute are replaced by calls to these methods.
PullUpMethod moves identical methods from subclasses into a common superclass.

2.2 Source code example

The example that we will use for our experiments consists of a simple Java package containing 4 classes: *Packet*, *Machine* and two subclasses *Workstation* and *PrintServer*. It is part of the implementation of a Local Area Network simulation (LAN). Although the code is in Java, other implementation languages could serve just as well, since we restrict ourselves to representing core object-oriented concepts only. For more information about this example, see [13].

```
01 public package LAN {
02  public class Machine {
03   public String name;
04   public Machine nextMachine;
05   public void accept(Packet p) {
06     System.out.println(name
07                   + " is accepting "
08                   + nextMachine.name);
09     this.send(p); }
10   protected void send(Packet p) {
11     System.out.println(name
12                   + " is sending "
13                   + nextMachine.name);
14     this.nextMachine.accept(p); }
15   }

16  public class Packet {
17   public String contents;
18   public Machine originator;
19   public Machine addressee;
20   }
```

```
21  public class PrintServer
22              extends Machine {
23   public void print(Packet p) {
24     System.out.println(p.contents); }
25   public void accept(Packet p) {
26     if(p.addressee == this)
27       this.print(p);
28     else super.accept(p); }
29   }

30  public class Workstation
31               extends Machine {
32   public void originate(Packet p) {
33     p.originator = this;
34     this.send(p); }
35   public void accept(Packet p) {
36     if(p.originator == this)
37       System.err.println("no dest");
38     else super.accept(p); }
39   }
40  }
```

We will describe the three refactor operations by example, and discuss their intuitive impact on quality.

ExtractMethod can be performed to extract the println-statement from methods $Machine.accept$ and $Machine.send$ to a separate method $log(String)$. To do this, we have to introduce a new method log in the $Machine$ class, with a single parameter. We can then move the println-statement from the $accept$ or $send$ method, and replace the String literal with a reference to the parameter. Thereafter, we have to replace the println-statement in both the $accept$ and $send$ methods with a method call to the log method, passing to it the appropriate String literal. Intuitively, performing this refactoring generalises the println-statement, which increases code reuse and reduces the impact of changes. When the output format needs changes, we will only need to change the log body, and not the $accept$ or $send$ methods.

EncapsulateField can be performed in order to shield the attribute $Machine.nextMachine$ from direct references. This causes the introduction of two methods in class $Machine$: a 'getter' which accesses the attribute and returns it to the caller and a 'setter' which takes the new value of the attribute as a parameter and updates the attribute. The attribute itself is made private. Intuitively, shielding the attribute from direct references reduces data coupling. This allows the attribute to change its data format without affecting clients using the attribute value.

PullUpMethod can be applied to $PrintServer.print(Packet)$, so that future subclasses of $Machine$ such as a FileServer class can reuse this code. This requires the introduction of a method in superclass $Machine$, to which the body of the $PrintServer.print$ method will be moved, and the removal of the former $PrintServer.print$ method. Intuitively, pulling up a method generalises specific behaviour, making it possible for subclasses to reuse and specialise the behaviour.

While the impact of these example applications of refactor operations is dependent on the situation in which they are applied, the following section will explain how we can set up a formalism to describe this impact in a generic way, for all situations.

3 Representing the program structure

The way in which we represent software is very straightforward: the source code is transformed into an abstract syntax tree representation extended with cross-references.

3.1 Abstract syntax tree representation

The program syntax tree directly reflects the natural containment relationship. A system contains packages. A package contains classes (or recursively another package). A class contains attributes and methods. A method contains expressions, local attributes and parameters.

All the nodes in the abstract syntax tree have a specific type: **S**(ystem), **PA**(ckage), **C**(lass), **M**(ethod), **A**(ttribute), (actual) **P**(arameter or return value), **L**(ocal variable), **E**(xpression).

Figures of the AST of the example have been omitted as the concept is well known in the context of software engineering.

3.2 Abstract Syntax Tree extensions

On top of this abstract syntax tree representation, we need to superimpose extensions to represent cross-reference relationships between software entities (such as class inheritance, method calls, attribute accesses and updates, type information). These are represented by *edges* between the corresponding tree nodes.

As for nodes, the superimposed edges have a specific type: **i**(nheritance), (method) **c**(call), (attribute or local variable) **a**(ccess), (attribute or local variable) **u**(pdate) and **t**(ype). The enrichment of the AST representation provided by these cross-references makes it easier to reason about code from the context of refactoring.

3.3 Program structure notation

We will now introduce a number of notations in terms of the program structure representation that are needed in the remainder of the paper to enable us to describe the impact of refactor operations.

Let r be an AST node, S_i a set, ν a node type, and ϵ a regular expression of edge types which adheres to the standard regular expression rules used in the popular UNIX grep-tool.

$\overline{S_i}$ denotes the complement of S_i.

$T(r)$ denotes the set of all nodes in the subtree with root *r*.

$ET(c, \epsilon)$ denotes the set of all edges incident to $T(c)$ of type ϵ, which are part of the Extended Tree.

$ET(c, \epsilon)_{inc}$ denotes those edges of $ET(c, \epsilon)$ which only have their target node in $T(c)$.

$ET(c, \epsilon)_{out}$ denotes those edges of $ET(c, \epsilon)$ which only have their source node in $T(c)$.

$ET(c, \epsilon)_{int}$ denotes those edges of $ET(c, \epsilon)$ which have both their source *and* target node in $T(c)$.

$\#S_1$ denotes the number of elements in set S_1.

$\nu(S_1)$ denotes the set of all nodes of type ν contained in the set S_1.

$S_1 \xrightarrow{\epsilon} S_2$ denotes the set of edges of type ϵ whose source node belongs to node set S_1 and whose target node belongs to node set S_2.

$target(S_1 \xrightarrow{\epsilon} S_2)$ denotes the set of target nodes of the given edge set.

$\Delta(S_i)$ denotes the change in S_i due to the application of a refactoring.

For example, $\#$**A**(*T(Machine)*) denotes the number of **A**-nodes (i.e., attributes) recursively contained in the subtree with root *Machine*. Table 1 describes the AST representation of the *T(Machine)* subtree, including our extension.

As another example, $\#ET(r, [au])_{out}$ denotes the number of **a**- and **u**-edges (attribute accesses and updates) from a node in the subtree of *r* to a node outside the subtree of *r*.

Table 1 describes the structure of the AST extension for the subtree T(*Machine*) by counting the superimposed cross-reference edges. Empty fields in the table indicate the impossibility of those occurences of those specific edges. For example, **i**-edges will never occur inside a class tree since inheritance only makes sense between two different classes. We will describe the AST and its extension subsequently. These AST descriptions (as illustrated in Table 1) will play a vital role in the formalism explained in the next section.

Class $Machine$ contains two methods and three attributes (implicit *this* attribute). Each of the two methods has one actual parameter, and one local variable (temporary string-variable). The number of expressions is irrelevant for the purposes of this paper, yet also provided on the left of Table 1.

Classes $Workstation$ and $PrintServer$ derive from $Machine$, represented by two incoming inheritance references. Attribute $name$ and the local variable in each method are of type $String$, which together with the two method parameters of type $Packet$ brings the number of outgoing type references to five. The only internal type reference is due to attribute $nextMachine$ of type $Machine$. Each method of $Machine$ calls the addition operator twice, println once, and the other method of $Machine$, which together makes eight (2x4). The $accept$ and $send$ method perform six and five internal accesses respectively, and each a single outgoing access. No updates are performed. This summarises the AST extension on the right of Table 1.

type	$\Delta T(c)$	type ϵ	$\#ET(Machine,\epsilon)_{int}$	$\#ET(Machine,\epsilon)_{inc}$	$\#ET(Machine,\epsilon)_{out}$
M	2	**i**		**2**	**0**
A	3	**t**	**1**	**2**	**5**
P	2	**c**			**8**
L	2	**a**	**11**		**2**
E	24	**u**	**0**	**0**	**0**

Table 1. Description of the AST (on the left, in terms of node types) and cross-references (on the right) of class $Machine$.

4 Describing the impact of refactorings on program structure

In this section, the impact of our selected refactorings on program structure is described. We split up the effect description in an impact on the AST representation, and an impact on its cross-references (from now on all together called the extended tree representation).

ExtractMethod($setE$:Set(E),m_1:M, m_2:String, c_1:C)

First, ExtractMethod introduces a new method m_2 in class c_1, for which possibly a number of actual parameters and a return value are required. A return value for m_2 is necessary when exactly one (multiple is not allowed) local variable or actual parameter of m_1 was updated by one of the expressions of $setE$. We calculate the number of new actual parameters of m_2 as the number of accessed local variables or actual parameters of m_1, even though Fowler [9] indicates not to create actual parameters for those local variables or actual parameters of m_1 whose first reference is an update (and are therefore immediately overwritten). The elaboration would contribute little to the overall goal of this paper and is left as an exercise for the enthousiastic reader. Consequently, we introduce no new local variables for method m_2.

The set of expressions $setE$ is copied to the new method m_2, and replaced inside m_1 by a method call to m_2 (1 **E**-node), with an actual parameter for each of the n accessed local variables or actual parameters of m_1 (n **E**-nodes). In case a local variable or actual parameter of m_1 was updated in $setE$, an extra expression is required to update that local variable or actual parameter (1 **E**-node). This summarises the impact on the AST of class c_1 as described on the left in Table 2.

Second, the introduction of new actual parameters and return value for method m_2 causes the introduction of cross-references to the types of those parameters, being either class c_1 itself - introduces an internal type reference - or another class - introduces an outgoing type reference. Naturally, ExtractMethod causes m_1 to call m_2 adding an extra internal call reference. Passing the arguments for the method call and returning the return value causes an increase of the number of internal access and update references, which summarises the right part of Table 2.

A superficial observation might lead to the interpretation that the application of ExtractMethod makes the program structure more complex in terms of our extended tree representation. Yet, these descriptions consist of both copying the expressions to the new method and thereafter replacing the set of expressions. Multiple applications of ExtractMethod on identical sets of expressions therefore only cause the former step to be performed once, while removing duplicate code by performing the latter step multiple types, as can be illustrated by extracting the println-statement from both $Machine.send$ and $Machine.accept$ in the example. Performing the last step multiple times removes code duplication.

type	$\Delta T(c_1)$
M	1
A	0
P	$\#target(T(setE) \xrightarrow{[au]} T(m_1))$
L	0
E	$1 + \#target(T(setE) \xrightarrow{a} T(m_1))$ $+\#target(T(setE) \xrightarrow{u} T(m_1))$

ϵ	$\Delta ET(c_1, \epsilon)_{int}$	$\Delta ET(c_1, \epsilon)_{out}$
t	$\#target(T(setE) \xrightarrow{[au]t} \{c_1\})$	$\#target(T(setE) \xrightarrow{[au]t} \overline{T(c_1)})$
c	1	0
a	$\#target(T(setE) \xrightarrow{a} T(m_1))$	0
u	$\#target(T(setE) \xrightarrow{u} T(m_1))$	0

Table 2. Impact of $ExtractMethod(setE, m_1, newMethod, c_1)$ refactoring on class c_1.

EncapsulateField(c_1:C,$attr$:A,$getter$:String,$setter$:String)

First, EncapsulateField introduces a *setter* and *getter* method, which respectively updates and accesses the attribute *attr* inside class c_1. Therefore, two methods and two parameters (actual parameter for setter and return value for setter) are added. As each new method consists of one expression (access or update), two expressions are added. This summarises the impact on the AST of class c_1, as described on the left in Table 3.

Second, the creation of the two new methods introduces a type edge from the actual parameter of the *setter* and one from the return value of the *getter* method to the type attribute, which can be either class c itself or another class. Then all former accesses and updates to the attribute $attr$ are replaced respectively by parameterless method calls to the *getter* and method calls to the *setter* method with the new value as an actual parameter. Finally, the *getter* method will update the return value with an access to the attribute, and the *setter* method will update the attribute with an access to the actual parameter.

type	$\Delta T(c)$
M	2
A	0
P	2
L	0
E	2

ϵ	$\Delta ET(c_1, \epsilon)_{int}$	$\Delta ET(c_1, \epsilon)_{inc}$	$\Delta ET(c_1, \epsilon)_{out}$	$\Delta \overline{ET(c_1, \epsilon)}$
t	$2 * \#(\{attr\} \xrightarrow{t} \{c_1\})$	0	$2 * \#(\{attr\} \xrightarrow{t} \overline{T(c_1)})$	0
c	$\#(T(c_1) \xrightarrow{[au]} \{attr\})$	$\#(\overline{T(c)} \xrightarrow{[au]} \{attr\})$	0	0
a	$-\#(T(c_1) \xrightarrow{a} \{attr\}) + 2$	$-\#(T(c) \xrightarrow{a} \{attr\})$	0	$-\#(\overline{T(c)} \xrightarrow{a} \{attr\})$
u	$-\#(T(c_1) \xrightarrow{u} \{attr\}) + 2$	$-\#(T(c) \xrightarrow{u} \{attr\})$	0	$-\#(\overline{T(c)} \xrightarrow{u} \{attr\})$

Table 3. Impact of $EncapsulateField(c_1, attr, getAttr, setAttr)$ refactoring on class c.

PullUpMethod($setM$:Set(M), $setC$:Set(C), c_s:C)

As PullUpMethod impacts subclasses c_i (with identical methods m_i) and superclass c_s, we will describe each of them separately and begin with the impact on the superclass.

First, PullUpMethod introduces a method m_s in superclass c_s with actual parameters and return value identical to those of m_i. The complete body of one of the identical methods m_i of subclasses c_i is copied to method $c_s.m_s$, which includes the local variables and expressions. This summarises the impact on the AST of superclass c_s, as described at the top of Table 4.

Second, the copying of these expressions causes all cross-references to be copied as well, consisting of type references, method calls, accesses and updates. This also means that former edges of the subclass method towards the superclass become internal edges of the superclass, and former edges from the subclass method towards other classes to become outgoing edges of the superclass. This impact is described at the bottom of Table 4.

The impact on any subclass c_i is identical, being the opposite of the impact on the superclass. The identical subclass methods m_i are removed, causing all actual parameters, local variables and expressions to be removed as well. Analogue, internal cross-references of $c_i.m_i$ are erased. The remainder of Table 5 describes the transformation as explained for the superclass.

Concluding, this section described the impact of the three refactorings on our extended AST representation of the source code. In the next section, we will introduce the formalisation of object-oriented program quality metrics on top of the extended tree representation.

type	$\Delta T(c)$
M	1
A	0
P	$\#P(m_i)$
L	$\#L(m_i)$
E	$\#E(m_i)$

ϵ	$\Delta ET(c_s,\epsilon)_{int}$	$\Delta ET(c_s,\epsilon)_{inc}$	$\Delta ET(c_s,\epsilon)_{out}$	$\Delta\overline{ET(c_s,\epsilon)}$
t	$\#(T(m_i) \xrightarrow{t} \{c_s\})$	$-\#(T(m_i) \xrightarrow{t} \{c_s\})$	$\#(T(m_i) \xrightarrow{t} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{t} (\overline{T(c_i)} \backslash T(c_s)))$
c	$\#(T(m_i) \xrightarrow{c} \{c_s\})$	$-\#(T(m_i) \xrightarrow{c} \{c_s\})$	$\#(T(m_i) \xrightarrow{c} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{c} (\overline{T(c_i)} \backslash T(c_s)))$
a	$\#(T(m_i) \xrightarrow{a} \{c_s\})$	$-\#(T(m_i) \xrightarrow{a} \{c_s\})$	$\#(T(m_i) \xrightarrow{a} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{a} (\overline{T(c_i)} \backslash T(c_s)))$
u	$\#(T(m_i) \xrightarrow{u} \{c_s\})$	$-\#(T(m_i) \xrightarrow{u} \{c_s\})$	$\#(T(m_i) \xrightarrow{u} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{u} (\overline{T(c_i)} \backslash T(c_s)))$

Table 4. Impact of $PullUpMethod(setM, setC, c_s)$ refactoring on superclass c_s.

type	$\Delta T(c_i)$
M	-1
A	0
P	$-\#P(T(m_i))$
L	$-\#L(T(m_i))$
E	$-\#E(T(m_i))$

ϵ	$\Delta ET(c_i,\epsilon)_{int}$	$\Delta ET(c_i,\epsilon)_{out}$	$\Delta\overline{ET(c_i,\epsilon)}$
t	0	$-\#(T(m_i) \xrightarrow{t} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{t} \overline{T(c_i)})$
c	0	$-\#(T(m_i) \xrightarrow{c} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{c} \overline{T(c_i)})$
a	$-\#ET(m_i,a)_{int}$	$-\#(T(m_i) \xrightarrow{a} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{a} \overline{T(c_i)})$
u	$-\#ET(m_i,u)_{int}$	$-\#(T(m_i) \xrightarrow{u} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{u} \overline{T(c_i)})$

Table 5. Impact of *PullUpMethod(setM, setC, c_s:C)* refactoring on any subclass c_i.

5 Analysing the impact on object-oriented program quality metrics

We will now illustrate how we can use our previous results to analyse the impact of refactorings on object-oriented software metrics. This is crucial to assess the impact of refactorings on program quality, since software metrics are typically used as internal quality factors [8].

The general idea is quite simple: we can formally specify object-oriented program quality metrics in terms of the extended AST of the program structure presented earlier. As such, the impact of a refactoring on the program structure, as denoted in the impact tables provided in the previous section, can be directly translated into the impact of a refactoring on the object-oriented program quality metrics. This formalism for defining metrics is analogue to the one provided in [14], where a graph-based formalisation of object-oriented software metrics is introduced.

5.1 Selected metrics

As the list of object-oriented program quality metrics is virtually endless (i.e. [26] alone describes more than 200 complexity metrics), and the page limit for this paper is not, we will focus on those program metrics which are most commonly used, being Number of Methods, Cyclomatic Complexity, Number of Children, Coupling Between Objects, Response For a Class and Lack of Cohesion among Methods. It can be argued that some of these metrics are not so much measures of program quality but of program size. Yet, as previous work from the context of formalizing object-oriented program quality metrics [2] uses similar primitives to calculate design quality metrics, we are confident that the current set provides a sound sample for the specific purpose of demonstrating the feasibility of using our formalism to investigate the quality drift caused by the application of refactorings. Moreover, [3] validated the Number of Children, Coupling Between Objects and Response For a Class metrics as quality indicators by investigating the relationship with fault probability.

Definitions for these metrics are:

Number of Methods calculates the number of methods of a class. It is an indicator of the functional size of a class.
Cyclomatic Complexity counts the number of possible paths through an algorithm. It is an indicator of the logical complexity of a program, based on the number of flow graph edges and nodes [7].
Number of Children measures the immediate descendants of a class [5]. It is an indicator of the generality of the class.
Coupling Between Objects is a measure for the number of collaborations for a class [18]. It is an indicator of the complexity of the conceptual functionality implemented in the class.
Response For a Class is the number of both defined and inherited methods of a class, including methods of other classes called by these methods [5]. It is an indicator of the vulnerability to change propagations of the class.

Lack of Cohesion among Methods is an inverse cohesion measure (high value means low cohesion). Of the many variants of LCOM, we use LCOM1 as defined by Henderson-Sellers [10] as the number of pairs of methods in a class having no common attribute references. It is an indicator of how well the methods of the class fit together.

Table 6 formalises these metrics on top of our source representation. This will allow us to project the impact of refactorings on program structure - as described in the previous section - in the area of software quality.

Metric	Formula	Metric($Machine$)
NOM	$\#M(T(c))$	2
CC	insufficient model information	/
NOC	$\#ET(class, i)_{inc}$	2
CBO	$target(ET(class, t)_{out} \cup ET(class, [au][tm])_{out} \cup ET(class, cm)_{out})$	4
RFC	Assume $setM = M(target(ET(class, i*)_{out}) \cup \{class\})$ then $RFC = \#(setM \cup \{m_2 \mid \exists m_1 \in setM \wedge m_2 \in target(ET(m_1, c)_{out})\})$	3
LCOM	$\#\{\{m_1, m_2\} \mid m_1, m_2 \in M(T(c)) \wedge m_1 \neq m_2 \wedge target(T(m_1) \xrightarrow{[au]} T(c)) \cap target(T(m_2) \xrightarrow{[au]} T(c)) = \emptyset\}$	0

Table 6. Formalization of selected metrics on top of our source representation, calculated for the $Machine$ class from the example of section 2.2.

The formalizations provided in Table 6 are defined in terms of our extended tree representation, which is a formal description of program structure. This allows an analysis of the impact of refactorings on internal program quality metrics, by translating the structural changes to the program structure, as described in the impact tables of section 4, into changes on the various metrics.

5.2 Analysing the impact of refactorings

For the purpose of clarifying whether the internal quality (represented by the metric) increases or decreases, we need to analyse in which direction this drift could occur. Therefore, we categorise the effects on a metric value in the following three categories (analogue to the work presented in [23]):

Impact	Symbol	Range of effect on metric value
nil	0	[0,0]
positive	+	[0,+∞[
negative	-	]-∞,0]

A *nil* impact represents a structural change which *will never* affect the value of the internal program quality metric. A *positive* impact represents a structural change which *might increase* the value of the internal program quality metric or leave it unchanged, yet can never decrease it. Lastly, a *negative* impact represents a structural change which *might decrease* the value of the internal program quality metric or leave it unchanged, yet can never increase it.

In order to illustrate our technique of analysing the impact of refactoring on internal program quality metrics, we elaborate on the most interesting metrics.

As the metric formalizations, denoted in Table 6, are constructed out of a number of different terms, we can analyse the impact of a refactoring on the metric value by analysing its impact on these various terms. To do this, we split out the different terms, and use the impact tables provided in section 4 to identify the impact category (nil, positive or negative).

Table 7 analyses the impact of the refactorings on the Coupling Between Objects metric value by clarifying the potential influence of each refactoring on the different terms of the metric formalization (analogue tables are provided for Response For a Class and Lack of Cohesion among Methods in tables 8 and 9). When the impact of a refactoring is positive for at least one term, and negative for none (nil impacts allowed), the total impact of the refactoring on the metric value is a positive impact (potentially cause the metric value to increase). Conversily, when the impact of a refactoring is negative for at least one term, and positive for none (nil impacts allowed), the total impact of the refactoring on the metric value is a negative impact

Refactoring	$\Delta target(ET(c,t)_{out})$	$\Delta ET(c,cm)_{out}$	$\Delta ET(c,[au][tm])_{out}$	CBO impact
ExtractMethod	+	0	0	+
EncapsulateField	0	0	0	0
PullUpMethod-Superclass	+	+	+	+
PullUpMethod-Subclass	-	-	-	-

Table 7. Analysis of factors which could cause drift of the CBO metric value.

Refactoring	$M(T(c))$	$\Delta target(ET(c,i*)_{out})$	$\Delta target(ET(c,c)_{out})$	RFC impact
ExtractMethod	+	0	0	+
EncapsulateField	+	0	0	+
PullUpMethod-Superclass	+	0	+	+
PullUpMethod-Subclass	-	0	-	-

Table 8. Analysis of factors which could cause drift of the RFC metric value.

Refactoring	$M(T(c))$	$\Delta target(ET(c,[au])_{int})$	RFC impact
ExtractMethod	+	+	+
EncapsulateField	+	0	+
PullUpMethod-Superclass	+	+	+
PullUpMethod-Subclass	-	-	-

Table 9. Analysis of factors which could cause drift of the LCOM metric value.

(potentially causes the metric value to decrease). Two exceptions arise in the reasoning about the impact of a refactoring on a metric value.

First, the selection of the target-nodes of a set of edges inside the metric formalization makes the analysis more complex. It requires semantical reasoning about whether the removal of an edge from a set of edges setE also reduces target(setE). This is an important issue as most of our metric formalizations explicitly depend on the target of a set of edges. I.e. while EncapsulateField increases the number of type-edges departing from the class subtree, it does not affect the target of this set of edges (the classes of which an instance was referenced). This semantic information is lacking from the impact tables as they provide a quantititative description of the change to the cardinality of the entities of the extended tree representation. In the next section, we will describe how to make the analysis of impacts in these situations more easy.

Second, when a refactoring has a different impact on the various terms of a metric value (positive for some, negative for others), a deeper semantical analysis is required, possibly even up to the level of inspection of the specific source code context. This limitation is also discussed in detail in the next section.

The result of this impact analysis is summarised in Table 10. We verified the impact catalogue by applying the refactorings on the LAN example and comparing post- and pre-refactoring measurements, as done in [11]. The drift noticed in these comparisons confirmed our formal analysis.

This impact catalogue can be used as an a-priori feedback on the efficiency of applying specific refactorings, from the perspective of various internal program quality metrics. In example, the table clarifies that applying the Pull Up Method refactoring has an impact which is opposite for the superclass and the subclass. While it potentially decreases the metric values for the internal program quality metrics Number of Method, Coupling Between Objects, Response For a Class and Lack of Cohesion among Methods of the subclass, it potentially increases these metric values of the superclass. This is a detailed description for the fact that the quality drift on the superclass, caused by moving a method up the inheritance hierarchy, is the inverse of the quality drift on the subclass. This allows us to envision that when we want to improve the quality of the subclass, this could possibly cause a deterioration of the superclass quality.

The next section discusses the current limitations of using this technique to tackle the question of quality drift caused by the application of refactoring, and elaborates on a their solutions.

Refactoring	NOM	NOC	CBO	RFC	LCOM
EncapsulateField	+	0	0	+	+
PullUpMethod subclass	-	0	-	-	-
PullUpMethod superclass	+	0	+	+	+
ExtractMethod	+	0	0	+	+

Table 10. Refactoring impact table indicating the impact of a particular refactoring on a particular class quality metric

5.3 Limitations and solutions

Our technique for analysing the impact of refactorings on internal program quality metrics allows the clarification of the drift of specific internal program quality metrics, as caused by the application of particular refactorings. While some early results were presented which demonstrated the feasibility of applying this technique for a number of refactorings and a number of internal program quality metrics, it is clear that the applicability of the technique has a number of limitations.

First, our representation for program structure is a limiting factor, as the impact analysis can only use the information contained in this program representation. We found an example of an internal program quality metric on which the impact of refactorings could not be analysed due to lacking model information (no control flow information in our program structure representation). A solution to this problem could be to simply extend our model, yet this will inevitably make our model more complex. Moreover, the metamodels used in related research on the formalization of metrics demonstrates that most of the measures of the current metric suites can be operationally defined on a program structure representation similar to ours [1, 4, 19, 13]. A detailed investigation of how this limitation reduces the number of internal program quality metrics on which the impact can be analysed requires a deeper study on the operational definitions of currently known program quality metrics.

Second, our formal descriptions of the structural changes on the program structure, expressed in terms of an extended tree representation, lacks semantical information about the sources and targets of the cross-reference edges which are added or removed during the refactoring. This information is currently implicitly contained in the informal description of the refactorings. Therefore, one of the lessons we learned is that a complete formal description of the structural changes caused by the application of a refactoring is required in order to automate the impact analysis process, i.e. using logic engines such as Prolog. While necessary for the next step of analysis of a more extended set of refactorings, the scale of our current work, serving the purpose of a proof-of-concept, did not require automated analysis.

Summarizing, we identified solutions for the two major limitations of our technique, which will simplify the analysis of the impact of refactorings on internal program quality metrics, making it possible to automate the impact analysis process. Such an automation is essential to cope with the massive amount of combinations between refactorings and internal program quality metrics.

6 Related work

Formalisations of software metrics have been provided from the mathematical perspective [1, 4, 19, 13] and the formal specifications perspective, in example Z [15] and OCL [2]. Our formalism is analogue to the mathematical approach of [1], yet they do not provide a formal metamodel specification but rely directly on the cardinality of informally described model features. We feel that the formal metamodel specification helps us in reasoning about program transformations. None of these metamodels incorporated information not contained in the model for program structure presented in this work (except information about modifiers such as abstract, final, public, protected, private).

In previous work, we introduced the existing research field of refactoring, and proposed an extensive list of directions for future research [12]. Most recently, an extension to the UML 1.4 metamodel for the purpose of facilitating refactoring at the UML level while remaining consistent with the source-code was proposed by members of our research group [25].

An experience report on metric collection during a refactoring phase is provided in [22]. A formalization of program transformations is introduced in [13], which formed the basis of this work. The same graph-rewriting foundation for describing refactorings is used in [24], which introduces a hierarchical representation for visualizing program structure.

The work which lies most closely to ours is provided in [23], in which the impact of meta-pattern transformations on an object-oriented metrics suite is provided. Our work is similar in that they are also interested in a-priori feedback on the impact of source-code transformation, and therefore also provide an impact catalogue of source code transformations on object-oriented metrics. Our work is different in that we focus on the impact analysis technique itself and therefore formalise the process of analysing the impact of catalogued refactorings provided by Fowler on internal program quality metrics, while they focus on the reengineering strategy of resolving design flaws through the application of meta-patterns.

A quantitative evaluation method to measure the maintainability enhancement effect of program refactoring is presented in [11]. They analysed three phases in the process of program refactoring, of which their contribution is towards the phase of validation of the refactoring effect. They analyse the effect of a number of refactorings on coupling metrics by pre- and post-refactoring measurements.

Detection of refactoring-candidates using visualisation techniques is introduced in [21]. Automatic detection of transformations is described in [20], in which rules for candidate selection are defined in terms of metric thresholds.

7 Conclusion and Future Work

Our technique for analysing the impact of refactorings on internal program quality metrics allows the clarification of the drift or improvement of specific internal program quality metrics, as caused by the application of particular refactorings. The results presented in this work demonstrated the feasibility of applying this technique for a number of refactorings and a number of internal program quality metrics. The limitations of our current approach were identified and solutions were discussed to resolve them.

In this paper, we presented both a formalism for describing and a technique for analysing the impact of refactorings on internal program quality metrics as indicators of quality factors. As a case study, the technique was applied to a number of representative refactorings from the refactoring categories Composing Methods, Organizing Data and Dealing with Generalization [9], and a number of commonly used internal program quality metrics (Number of Methods, Number of Children, Coupling Between Objects, Response For a Class, Lack of Cohesion among Methods).

The resulting classification of the impact in positive or negative contributions to internal quality metrics delivers a-priori feedback to software maintainers, enabling them to predict the quality drift caused by the application of (a series of) refactorings.

Our technique, improved by the suggestions to counter the limitations, remains to be applied to a more extended set of refactor operations and object-oriented program quality metrics, to form a catalogue of the impact of refactorings on internal quality metrics. Guided by this impact-catalogue on internal quality metrics, we plan experiments to gather empirical data about the impact of refactoring on external program quality metrics (performance, mean time between repair,...).

8 Acknowledgements

This research is funded by the FWO Project G.0452.03 “A formal foundation for software refactoring”. We thank Pieter Van Gorp, Hans Stenten, Serge Demeyer and Andy Zaidman for their useful comments on drafts of this paper.

References

[1] F. B. Abreu and R. Carapuca. Object-oriented software engineering: Measuring and controlling the development process. In *Proc. 4th Int’l Conf. Software Quality*, October 1994.

[2] A. L. Baroni. Formal definition of object-oriented design metrics. Master’s thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, Belgium, 2002.

[3] V. R. Basili and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

[4] L. C. Briand, J. Daly, and al. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[6] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, pages 44–49, August 1994.

[7] J. C. Coppick and T. J. Cheatham. Software metrics for object-oriented systems. In *Proceedings of the 1992 ACM annual conference on Communications*, pages 317–322. ACM Press, 1992.

[8] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.
[9] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
[10] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
[11] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. Int'l Conf. Software Maintenance*, pages 576–585. IEEE Computer Society Press, 2002.
[12] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.
[13] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proceedings First International Conference ICGT 2002, Barcelona, Spain.
[14] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
[15] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings Int'l Conf. OOPSLA '96*, ACM SIGPLAN Notices, pages 235–250. ACM Press, 1996.
[16] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[17] W. Opdyke and R. Johnson. Creating abstract superclasses by refactoring. In *Proc. ACM Computer Science Conference*, pages 66–73. ACM Press, 1993.
[18] R. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 2001.
[19] R. Reissing. Towards a model for object-oriented design measurement. In F. B. e Abreu, editor, *Proc. 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 71–84, 2001.
[20] H. A. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proc. International Conference on Software Maintenance*, pages 154–162, october 2000.
[21] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. European Conf. Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.
[22] E. Stroulia and R. V. Kapoor. Metrics of refactoring-based development: An experience report. In *Proc. of the 7th International Conference on Object-Oriented Information System*, pages 113–122. Springer Verlag, 2001.
[23] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proc. European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE Computer Society Press, 2003.
[24] N. Van Eetvelde and D. Janssens. A hierarchical program representation for refactoring. In *Proc. of UniGra'03 Workshop*, 2003.
[25] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of UML 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.
[26] H. Zuse. *Software Complexity*. Walter de Gruyter, Berlin, 1991.

J2EE or .NET: *A Managerial Perspective*

Neil Chaudhuri
Research Fellow
Logistics Management Institute
McLean, VA 22102 USA

Keywords: *software evolution, J2EE, .NET, project management, enterprise architectures, enterprise systems, web services*

Abstract

With the evolution of enterprise systems from the traditional client-server paradigm, Sun Microsystems' Java 2 Enterprise Edition (J2EE) and Microsoft .NET have emerged as fierce competitors for recognition as the leading choice for building enterprise solutions. After first engaging in a high-level discussion of the architectures, this paper describes the criteria by which project managers should choose between them. It concludes with a discussion of what continued evolution of the enterprise realm, namely towards the web services realm, may mean for project managers.

Introduction

The only certainty in the information technology industry is the rapid and constant evolution of software technology. Nowhere is this more evident than in the realm of enterprise systems, which evolved from client-server systems as a means of physically and logically decoupling the critical components of the architecture—namely the presentation, middleware, and database tiers. Not surprisingly, the size and complexity of enterprise systems demand sophisticated solutions. Assembled from its previously established technologies, Sun Microsystems' Java 2 Enterprise Edition (J2EE) architecture enjoyed prominence as the leading choice for building enterprise solutions. Never to be outdone in its effort to maintain its perch atop the information technology industry, Microsoft Corporation unveiled an alternative enterprise-level solution, .NET, which also represents the next generation of its own previously established technologies. As these solutions have evolved to meet the needs of the evolving enterprise, software engineers on both sides have inundated the literature with perspectives on which is technically superior. To this point, however, project managers charged with overseeing the development of enterprise systems have been largely left out of the discussion and consequently have been unable to engage in any meaningful process of natural selection. This paper provides a comparative overview of J2EE and .NET and describes the criteria by which project managers should choose between them as the solution for an enterprise-level development effort. Finally, this paper suggests how the technologies themselves and project managers' understanding thereof may evolve over time as the enterprise realm continues to evolve--primarily towards web services.

Overview of the Architectures

J2EE

It is the most basic quality of Sun's J2EE architecture that is the initial source of confusion for most project managers. J2EE is *not* a product; rather, it is a *specification*. With each successive specification since the first was issued in 1999, various vendors allied with Sun have built application servers that conform to it, and J2EE applications are deployed to these servers [5]. Among the more notable are IBM's WebSphere, Oracle's OC4J, and BEA's WebLogic Server, which is generally considered the industry leader. During its lifetime J2EE has gained a measure of credibility as a viable option in mission-critical systems, for United Airlines and American Express are two prominent examples among many organizations that have successfully implemented J2EE solutions [7].

With regards to the technical details of J2EE, at its core rests the Java programming language. Over nearly two decades the object-oriented paradigm has become preeminent among programming languages; and in turn Java, another Sun specification, has become preeminent among object-oriented languages. This played no small role in the emergence of J2EE in the enterprise software realm. An even more significant contributor to its marketability is the portability of the Java code that comprises a J2EE application. For example, because both servers meet the J2EE specification, code deployed on WebLogic can be deployed seamlessly on WebSphere should the need arise, and both can operate on any platform (*e.g.* Windows, Linux, *etc.*). Therefore, J2EE does not render an organization vulnerable to the whims of a single vendor. This may be the most powerful argument in favor of J2EE. However, there are some caveats that to be explored a bit later.

While literature concerning the J2EE architecture has been prolific over the last few years, that concerning its components has been even more so. The J2EE specification consists of a potpourri of various technologies, each with its own specification. Moreover, some of these actually predate the first J2EE specification. Therefore, if the age of the J2EE architecture as a whole is an argument for its reliability, the claim is further fortified by the age of its components. The most notable among these are the following:

- **Servlets** and **Java Server Pages** (**JSP**s) for generating web content
- **Java Database Connectivity** (**JDBC**) for storing (or in the vernacular of enterprise architecture, *persisting*) data in databases
- **Enterprise JavaBeans** (**EJB**s) for business logic processing in the middle tier. This is the centerpiece of the J2EE architecture.

As a result of the fragmented nature of J2EE, vendors may produce servers that do not comply with the whole of the J2EE specification but rather with portions thereof. Such servers thus cannot support a complete J2EE application but may still be very useful. A prominent example is Apache's Tomcat, an industry-leading, open-source web server that only supports the servlet and JSP specifications. Hence it is best suited for client-server applications, which remain significant even with the emergence of the *n*-tier paradigm.

Yet despite its excellent performance to that end, Tomcat by itself is quite ill-suited to enterprise architectures.

.NET

While it is designed to solve the same problems as J2EE, Microsoft's .NET architecture takes a starkly contrasting approach to enterprise systems development. The most obvious point of contrast is the status of .NET as a *product* of Microsoft and Microsoft alone as .NET at its core does not rest upon alliances with other vendors. Yet the most significant point of contrast between .NET and J2EE is that the former is virtually brand new. Technically, .NET has been available since 2001, but it has undergone so many modifications since that it is difficult to gauge its readiness as a viable option for mission-critical solutions [3].[1] However, as the giant in the information technology industry, Microsoft has a longstanding reputation, especially within the American government, for producing working solutions. Furthermore, its massive support structure is at the disposal of those who choose to adopt a .NET solution. Therefore, the longevity and stability of Microsoft Corporation goes very far in offsetting the apparent lack thereof in its .NET architecture.

With regards to the technical details of .NET, at the heart of the architecture rests not one but in fact several programming languages. All of the so-called ".NET family of languages" are object-oriented, so an organization need not abandon the paradigm should it choose to pursue this architecture. There are two prominent languages in the family. The first is Visual Basic .NET (VB .NET), which is based on the popular Visual Basic language that has become familiar to countless developers over the last several years. The second is C# (pronounced *C sharp* as in music), a brand new language created by Microsoft. Although the claim is that C# is the next generation in the evolution of the popular C++ programming language, the influence of Java is unmistakable. C# is the centerpiece of the .NET family of languages, and it provides the most effective use of .NET capabilities [1].

The most significant consequence of committing to a .NET architecture is restriction to the Windows platform. Microsoft is often criticized for its reluctance to build products that integrate seamlessly with those from other vendors, and .NET does nothing to assuage the criticism. Yet the Windows operating system represents the very means by which Microsoft ascended to its perch atop the information technology industry. Therefore with the incalculable number of Windows-based systems in operation throughout the world, many organizations would consider a restriction to Windows no restriction at all. Even still, there is a series of open-source initiatives towards moving .NET to other platforms, but they are far from complete [1].

The components of the .NET architecture are the closest point of similarity to the J2EE architecture. The most notable among these are the following:

- **ASP .NET** for web content and based largely on the popular Active Server Page (ASP) technology developed by Microsoft
- **ADO .NET** for data persistence and based largely on the popular ActiveX Data Object (ADO) technology developed by Microsoft
- **Windows Forms** (or **WinForms**) for graphical user interfaces (GUIs) utilized on client machines and based largely on the popular Visual C++ and Visual Basic technologies developed by Microsoft.[2]
- **COM+** (or **Enterprise Services**) for business logic processing in the middle tier and based largely on the Component Object Model (COM) technology developed by Microsoft [4]

From this list one can discern that .NET has hardly emerged from a vacuum. Microsoft is clearly hoping to lure its vast following to its latest innovation by creating next-generation implementations of its prior successes—staples of the industry like ASP, ADO, and COM—and assembling them into its service-based, evolved .NET architecture.

Development and Deployment

J2EE

With the completion of this brief overview of the J2EE and .NET architectures, it is time to explore them in more depth and consider how they compare in development and deployment. The initial step in developing a J2EE application is acquisition of an application server[3], and the best ones, like the aforementioned WebLogic Server, are quite costly. There are cheaper alternatives—including the open-source JBoss available at no cost—but these lack the support mechanisms that can prove invaluable during the course of a project. Ultimately, a leading application server will prove the better value over time despite the heavy cost upfront, but project managers should choose wisely. While the portability of J2EE code across servers is a powerful feature, it is offset by their cost. Indeed, as expensive as a single server may be, to move to another would deplete all but the most lavish budgets. This is simply another case where the reality of the marketplace thwarts the idealism of a technology.

Despite their cost, J2EE application servers provide so many services that they are valuable assets to any development effort. Notable among these is a Java Runtime Environment (JRE), the Sun-specified realm in which all Java applications run. The JRE spares application developers from low-level tasks like memory management which can be excruciatingly difficult to implement. All servers are also endowed with the standard J2EE Application Programming Interfaces (API's) for designing code as well as proprietary API's, which merit particular attention in this discussion.

While all application servers behave according to the standard dictated by the J2EE specification, the underlying implementations are not standard. Thus, the inevitable idiosyncrasies across servers can cause the same code to run faster on one than another. Server vendors therefore provide their own APIs to optimize certain operations like database accesses, and these can lend a rather significant boost to performance. However, these

[1] According to *.NET Magazine*, TRX Travel Services, a provider of reservation-processing services to the travel industry, recently migrated its legacy systems to .NET with excellent results primarily in the areas of scalability and performance [6].

[2] It should be noted that the Java programming language has a similar mechanism in the form of the Abstract Windowing Toolkit (AWT), Swing, and the new Standard Widget Toolkit (SWT). Technically, however, this client-side functionality rests outside the realm of J2EE.

[3] Throughout the course of the discussion on J2EE, the terms *application server* and *server* will be used interchangeably, and both will refer to platforms that meet the J2EE specification.

should be used only when absolutely necessary, for the performance gain comes at the expense of portability. For example, while OC4J database APIs may increase performance by 40%, they will simply not function on WebLogic, and the API-based code would have to undergo an inevitably costly revision if a switch were made. Therefore, heavy reliance on proprietary APIs will all but shackle your organization to a particular vendor's application server, and this negates the single greatest advantage J2EE has over .NET [1]. Project managers must weigh the benefits of both approaches and choose which makes the most sense for the application.

Integral to the services provided by J2EE servers are deployment descriptors, Extensible Markup Language (XML) files which allow critical functionality to be defined without the need for any Java expertise. With only a mere text editor, one can configure essential and otherwise painstakingly difficult services like load balancing and database connection pooling. Moreover, should the requirements for these services change, only the deployment descriptors have to be modified while the code remains untouched. The time that is saved allows developers to focus on the code supporting the business logic of the application rather than that supporting low-level services.

Should a J2EE solution employ EJB's, which is more than likely, deployment descriptors may provide two vital services beyond those previously described. The first concerns data persistence. Charged with this task is a category of EJBs known as *entity beans*. One would think that developers must endow their entity beans with persistence code that utilizes the pervasive but at times complicated Standard Query Language (SQL). Indeed, developers have this option. However, should they so choose, developers may in fact forego writing a single line of persistence code and instead direct the application server to manage persistence.[4] This is achieved by editing proprietary deployment descriptors and specifying data persistence strategies therein. While time must be invested to determine the precise manner in which a particular vendor demands its descriptor to be modified, it is quite easily offset by the time saved by not having to generate the Java and SQL code necessary to manage persistence.[5] Furthermore, the application server will also provide its own optimizations to the persistence strategies outlined in the descriptor. Hence, every effort should be made to have the server manage persistence, for time is saved and performance enhanced as well.

The other significant role that deployment descriptors play in EJB development is in transaction management. Simply put, *transactions* in the context of the enterprise are a chain of operations—almost invariably involving a database—that must all be successful for the transaction as a whole to be considered successful. In that case any database changes made during the course of the transaction are made permanent, or *committed* in the vernacular of enterprise architectures. If even one operation fails, however, the entire transaction fails. In that case all database changes made during the course of the transaction are nullified—or *rolled back* in the vernacular—and the database returns to its original state before the transaction. The concept of transactions is among the most powerful in the enterprise realm, and not surprisingly it is also among the most complex to develop. In a J2EE environment, developers have the option to write code to do this; but as with persistence, they may choose to edit deployment descriptors to call upon the application server to manage transactions. As neither task is trivial, it is quite a boon to developers that they may leave the daunting tasks of persistence and transaction management to the server while concentrating their time and energy on the complex business logic that drives enterprise applications.

When developing an application of any kind, it is necessary to consider carefully which brand of software—known as integrated development environments (IDE's)—will be utilized to write the code that will support it. In the context of J2EE, the situation with IDE's is exactly as with application servers. There are numerous options ranging from free to rather costly, and the number of features available in each is roughly proportional to its cost. Moreover, most organizations would be better served by investing in a leading IDE, for the features it provides will ultimately balance any high costs upfront that may be incurred. For example, while many IDEs like IntelliJ's IDEA provide developer-level functionality like automatic generation of skeleton code, others like Together's ControlCenter also provide architect- and manager-level functionality with Unified Modeling Language (UML) generation and configuration management tools, which may preclude the need for tools devoted to those tasks alone. Yet no matter how sophisticated the IDE, J2EE applications are so complex that development and deployment are hardly ever trivial. A thorough understanding of the subtle points of the architecture is required of all development teams if they are to prevail, and project managers must therefore not presume that investment in a leading IDE will by itself lead to success.

.NET

Development and deployment of .NET are in many ways much simpler matters. Like J2EE, .NET enterprise applications require investment in an application server from Microsoft--most likely Windows Server.[6] Otherwise, there is far less financial investment required than is generally the case for J2EE, for as is custom with Microsoft, the pieces of the architecture are available for free download. Most notable among these are the .NET Extensions to Microsoft's Internet Information Services (IIS) web server and the .NET Framework. The former is a rather trivial matter; as the name suggests, the .NET extensions simply augment the capabilities of the prevalent IIS infrastructure. The latter merits a more rigorous discussion.

Principal within the .NET Framework is the Common Language Runtime (CLR), which is essentially the equivalent of the Java Runtime Environment [3]. The Framework also includes the APIs for developing code in all of the .NET family of languages. The freedom in languages offers great flexibility and tremendous potential for code reuse, but managers should take heed. Having different modules in the same project coded in different languages

[4] It should be noted that although application servers can manage fairly sophisticated persistence code, there are instances when the code is just so complicated that developers have no choice but to write it themselves. Thankfully, these instances are rare.

[5] It is true that switching to another application server would demand the editing of another proprietary descriptor to enable it to manage persistence, but the time loss is probably insignificant when weighed with the benefits, including not having to modify a single line of code.

[6] It should be stressed that Windows Server is only required when utilizing COM+ objects for systems which are truly enterprise-level, the primary focus of this discussion [4]. Smaller systems do not require such an elaborate and somewhat costly infrastructure.

will likely result in a configuration nightmare. Moreover, it will limit accessibility to the code base among the development team. For example, a VB .NET developer will be helpless should the need arise to modify C# code developed by a colleague for the same project. Therefore, project managers should only take advantage of the language freedom provided by .NET in the planning stages of a project and designate a single language as the development standard.

Unlike J2EE, there are few choices for .NET IDE's, and there is a distinct leader in the field: Microsoft's own Visual Studio .NET.[7][8] The tool is expensive, but like the best J2EE development tools, it offers many services like configuration management. Also, borrowing from its successful past, Microsoft endows Studio with both the time-tested drag-and-drop methodology for visually designing applications and GUIs for specifying the properties of objects like the location of a backend database. Moreover, each of these operations results in automatically generated code. Therefore, developers can design a user interface very quickly, and they are spared having to generate the code related to look-and-feel and other more trivial concerns and may instead focus on the business logic. Lest one believe that this is without its cost, however, one must be aware that there are times when it is necessary to understand and modify the generated code to optimize performance, and this may not be an easy task.

As with J2EE, .NET applications contain deployment descriptors, but they do not play a role nearly as significant as that played by their counterparts. Also XML files, .NET descriptors provide the expected services like load balancing and database connection pooling, but otherwise they lack the sophistication of J2EE descriptors. .NET deployment descriptors do not endow COM+ with persistence management capabilities, and this leaves the responsibility for this in the hands of developers [8]. On the other hand, .NET does indeed support declarative (*i.e.* non-programmatic) transaction management, but it is in the form of attributes placed physically in source code files rather than in deployment descriptors or any other kind of configuration files. Although .NET deployment descriptors do not provide the same level of services as those in J2EE, it is reasonable to expect that Microsoft will address this as .NET matures over time.

Choosing Between the Architectures

Understanding the manner in which J2EE and .NET applications are developed and deployed provides the foundation for a discussion of how project managers should choose between them when planning the development phase of a task. There are several critical points to consider, and managers must understand and prioritize them in order to make the right choice.

Establishing the Need for an Enterprise Solution

Foremost among the manager's responsibilities is to determine if the system in question truly represents an enterprise system. This may seem obvious, but it is alarming how often this is overlooked. Part of the problem is that the literature offers no single definition of what constitutes an enterprise system. It would appear that the consensus definition is an architecture comprised of more than two tiers and where each tier may have multiple components (*e.g.* multiple databases residing on different machines). As one might imagine, such a system is terribly complicated and naturally demands the enormous financial and philosophical commitment required by both J2EE and .NET. On the other hand, most systems are not enterprise systems, so it is wasteful to engage in an inevitably rigorous effort utilizing either technology. It is far more sensible instead to utilize individual components of J2EE and .NET—or perhaps even different technologies entirely like ColdFusion. Ultimately, neither a J2EE nor .NET application is trivial to build, so it behooves managers to ensure that the problem is complex enough to merit a complex—and expensive—solution.

Project Funding

Regardless of the project or the chosen solution, it goes without saying that funding is the paramount concern of project managers. Whether pursuing J2EE or .NET, managers can expect to allocate substantial financial resources to training, albeit for different reasons—J2EE because of its numerous component specifications and their rapid changes to meet the demands of the open-source community and .NET because of its own rapid changes in its effort to grow into a robust technology. Aside from training costs, each solution also has similar infrastructural costs associated with it. J2EE demands a large financial investment in licenses for a leading IDE and application server. .NET demands investment in Visual Studio to support application development, Windows Server to support COM+, and perhaps IIS to support web-based interfaces in the unlikely event the organization does not already have it [4].[9] It is difficult to say whose infrastructure is more costly, but it is important to note that these are one-time costs. However, managers who choose .NET at this stage will very likely incur recurring costs in the form of support requests to Microsoft because of the unavoidable flaws in the immature architecture [7]. Given all the variables, only a project manager with knowledge of the existing capabilities of the organization and the development team can decide which is the cheaper alternative.

Existing Client Infrastructure

Project managers must also consider the existing infrastructure of the client when choosing between the architectures. Microsoft solutions in the past have gained wide acceptance in the public sector of the United States. Consequently, public sector clients may not even entertain the possibility of a J2EE solution, and managers will have no choice to make. It would seem logical that the transition to .NET would be trivial, for Microsoft has claimed that legacy objects utilizing older Microsoft technologies will integrate seamlessly into .NET. This is technically true, but there is a significant caveat. Legacy objects imported into .NET run outside the CLR and thus have no access to its services (most notably, as mentioned previously, memory

[7] Throughout the course of the discussion on .NET, the terms Visual Studio .NET, Visual Studio, and Studio will be used interchangeably.

[8] A notable .NET IDE is Web Matrix, a free, open-source development tool that features many of the niceties of Visual Studio. However, it is only useful for the development of ASP .NET applications. Thankfully, web applications that would utilize ASP .NET are so pervasive that the restriction may prove negligible.

[9] .NET enterprise applications may require investment in Microsoft's BizTalk server as well, which serves to integrate the components of the system and is particularly valuable for integrating legacy systems [4]

management) [1]. Unmanaged legacy code, if poorly written, will cause the application to stumble or even fail. Ultimately, legacy objects will almost certainly have to be rewritten as .NET objects [1]. Hence if a rewrite is required anyway, and if the client is amenable to it, it may be advisable to consider a J2EE solution, which as mentioned previously will run on all platforms and thus make the existing infrastructure of the client a non-issue. Whatever the outcome, it is simply crucial that project managers understand that moving from legacy Microsoft solutions to .NET is not as simple a matter as it may seem.

Project Timetable

The factors discussed to this point offer no clear choice between J2EE and .NET because they are a function of circumstances unique to particular projects. However, there are two critical factors where the better choice is much more obvious. The first is the *timetable for completion* of the project. If the project schedule is short, then .NET is almost certainly the better choice. As discussed previously, Visual Studio offers numerous advantages towards Rapid Application Development (RAD). Even the most sophisticated J2EE IDE's cannot compete with Studio in mitigating the complexity of its component technologies and deployment procedures. Moreover, J2EE has a significant flaw in the context of RAD—the elaborate deployment procedures associated with a large, intricate solution are essentially the same as those associated with a small, simpler solution. This makes it difficult to produce systems quickly in J2EE, and .NET therefore has an apparent advantage when time is a significant concern. Of course, it should be noted that this advantage might be mitigated by the expertise of the development team. If an organization only has expert J2EE developers at its disposal, they will almost certainly be able to deploy a J2EE application quickly, and time lost training them in .NET will accrue no net benefit [1]. Thus while .NET lends itself much better to RAD than J2EE, the expertise of the development team can nullify this advantage.

Project Complexity

The other factor where the choice between J2EE and .NET is more obvious is the *complexity* of the project. If the requirements for an application demand a sophisticated solution (*e.g.* multiple servers, multiple backends), then J2EE is the better option. One reason is the ease with which J2EE integrates with multiple platforms. Another is the robustness of EJB's, which by means of deployment descriptors offer persistence and transaction management without the need for a single line of code. Of course, to use all of the features J2EE offers requires significant knowledge on the part of the development team, but when properly implemented these features provide tremendous value to the process. On the other hand, .NET has not yet proven that it has grown enough to meet the needs of a truly complex system [7]. Microsoft has always been somewhat reluctant to enable seamless integration of its products with those of other vendors, and an intricate enterprise system with many components will almost certainly require integration of products from multiple vendors. Moreover, as described previously, .NET's COM+ technology has yet to develop a framework for persistence and transaction management outside the code realm [8]. There is also the issue of Microsoft's poor reputation in the realm of security, which while exaggerated by the pervasiveness of Microsoft products is a significant concern in an enterprise where data are regularly moving over the network. Finally, .NET has only just begun to prove itself as a reliable solution for large-scale, mission-critical systems, and as a result it is impossible to predict just how it will respond to the demands of a particular enterprise. However, Microsoft's stature in the industry all but guarantees that .NET shall have ample opportunities to prove its mettle over time. Therefore, it is only with time that .NET will establish itself as a proven, robust enterprise solution.

Summary Remarks and the Future of Enterprise Software Evolution

During the course of this discussion, a high-level understanding of the components of the J2EE and .NET architectures and the manner in which they are developed and deployed laid the foundation for an examination of the critical points project managers must consider when choosing between them. The analysis led to the conclusion that neither choice is clearly superior in all cases. Rather, as each has its advantages over the other, the suitability of either architecture is a function of the particular circumstances surrounding the project. Understanding the strengths and weaknesses of J2EE and .NET will enable project managers to engage in a meaningful process of natural selection and produce successful results for their customers.

Of course, the very fabric of evolution is woven with the threads of innovation. Thus while the enterprise paradigm--and its implementation strategies in the form of the J2EE and .NET architectures--represent the latest innovation in software development, it is only natural to wonder where evolution will take the industry in the future. It would seem that there may already be an answer: web services.

The Emergence of Web Services

The concept of web services has dominated the literature for some time, yet the prolific rhetoric has actually served to obscure any legitimate understanding of what web services truly represent. Perhaps the best source for an accurate and complete definition of web services is the World Wide Web Consortium (W3C), the standards body that serves as the steward of XML and web services. The W3C provides the following:

> *Definition: A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols.* [9]

The goal of web services is *interoperability* among software systems regardless of their underlying frameworks, implementations, platforms, or other idiosyncrasies. This is achieved by communications in the form of XML-based messages transported over networks by HyperText Transfer Protocol (HTTP), both of which are open standards. One can easily see why web services have generated such fervor, for the prospect of integrating disparate systems seamlessly by way of non-proprietary standards is an exciting one.

As the enterprise evolves rapidly towards web services, project managers charged with producing enterprise solutions must have their understanding evolve in parallel. With J2EE and .NET the leading choices for

developing enterprise systems, this paper will now briefly discuss the extent to which they support web service development. Moreover, the purpose of this paper has been to identify the criteria by which project managers should choose between J2EE and .NET as the solution for an enterprise-level development effort. In the interest of completeness, therefore, this paper will also address any additional criteria that must be considered when choosing between J2EE and .NET as the solution for a *web service* development effort.

J2EE Web Services

It would seem that J2EE lacked foresight with regards to the zeal generated by the web services paradigm. As a result the most recent specfication lacks robust support for web services. Transport protocols provide a glaring example. Application servers do not support HTTP as a native communication protocol, so web service requests transported over HTTP must be bridged to another protcol to activate J2EE web services [11]. Ultimately, outside of API's for processing XML utilizing standard interfaces, J2EE is missing a great deal when it comes to web services, and vendors are forced to provide proprietary extensions to fill in the gaps.

The upcoming J2EE specification due for release in the fall of 2003 seeks to rectify many of these problems. The new specification, for example, provides for exposing EJB's as web services for discovery and utilization by clients [10]. Also included is more robust support for processing XML-based messages sent over HTTP [10]. However, as promising as all of this may be, it is all purely theoretical since the specification has yet to be released. The rate at which vendors produce application servers that meet the specification remains to be seen. Moreover, the manner in which IDE's automate web service development to support the new specification also is unknown. Therefore, J2EE developers must currently rely on web service development that is heavily proprietary, and at best they can only be cautiously optimistic for the future.

.NET Web Services

In stark contrast to J2EE, .NET's support for web service development is quite possibly its best feature. Indeed, .NET has from its beginnings demonstrated tremendous foresight in the web services realm. For example, Windows Server has native support for HTTP communication [11]. Furthermore, reflecting Microsoft's commitment to XML, .NET features a rich library of API's for processing XML-based messages. Finally, as one would expect from the powerful IDE, Visual Studio has a number of features to automate the development both of web services themselves and of clients for existing web services.

.NET has proven itself in industry to be an effective architecture for web service development. The aforementioned TRX Travel, for example, has utilized web services to manipulate travel data and to create a generic, reusable interface to its business logic layer [6]. Another .NET web services success story is the Central Bank of Costa Rica (CBCR), who recently ported its legacy application to .NET [12]. Among the web services built by CBCR are a service for messaging, a service for managing financial setlements, and even a service for integrating Java-based financial applications which exemplifies the very interoperability that motivated the genesis of the web service paradigm [12]. The ability to produce such a wide array of web services so quickly has helped .NET to take the lead at this stage in the evolution of web services.

Choosing Between the Architectures

Web services clearly represent a remarkable branch in the evolution of enterprise systems. Though the enthusiasm has been tempered somewhat by the realities of the marketplace, the web services paradigm shall remain a vibrant one[10]. Therefore, as with all enterprise systems, project managers must be able to engage in a legitimate process of natural selection between J2EE and .NET when deciding which will be the architecture for a web service solution.

All of the criteria and considerations discussed previously for a conventional enterprise system still apply when it comes to web services. For example, J2EE's platform independence could be the deciding factor if the platform to which a web service will be deployed is unknown or likely to change. However, there is one signifcant exception to the conclusions drawn previously. Earlier it was suggested that J2EE can quite reasonably claim to be the more proven and more robust solution for an enterprise application. Yet when it comes to web services, the opposite is true. As mentioned before, .NET surpassed J2EE by a wide margin in its appraisal of the web services landscape, and as a result .NET provided web service developers with a great deal of support. J2EE has worked quickly to narrow the gap, but the extent to which it will succeed will only become apparent over time. Thus, with .NET being so far ahead in the web services realm and generally, as discussed previously, being the better choice when time is a factor for any enterprise development effort, it would seem that for now .NET is the better choice when developing web services.

Concluding Remarks

With the enterprise paradigm having evolved from the client-server paradigm and in turn evolving in some measure towards the web services paradigm, project managers must contend with many complex issues when choosing between J2EE and .NET. What makes their task even more difficult is the rapid pace with which the architectures themselves are evolving in an effort to become more robust. This paper has attempted to articulate and clarify the criteria that project managers must consider when making their choice.

It is unclear if natural selection by the marketplace will ultimately determine a victor in the battle for the enterprise between J2EE and .NET. Rather, it is far more likely that the two will simply coexist in the ecosystem of enterprise architectures. In fact, we can make only two claims with any certainty: the rivalry will continue to make for fascinating theater, and more importantly, the true victors will be project managers and their development teams, all of whom will reap the benefits of the evolved functionality that will inevitably result from the competition.

[10] As true interoperability among systems is an extremely difficult goal to achieve, web services have come to encounter resistance in several forms, including the rapid evolution of standards (especially in the area of security of XML-based messages), the deliberate pace with which vendors adopt new standards, database concurrency issues [13], and the performance cost of XML-based messaging. The W3C must address these and other problems—and vendors must adhere to its recommendations—if web services are to continue to flourish.

Acknowledgements

The author would like to recognize the following individuals who helped to improve this discussion: Geoffrey Simpson, Mauricio Calabrese, Randa Khoury of the National Academy of Sciences, and Jonathon Leete and Sam Stange of Logistics Management Institute (LMI). The author would also like to express his heartfelt gratitude to Vice-President Dr. Susan Marquis and Program Director Robert Hutchinson of LMI, whose continuous encouragement and support were invaluable. Finally, the author wishes to express a special note of thanks to John Kupiec of LMI for his mentoring guidance, unwavering patience, and sage wisdom. This paper would not have been possible without him.

References

[1] McAllister, Neil. *New Architect*: "The Great Migration: The Rocky Road to J2EE and .NET." March 2003.

[2] Roman, Ed. TheServerSide.com: "A few tips on deciding between EJB and COM." Unknown date of publication.

[3] Lowy, Juval. *.NET Magazine*. "Set a New Course With .NET" December 2001.

[4] Sessions, Roger. Java 2 Enterprise Edition (J2EE) versus The .NET Platform: Two Visions for eBusiness. March 2001.

[5] Marinescu, Floyd. TheServerSide.com (www.theserverside.com): "The State of The J2EE Application Server Market: History, important trends and predictions." March 2001.

[6] Bustamante, Michèle Leroux. *.NET Magazine*: "TRX Travel Services Goes Live With .NET." May 2003.

[7] Hatem El-Sebaaly. UC Irvine Extension (unex.uci.edu): "J2EE vs.Microsoft.NET: Choosing an Enterprise System." August 2002.

[8] MacHale, Robert. Microsoft Developers Network (msdn.microsoft.com): "Distributed Transactions in Visual Basic .NET." February 2002.

[9] Champion, Michael; Ferris, Chris (eds.) *et al*.: *Web Services Architecture*, W3C Working Draft, November 2002.

[10] Varhol, Peter. *JavaPro Magazine*. "J2EE 1.4: A Web Services Kit." August 2003.

[11] Newcomer, Eric. *.NET Magazine*. "Decide Between J2EE and .NET Web Services." October 2002.

[12] Thé, Lee. *.NET Magazine*. "CBCR Ports Critical App to .NET." August 2003.

[13] Ambler, Scott. Lecture: "Agile Database Techniques – Data Doesn't Have to be a Four-Letter Word Anymore." August 2003.

Using Coordination Contracts for Evolving Business Rules*

Michel Wermelinger
Dep. de Informática
Univ. Nova de Lisboa
2829-516 Caparica, Portugal
mw@di.fct.unl.pt

Georgios Koutsoukos,
Hugo Lourenço, Richard Avillez,
João Gouveia, Luís Andrade†
ATX Software SA
Alameda António Sérgio, 7, 1C
2795-023 Linda-a-Velha, Portugal

José Luiz Fiadeiro
Dep. of Computer Science
Univ. of Leicester
University Road
Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract

This experience paper reports on the use of coordination contracts in a project for a credit recovery company. We have designed and implemented a framework that allows users to define several business rules according to pre-defined parameters. However, some rules require changes to the services provided by the system. For these, we use coordination contracts to intercept the calls to the underlying services and superpose whatever behaviour is imposed by the business rules applicable to that service. Such contracts can be added and deleted at run-time. Hence, our framework includes a configurator that, whenever a service is called, checks the applicable rules and configures the service with the given parameters and contracts, before proceeding with the call. Using this framework we have also devised a way to generate rule-dependent SQL code for batch-oriented services.

Based on our experience, we feel that coordination contracts facilitate the evolution of the system in order to accomodate new business rules that change the "normal" behaviour of the provided system's functionalities.

1 Introduction

This paper describes an architectural approach to system development that facilitates adaptation to change so that organisations can effectively depend on a continued service that satisfies evolving business requirements. This approach has been used in a real project in which ATX Software developed an information system for a company specialised in recovering bad credit. The approach is based on:

- the externalisation of the business rules that define the dependency of the recovery process on the financial institution and product (e.g., house mortgage) for which the debt is being recovered;
- the encapsulation of parts of behaviour into so-called coordination contracts that can be created and deleted at run-time, hence adapting computational services to the context (e.g., institution and product) in which they are called.

*Work partially supported by project AGILE (IST-2001-32747) funded by the European Commission; by project POSI/32717/00 (Formal Approach to Software Architecture) funded by Fundação para a Ciência e Tecnologia; and by the research network RELEASE (Research Links to Explore and Advance Software Evolution) funded by the European Science Foundation.

†E-mail: {hlourenco,ravillez,jgouveia,landrade}@atxsoftware.com

These two mechanisms have two different stakeholders as target. Business rules are intended for system users, who have no technical knowledge, so that they can parameterise the system in order to cope with requirements of new financial institutions and products. Coordination contracts are intended for system developers to add new behaviour without changing the original service implementation. This is made possible by the ability of coordination contracts to intercept calls to the service's methods and execute the contract's code instead.

Coordination contracts [1] are a modelling and implementation primitive that allows transparent interception of messages and as such replace the service's method by the code provided by the coordination contract. Transparent means that neither the service nor its client are aware of the existence of the coordination contract. Hence, if the system has to be evolved to handle the requirements imposed by new institutions or products, many of the changes can be achieved by parameterising the service (data changes) and by creating new coordination contracts (behaviour changes), without changing the service's nor the client's code. This was used, for instance, to replace the default calculation of the debt's interest by a different one. The user may then pick one of the available calculation formulae (i.e., coordination contracts) when defining a business rule.

To be more precise, a coordination contract is applicable to one or more objects (called the contract's participants) and has one or more coordination rules, each one indicating which method of which participant will be intercepted, under which conditions, and what actions to take in that case. In our approach to the system we developed, all coordination contracts are unary, the participant being the service affected by the business rule to which the coordination contract is associated. Moreover, each contract has a single rule. We could have joined all coordination rules that *may be* applicable to the same service into a single contract, but that would lead to less efficiency and to more complex rule conditions. The reason is that once a contract is in place, it will intercept *all* methods given in all the contract's rules, and thus the rule conditions would have to check at run-time if the rule is really applicable, or if the contract was put in place because of another coordination rule.

We should also mention that in this project we used our environment to develop Java applications using coordination contracts [3]. The environment is freely available from www.atxsoftware.com. The tool allows writing contracts, and to register Java classes (components) for coordination. The code for adapting those components and for implementing the contract semantics is generated based on a micro-architecture that uses the Proxy and Chain of Responsibility design patterns [2]. This microarchitecture handles the superposition of the coordination mechanisms over existing components in a way that is transparent to the component and contract designer. The environment also includes an animation tool, with some reconfiguration capabilities, in which the run-time behavior of contracts and their participants can be observed using sequence diagrams, thus allowing testing of the deployed application.

The structure of the paper is as follows. The next section introduces some example business rules, taken from the credit recovery domain, and shows how coordination contracts are used to change the default service functionalities according to the applicable business rules. Section 3 sketches the framework we implemented, describing how the service configuration is done at run-time according to the rules. Section 4 explains how the same framework is used to generate rule-dependent SQL code to be run in batch mode. The last section presents some concluding remarks.

2 Business Rules and Coordination Contracts

ATX Software was given the task to re-implement in Java the information system of Espírito Santo Cobranças, a debt recovery company that works for several credit institutions, like banks and leasing companies. The goal was not only to obtain a Web-based system, but also to make it more adaptable to new credit institutions or to new financial products for which the debts have to be collected. This meant that business rules should be easy to change and implement.

The first step was to make the rules explicit, which was not the case in the old system, where the conditions that govern several aspects of the debt recovery process were hardwired in tables or in the application code itself. We defined a business rule to be given by a condition, an action, and a priority. The condition is a

boolean expression over relations (greater, equal, etc.) between parameters and concrete values. The available parameters are defined by the rule type. The action part is a set of assignments of values to other parameters, also defined by the rule type. Some of the action parameters may be "calculation methods" that change the behaviour of the service to which this rule is applicable. The priority is used to allow the user to write fewer and more succint rules: instead of writing one rule for each possible combination of the condition parameter values, making sure that no two rules can be applied simultaneously, the user can write a low priority, general, "catch-all" rule and then (with higher priority) just those rules that define exceptions to the general case. As we will see later, rules are evaluated by priority order. Therefore, within each rule type, each rule has a unique priority.

To illustrate the concept of business rule, consider the agreement simulation service that computes, given a start and ending date for the agreement, and the number of payments desired by the ower, what the amount of each payment must be in order to cover the complete debt. This calculation is highly variable on a large number of factors, which can be divided into two groups. The first one includes those factors that affect how the current debt of the ower is calculated, like the interest and tax rates. This group of factors also affect all those services, besides the agreement simulation, that need to know the current debt of a given person. The second group covers factors concerned with internal policies. Since the recovery of part of the debt is better than nothing, when a debt collector is making an agreement, he might pardon part of the debt. The exact percentage (of the total debt amount) to be pardoned has an upper limit that depends on the category of the debt collector: the company's administration gives higher limits to more experienced employees.

As expected, each group corresponds to a different business rule type, and each factor is an action parameter for the corresponding rule type. The condition parameters are those that influence the values to be given for the action parameters. As a concrete example, consider the last group in the previous paragraph. The business rule type defines a condition parameter corresponding to the category of the debt collector and an action parameter corresponding to the maximum pardon percentage. A rule (i.e., an instance of the rule type) might then be `if category = ''senior'' or category = ''director'' then maxPardon = 80%`. The priorities might be used to impose a default rule that allows no pardon of the debt. The lowest priority rule would then be `if true then maxPardon = 0%`.

However, a more interesting rule type is the one corresponding to the calculation of the debt (the first group of factors for the agreement service). The debt is basically calculated as the sum of the loan instalments that the ower has failed to pay, surcharged with an amount, called "late interest". The rules for calculating this amount are defined by the credit institution, and the most common formula is: instalment amount * late interest rate * days the payment is late / 365. In other words, the institution defines a yearly late interest rate that is applied to the owed amount like any interest rate. This rate may depend only on the kind of loan (if it was for a house, a car, etc.) or it may have been defined in the particular loan contract signed between the institution and the ower. In the first case, the rate may be given as an action parameter value of the rule, in the second case it must be computed at run-time, given the person for whom the agreement is being simulated. But as said before, the formula itself is defined by the institution. For example, there are instutions that don't take the payment delay into account, i.e., the formula is just `instalment amount * late interest rate`. For the moment, these are the only two formulas the system incorporates, but the debt recovery company already told us that in the forseeable future they will have to handle financial institutions and products that have late interest rates over different periods of time, e.g., quarterly rates (which means the formula would have the constant 90 instead of 365).

In these cases, where business rules impose a specific behaviour on the underlying services, we add an action parameter with a fixed list of possible values. Each value (except the default one) corresponds to a coordination rule that contains the behaviour to be superposed on the underlying service (which implements the default behaviour, corresponding to the default value of the parameter). However, from the user's perspective, there is nothing special in this kind of parameter; the association to coordination rules is done "under the hood". For our concrete example, the late interest rule type would have as condition parameters the institution and the product type, and as action parameters the interest rate (a percentage), the rate source (if it is a general rate or if it depends on the loan contract), and the rate kind (if it is a yearly rate or a fixed one). The last two parameters are associated to coordination rules and the first parameter (the rate) is optional, because it has to

be provided only if the rate source is general. Two rule examples are

- `if institution = 'Big Bank' and productType = 'car loan' then rate = 7%, source = 'general', kind = 'fixed';`
- `if institution = 'Big Bank' and productType = 'house loan' then source = 'contract', kind = 'yearly'.`

As for the coordination rules, we need one for each computation that differs from the default behaviour, which is implemented directly in the service because it is assumed to be the case occurring most often. For the example, we need a rule to fetch the rate from the database table that holds the loan contract information for all processes handled by the debt recovery company, and another rule to calculate the late interest according to the fixed rate formula.

Continuing with our example, the service has (at least) the following methods:

- `void setRate(double percentage)`, which is used to pass the value of the `rate` action parameter to the service;
- `double getRate()`, which is used by clients of the service, and by the next method, to obtain the rate that is applicable;
- `double getInterest()`, which uses auxiliary methods implemented by the same service to calculate the late interest to be paid. Its implementation is `return getInstalment() * getRate() * getDays() / 365;`.

Given these methods, the coordination rules are as follows:

Fixed Rate This rule intercepts the `getInterest()` method unconditionally, and executes: `return getInstalment() * getRate()`.

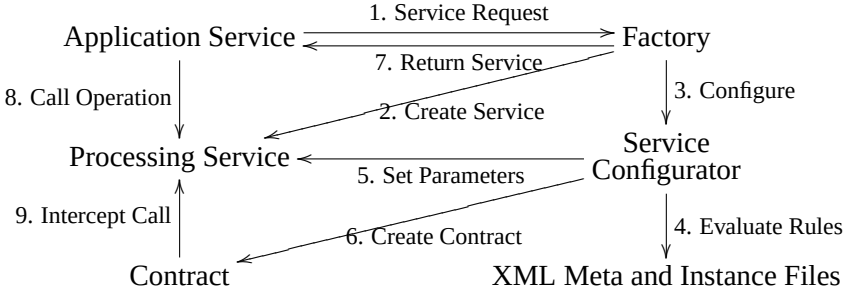
Contracted Rate This rule intercepts the `getRate()` method under the condition `!calculated`, and executes: `r =` the rate obtained by consulting the database`; setRate(r); calculated = true`.

The second rule requires the coordination contract to have a local boolean attribute `calculated`, initialized to false. The idea is that, no matter how often the service's clients call the `getRate()` method, the database lookup will be done only for the first call, and the rate is stored into the service object, as if it were given directly by a business rule.

The next section explains how the three parts (business rules, coordinations contracts, and services) work together at run-time in order to ensure that the correct (business and coordination) rules are applied at the right time to the right services.

3 Architectural Framework

The architecture of the configuration framework, and the steps that are taken at run-time, are shown next.



The process starts with the creation of an application service object to handle the user's request, e.g., the request for the simulation of the agreement. This object contains the necessary data, obtained from the data given by the user on the web page, and will call auxiliary processing services. Each service is implemented by a class, whose objects will be created through a factory (step 1 in the figure). After creating the particular instance of the processing service (step 2), the factory may call the service configurator (step 3), if the service is known to be possibly subject to business rules. The configurator consults two XML files containing information about the existing business rules. The one we called meta file defines the rule types (see Fig. 1 on page 6 for an example), while the instance file contains the actual rules (see Fig. 2 on page 7). The configurator first looks into the meta file to check which business rules are applicable for the given processing service. For each such rule, the meta file defines a mapping from each of the rule type's condition (resp. action) parameters into getter (resp. setter) methods of the service, in order to obtain from (resp. pass to) the service the values to be used in the evaluation of the conditions of the rules (resp. the values given by the action part of the rules). There is also the possibility that an action parameter is mapped to a coordination contract. With this information (which of course is read from the meta file only once, and not every time the configurator is called), the configurator calls the necessary getters of the service in order to obtain the concrete values for all the relevant condition parameters. Now the configurator is able to evaluate the rules in the instance file (step 4), from the highest to the lowest priority one, evaluating the boolean expression in the if part of each rule until one of them is true. If the parameter values obtained from the service satisfy no rules' condition, then the configurator raises an exception. If a suitable rule is found, the configurator reads the values of the action parameters and passes them to the service (step 5) by calling the respective setters. If the action parameter is associated to a coordination contract, the configurator creates an instance of that contract (step 6), passing to the contract constructor the processing service object as the participant. At this point the configurator returns control to the factory, which in turn returns to the application service a handler to the created (and configured) processing service. The application service may now start calling the methods of the processing service (step 8). If the behaviour of such a method was changed by a business rule, the corresponding contract instance will intercept the call and execute the different behaviour (step 9).

Of course, the application service is completely unaware that the processing service has been configured and that the default behaviour has changed, because the application just calls directly the methods provided by the processing service to its clients. In fact, we follow the strict separation between computation and configuration described in [4]: each processing service has two interfaces, one listing the operations available to clients, the other listing the operations available to the configurator (like the getters and setters of business rule parameters). The application service only knows the former interface, because that is the one returned by the factory. This prevents the application service from changing the configuration enforced by the business rules.

The user may edit the XML instance file through a tool we built for that purpose to browse,edit and create business rules. The tool completely hides the XML syntax away from the user, allowing the manipulation of rules in a user-friendly manner. Furthermore, it imposes all the necessary constraints to make sure that, on the one hand, all data is consistent with the business rules metadata (i.e., the rule types defined in the XML meta file), and, on the other hand, that a well-defined XML instance file is produced. In particular, the tool supplies the user with the possible domain values for required user input, it checks whether mandatory action parameters have been assigned a value, facilitates the change of priorities among rules and guarantees the uniqueness of priorities, allows to search all rules for a given institution, etc. You may notice from the presented XML extracts that every rule type, rule, and parameter has a unique identifier and a name. The identifier is used internally by the configurator to establish cross-references between the instance and the meta file, while the name is shown by the rule editing tool to the user.The `valueType` attribute of a parameter is used by the rule editor to present to the user (in a drop-down list) all the possible values for that parameter.

Notice that the user is (and must be) completely unaware of which services are subject to which rule types, because that is not part of the problem domain. The mapping between the rules and the service classes they affect is part of the solution domain, and as such defined in the XML meta file. As such, each rule type has a conceptual unity that makes sense from the business point of view, without taking the underlying services implementation into account.

```
<service class="ComputeDebt">
  <ruleType name="Late Interest" id="LateInterest">
    <condition>
      <conditionGroup>
        <conditionParameter name="Financial Institution"
                            id="Inst" type="string">
          <valueType name="Institution" />
          <getter name="getInstitutionCd" returnType="String" />
          <SQL>
            <expr>AT_LATE_INTEREST_CALC.INSTITUTION_CD</expr>
            <from>AT_LATE_INTEREST_CALC</from>
          </SQL>
        </conditionParameter>
        <conditionParameter name="Credit Type"
                            id="CredType" type="string">
          <valueType name="CreditType" />
          <getter name="getCreditType" returnType="String" />
          <SQL>
            <expr>ST_PROCESS_CONTRACT.CREDIT_TYPE_CD</expr>
            <from>ST_PROCESS_CONTRACT,AT_LATE_INTEREST_CALC</from>
            <join>ST_PROCESS_CONTRACT.PROCESS_NBR =
                  AT_LATE_INTEREST_CALC.PROCESS_NBR</join>
          </SQL>
        </conditionParameter>
        <!-- the current phase of the recovery process -->
        <conditionParameter name="Phase" id="Phase" type="string">
          <valueType name="ProcPhase" />
          <getter name="getProcessPhase" returnType="String" />
          <SQL>
            <expr>AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD</expr>
            <from>AT_LATE_INTEREST_CALC</from>
          </SQL>
        </conditionParameter>
        <!-- other condition parameters -->
      </conditionGroup>
    </condition>
    <!-- the action parameters would be given here -->
 </ruleType>
</service>
```

Figure 1: An extract of the XML meta file

```
<service class = "ComputeDebt" name = "ComputeDebt">
  <ruleType id = "LateInterest">
    <!-- other rules with higher priority -->

    <rule name = "Big Bank, judicial phases" id = "3" priority = "3">
      <conditionset type = "AND">
        <comparison id = "Inst" serviceValue = "0916"
           userValue = "Big Bank" operator = "equal"/>
        <conditionset type = "OR">
          <comparison  id = "Phase" serviceValue = "0005"
            userValue = "External judicial phase" operator = "equal"/>
          <comparison  id = "Phase" serviceValue = "0007"
            userValue = "Internal judicial phase" operator = "equal"/>
        </conditionset>
      </conditionset>
      <!-- the values for the action parameters come here -->
    </rule>

    <!-- remaining rules, with less priority -->
  </ruleType>
</service>
```

Figure 2: An extract of the XML instance file

4 Batch-oriented Rule Processing

The approach presented in the previous section is intended for the interactive, web-based application services that are called on request by the user with the necessary data. These data are passed along to a processing service. The configurator queries the processing service for the data in order to evaluate the conditions of the rules.

However, like most information systems, the debt recovery system also has a substantial part working in batch. For example, the calculation of the debt is not only needed on demand to project the future debt for the simulation agreement service, it is also run every night to update the current debt of all the current credit recovery processes registered in the system. In this case, the debt calculation is performed by stored procedures in the database, written in SQL and with the business rules hard-wired.

Hence, when we have a large set of objects (e.g., credit recovery processes) for which we want to invoke the same processing service (e.g., debt calculation), it is not very efficient to apply the service to each of these objects individually. It is better to apply a “batch” strategy, reversing the configuration operation: instead of starting with an object and then choosing the rule that it satisfies, we take a rule and then select all the objects that satisfy it. This is much more efficient because we may use the same configured processing service instance for objects A and B if we are sure that for both A and B the same rule is chosen.

We thus have the need to be able to determine for a given rule the set of objects that satisfy it. Pragmatically speaking, we need a way of transforming the if-part of a rule into an SQL condition that can be used in a SELECT query to obtain those objects. Therefore we extended the rule type information in the XML meta file, adding for each condition parameter the following information:

- an SQL expression that can be used to obtain the parameter value;
- the list of tables that must be queried to obtain the parameter value;
- a join condition between those tables.

Fig. 1 on page 6 shows a fragment of the meta information for the debt calculation service. There we see, for example, that in order to obtain the value of the product type parameter we have to write the following query:

```
SELECT ST_PROCESS_CONTRACT.CREDIT_TYPE_CD
FROM ST_PROCESS_CONTRACT,AT_LATE_INTEREST_CALC
WHERE ST_PROCESS_CONTRACT.PROCESS_NBR = AT_LATE_INTEREST_CALC.PROCESS_NBR
```

Using this information we can now take a rule condition and transform it into a SQL fragment. As an example, consider the rule condition (for the same service) in Fig. 2 on page 7: it is applicable to all recovery processes of "Big Bank" that are in the internal judicial phase (i.e., the company's lawyers are dealing with the process) or the external one (i.e., the case has gone to court). We may compose the information for each of the rule parameters in order to obtain a single SQL fragment for the rule condition. This fragment contains the following information:

- the list of tables that must be queried in order to evaluate the rule condition;
- an SQL condition that expresses both the join conditions between the several tables and the rule condition itself.

For our example, the meta file specifies that `Inst` and `Phase`, the two parameters occurring in the condition, only require the table `AT_LATE_INTEREST_CALC` to be queried. As for the rule condition, the meta file specifies that `AT_LATE_INTEREST_CALC.INSTITUTION_CD` corresponds to the usage of the `Inst` parameter, and `AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD` to the `Phase` condition parameter. By a straightforward replacement of these names in the boolean expression of the rule condition, we get the following SQL expression: `((AT_LATE_INTEREST_CALC.INSTITUTION_CD = '0916') AND ((AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0005') OR (AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0007')))`

The SQL representation of a rule is conveyed by an instance of class `SQLRule`, which is contained in the service configurator, because the XML files are accessed by the latter.

```
public class ServiceConfigurator {
  public class SQLRule  {
      public String getId() { ... }
      public String getName() { ... }
      public String getWhere() { ... }
      public String getFrom() { ... }
  }
}
```

Each processing service class provides a static method for obtaining all of its rules in this "SQL format". This method simply calls a method of the service configurator, passing the service identification, which returns all rules for that service in decreasing order of priority. In the example below we show how we can generate a specialized query for a rule. In this example we first obtain all the service rules in "SQL format" and then generate a query that returns the first object that satisfies the condition of the third rule.

```
ServiceConfigurator.SQLRule[] SQLrules = ComputeDebt.getSQLRules();

ServiceConfigurator.SQLRule rule = SQLrules[2];
System.out.println("Rule : " + rule.getId() + " - " + rule.getName());
String sql = "SELECT TOP 1 AT_LATE_INTEREST_CALC.PROCESS_NBR " +
  " FROM " + rule.getFrom() +
  " WHERE " + rule.getWhere() +
  " AND PROCESSED = false";
System.out.println(sql);
```

The output generated is the following:

```
Rule : 3 - Big Bank, judicial phases

  SELECT TOP 1 AT_LATE_INTEREST_CALC.PROCESS_NBR
  FROM AT_LATE_INTEREST_CALC
  WHERE ((AT_LATE_INTEREST_CALC.INSTITUTION_CD = '0916')
  AND ((AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0005')
  OR (AT_LATE_INTEREST_CALC.ACTUAL_PHASE_CD = '0007')))
  AND PROCESSED = false
```

Similar queries are generated for each rule and each query is executed. In this way we obtain, for each rule, one object satisfying its condition. This object is basically a representative of the equivalence class of all objects that satisfy the given rule conditions. Now, step 1 of the run-time configuration (section 3) is executed for each object. In other words, we execute the same call as if it were an application service, but passing one of the already created objects as data. The steps then proceed as usual. This means that after step 7, we obtain a service instance that has been configured according to the rule corresponding to the given object.

The generation of the SQL code for the batch version of a service proceeds as follows, for each i from 1 to n (where n is the number of rules for that service). First, generate the SQL query to select all objects satisfying the condition of the i-th rule. This is done as shown above, but without the `TOP 1` qualifier. Second, call a special method on the i-th service instance. This method will generate the SQL code that implements the service, based on a template that is customized with the action parameters that have been set by the configurator on the service.

In summary, the SQL code that will be executed in batch is made up of n "modules", one for each rule. Each module first selects all objects to which the rule is applicable, and then executes the parameterized SQL code that corresponds to the Java code for the interactive version of the service. The modules are run sequentially, according to the prioritization of the rules.

This raises a problem. If some object satisfies the conditions of two or more rules, only the one with the highest priority should be applied to that object. To preserve this semantics, each batch service uses an auxiliary table, initialized with all objects to be processed by the service; in the case of the debt calculation service, it is the `AT_LATE_INTEREST_CALC` table. This table has a boolean column called `processed`, initialized to false. As each batch module executes, it marks each object it operates on as being processed. Hence, when the next module starts, its query will only select objects that haven't been processed yet. In this way, no two rules will be applied to the same object.

The last, but not least, point to mention are coordination contracts. As said above, one service instance has been created for each rule, and configured accordingly. This means that coordination contracts may have been superposed on some service instances (step 6). Hence, the SQL code generated from those service instances cannot be the same as for those that haven't any coordination contracts. The problem is that services are unaware of the existence of contracts. The result is that when the code generation method of a service object is called (step 8), the service object has no way to know that it should generate slightly different code, to take the contract's behaviour into account. In fact, it *must* not know, because that would defeat the whole purpose of coordination contracts: the different behaviours would be hard-wired into the service, restricting the adaptability and flexibility needed for the evolution of the business rules. Since the Java code (for the web-based part of the system) and the SQL code (for the batch part) should be in the same "location", to facilitate the maintenance of the system, the solution is of course for each contract to also generate the part of the code corresponding to the new behaviour it imposes on the underlying service. For this to be possible, the trick is to make the code generation method of the service also subject to coordination. In other words, when a contract is applied to a service object, it will not only intercept the methods supplied by the service to its clients, it will also intercept the code generation method (step 9) in order to adapt it to the new intended behaviour.

5 Concluding Remarks

This paper reports on the first industrial application of coordination contracts, a mechanism we have developed for non-intrusive dynamic coordination among components, where "dynamic" means that the coordination may change during execution of the system.

One of the key requirements of the debt recovery system ordered to ATX Software was the flexibility of adaptation to new client institutions and new financial products. This flexibility was achieved by two means. The first is the definition of parameterised business rule types. The condition parameters can be combined in arbitrary boolean expressions to provide expressivity, and priorities among rules of the same type allow to distinguish between general vs. exceptional cases. The second means are coordination contracts to encapsulate the behaviour that deviates from the default case. At run-time, from the actual data passed to the invoked service, a configurator component retrieves the applicable rules (at most one of each rule type), parameterises the service according to the rules, and creates the necessary contract instances. The contracts will intercept some of the service's functionalities and replace it by the new behaviour associated to the corresponding business rule.

The architectural framework we designed can be used both for interactive as well as batch application services. The difference lies in the fact that the batch application service has to get one representative data object for each rule, and only then can it create one processing service for each such data. The application service then asks each of the obtained configured services to generate the corresponding SQL code. Coordination contracts will also intercept these calls, in order to generate code that corresponds to the execution of the contract in the interactive case.

This approach has proved to work well for the system at hand. On the one hand it guarantees that the system will automatically (i.e., without programmer intervention) behave consistently with any change to the business rules. On the other hand, it makes possible to incorporate some changes to existing rule types and create new rule types with little effort, because coordination contracts can be added in an incremental way without changing the client nor the service code. Furthermore, the code of the services remains simple in the sense that it does not have to entangle all the possible parameter combinations and behaviour variations.

The main difficulty lies in the analysis and design of the services and the rules. From the requirements, we have to analyse which rules make sense and define what their variability points (the parameters) are. As for the services, their functionality has to be decomposed into many atomic methods because coordination rules "hook" into existing methods of the contract's participants. As such, having just a few, monolithic methods would decrease the flexibility for future evolution of the system, and would require the coordination rule to duplicate most of the method code except for a few changes.

The approach is also practical from the efficiency point of view. The overhead imposed by the configurator's operations (finding the rules, passing action parameter values, and creating coordination contract objects) does not have a major impact into the overall execution time of the application and processing services. This is both true for the interactive and batch parts of the system. In the former case, the user does not notice any delay in the system's reply, in the latter case, the time of generating the SQL procedures is negligible compared to the time they will execute over the hundreds of thousands of records in the database. Moreover, the execution time of the generated SQL code is comparable to the original batch code, that had all rules hard-wired.

To sum up, even though we used coordination contracts in a narrow sense, namely only as dynamic and transparent message filters on services, and not for coordination among different services, we are convinced that they facilitate the evolution of a system that has to be adapted to changing business rules.

6 Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] L. Andrade, J. L. Fiadeiro, J. Gouveia, and G. Koutsoukos. Separating computation, coordination and configuration. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):353–369, 2002.

[2] J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS 38*, pages 184–196. IEEE Computer Society Press, 2001.

[3] J. Gouveia, G. Koutsoukos, M. Wermelinger, L. Andrade, and J. L. Fiadeiro. The coordination development environment. In *Proc. of the 5th Intl. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 323–326. Springer-Verlag, 2002.

[4] M. Wermelinger, G. Koutsoukos, J. Fiadeiro, L. Andrade, and J. Gouveia. Evolving and using coordinated systems. In *Proc. of the 5th Intl. Workshop on Principles of Software Evolution*, pages 43–46. ACM, 2002.

Toward a Taxonomy of Clones in Source Code: A Case Study

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

Abstract

Code cloning — that is, the gratuitous duplication of source code within a software system — is an endemic problem in large, industrial systems [9, 7]. While there has been much research into techniques for clone detection and analysis, there has been relatively little empirical study on characterizing how, where, and why clones occur in industrial software systems. In this paper, we present a preliminary categorization scheme for code clones, and we discuss how we have applied this taxonomy in a case study performed on the file system subsystem of the Linux operating system. Our case study yielded several interesting results, including that cloning is rampant both within particular file system implementations and across different ones, and that as many as 13% of the 4407 functions that are more than six lines long were involved in a clone-pair relationship.

1 Introduction

Code duplication, or code cloning, is generally believed to be common in large industrial systems [9, 17, 20, 18, 15, 2, 7]. Various problems are associated with code duplication, including increased code size and increased maintenance costs. While clone detection is an area of active research, and several tools exist to facilitate code clone detection, there has been relatively little empirical research on the types of clones that are found, or where they are found.

A code clone pair is a pair of source code segments that are structurally or syntactically similar. One of the segments is usually a copy of the other, perhaps with minor changes. Code cloning occurs when developers create two identical or similar code artifacts inside a software system. For example, developers may copy and paste code. Several methods exist for detecting code clones in software, such as simple string matching [9], using statistical fingerprints of code segments [12], function metrics matching [17, 20, 18], parameterized string matching [2, 15], and program graph comparison [7]. Problems related to clone cloning will be discussed in Section 2.

In the following case study, we begin to profile the code cloning activity within a large software system that is in widespread use in industry, the Linux operating system kernel. In doing so, we hope to gain more insight into how and why developers duplicate code, in an effort to aid the development of code clone detection techniques and code clone elimination strategies. We categorize different types of cloning activity using attributes such as location and size based on manual inspection of code clones found in the system. We then provide empirical analysis of these categories, and validation on our results using two different clone detection techniques. In this study we produce a taxonomy of code cloning which will help others examine code cloning, and we present a case study of a real software system.

The rest of the paper is structured as follows: in Section 2, we describe code cloning in more detail, as well as our study subject. In Section 3, we describe the tools we used and the methodology of our study. In Section 4, we describe the code clone categories we observed in the Linux file-system. In Section 5, we describe the empirical results we obtained. Section 6 describes related work, and Section 7 summarizes our work and indicate some future research.

2 Background

In this section, we provide background on code cloning as a problem in large software systems. We give examples of reasons why code cloning occurs, as well as several examples of problems caused by code cloning.

In addition, we give an overview of our candidate software system for this case study, the Linux kernel file-system subsystem. We will provide a brief description of the Linux file-system subsystem, as well as give reasons for choosing the file-system subsystem for our case study.

2.1 Code Cloning

Code cloning is considered a serious problem in industrial software [9, 12, 13, 8, 1, 17, 20, 18, 2, 15, 7]. It is suspected that 5 to 10% of many large systems is duplicated

code [9, 3], and it has been documented to exist at rates of over 50% in a particular COBOL system [9]. Code cloning occurs for a variety of reasons[12, 20, 18, 15, 2, 7]: the short term cost of forming the proper abstractions may outweigh the cost of duplicating code; this occurs when the developer is aware of the existence of code that already performs functionality similar to, or the same as, the functionality required. Developers may duplicate code because they are under time constraints; these constraints may be imposed by deadlines, or by LOC performance evaluation. Another likely and reasonable circumstance where developers duplicate code is they do not fully understand the problem, or the solution, but they are aware of code that can do some or all of the required functionality.

Several problems can develop as a result of code copying. The size of the source code, and ultimately the size of the object code, may become significantly larger as a result of excessive code cloning[2, 12]. Cloning code can lead to unused, or "dead", code in the system, which can cause problems with code comprehensibility, readability, and maintainability [12]. Duplication of code may also introduce improperly initialized variables, which may lead to unpredictable behavior of a system, especially if a two clone segments share a common variable. Cloning may be an indication of poor design [12]. Code duplication may indicate design problems such as improper or missing inheritance, or insufficient procedural abstraction[7]. Copying code may also result in copying bugs within the code as well.These effects contribute to "software aging" [12]; over time the program becomes hard to change and possibly less reliable and more inefficient.

2.2 Case Study Subject: Linux File System

Linux is a Unix-like operating system, written by Linus Torvalds with assistance from a distributed team of programmers across the Internet. Linux aims towards POSIX and Single UNIX Specification compliance. The version of the Linux kernel we used for this study was 2.4.19, the most recent stable version at the time of the writing.

We chose the Linux File System as the study subject for our project because we hypothesized that many of the supported file systems would contain clones among them due to the similarity of their basic functionality. In addition, we know in advance that several components of the file subsystem that were created with heavy influence from existing file system types, namely `ext2`/`ext3` and `autofs`/`autofs4`.

The Linux file system subsystem is organized as a layered design, with the upper most layer being the Virtual File System (VFS). The VFS provides a standard interface for the operating system to use when interacting with various file systems types. The underlying file system types, such as `ext2` and `intermezzo`, provide function pointers for the VFS to use when interacting with the file system.

Because the various file systems must interact with, or provide service to, the same upper layer, and are providing similar functionality, we expected to see at least some cloning between file systems. After a preliminary inspection we expected to see a lot of cloning between `ext2` and `ext3`; `jffs` and `jffs2`; `fat` and `msdos` and `umsdos` and `vfat`; `autofs` and `autofs4`. These systems were either closely related in functionality or were known to have evolved directly from the same code base.

The Linux file system subsystem consists of the VFS infrastructure plus 42 file system implementations: `adfs, affs, autofs, autofs4, bfs, coda, cramfs, devfs, devpts, efs, ext2, ext3, fat, freevxfs, hfs, hpfs, inflate_fs, intermezzo, isofs, jbd, jffs, jffs2, lockd, minix, msdos, ncpfs, nfs, nfsd, nls, ntfs, openpromfs, partitions, proc, qnx4, ramfs, reiserfs, romfs, smbfs, sysv , udf, ufs, umsdos, vfat`. There are a total of 538 `.c` and `.h` files, and 279,118 lines of code (including comments and blank lines).

3 Study Methods

In this section, we describe the two methods we used to gather code clone information from the system. First, we describe parameterized string matching, as implemented by the tool CCFinder. Second, we describe our approach to metrics-based clone detection, for which we used Understand for C/C++ to obtain the raw metric information, as well as a set of Python scripts that we created to perform the clone analysis. Finally, we describe our methodology for performing categorization and analysis.

3.1 Clone Detection

In this study we have primarily used the tool CCFinder, developed by Toshihiro Kamiya et al [15]. The tool uses a parameterized matching algorithm to search for code clones within C/C++, Java, and COBOL files. This type of clone detection is good at finding clones with name substitution and line structure changes; the former can cause problems for line by line matching algorithms. Baker introduced a similar algorithm in [2].

The tool CCFinder begins by performing a lexical analysis of the source code, resulting in the creation of a list of tokens as part of the syntax of the given programming language. The tokens of all the files are concatenated into a single string. As part of the code transformation, all white space is removed from the string and comments as well.

Next, several language specific transformation rules are applied. Then type, variable, and constant identifiers are replaced by a special identifier (such as $P).

Once the source code has been transformed into this abstract token stream, an exact match algorithm is performed to find maximal matching strings within the transformed code. This is done by constructing a suffix tree and locating matching substrings within the tree, as proposed by Baker [2, 3]

After the exact matches have been found, parameter matching is performed. That is, starting from the beginning of a pair of exactly matched transformed strings, CCFinder begins parameter matching of the parameters on each line. As the parameters are matched, if a conflict is found but a sufficiently large number of lines have been matched, the clone is reported, and parameter matching begins again after the line creating the conflict.

Once the clone detection phase is complete, the detected clones are mapped back onto their source files. Then, this information is used as input in the GeminiE user interface, where clone classes are generated and the results of the clone detection are presented. These clone classes are generated based on the fact that the clone relation is an equivalence relation [15, 14]. The clone relation exists when two code segments match according to parametric matching. A clone class is the equivalence class of the clone pair relation, i.e., it is the maximal set of clones for which the clone relation holds [15].

The results of the clone detection process are presented in several ways in a graphical user interface [22]. The interface provides a scatter plot showing the user the matches between files, highlighted source code, and clone class metrics. Users can browse the detected clones pair by pair or by clone class. For a small number of files, the scatter plot can provide useful information, but when a large number of files is present with many lines, i.e., 200,000 or more, significant clones become difficult to detect through visual inspection of the scatter plot. For this study, we found that we made the most use of the tool by browsing the clone pairs individually, and by browsing the clones classes.

Before using the clone pairs extracted by this tool, we filtered out many of the clones we felt were meaningless, to improve the accuracy and relevancy of our results. Meaningless clones are segments or code that match but are not necessarily cloned code, or clones that were of no importance if they are duplicated code. For example, the inner block of structure definitions and lists of function declarations would we often considered meaningless clones. These clones were often unrelated and only appeared because of the detection process in parameterized string matching. After the initial extraction of clone pairs, we were presented with 5000 clone pairs, and 1809 clone classes. We deleted 1996 clones in an effort to remove at least a significant number of meaningless clone pairs. This left us with 1604 clone classes, a decrease of only 200 clone classes. We do not claim to have removed all of the meaningless clones, but we believe that we have removed a significant number of them.

3.2 Metrics-Based Clone Detection

Metrics-based clone detection methods use groups of metrics to generate "fingerprints" for each function in the source code. These metrics are often gathered using both the program source, as in the case of number of lines of comments, and from an Abstract Syntax Tree, as in the case of cyclomatic complexity. Metrics-based clone detection was introduced by Mayrand et al. in [20] and Kontogiannis et al. in [18]. Further studies using function metrics as a basis of clone detection include [4, 6, 5, 8, 1, 17, 21].

In our case study we used the following set of metrics:

1. Line counts: total number of lines, count lines blank, count of lines of code, count lines of declarations, count lines of executable code, count lines of comment.
2. Count number of parameters, number of global variables used.
3. Count number of parameters or global variables modified.
4. Cyclomatic complexity.
5. Maximum level of nesting.

These metrics are different than those used in other studies such as [20, 18] but as stated in [1], in large systems the choice of metrics does not significantly affect the results. We have used a subset of those metrics used in previous work [20, 18]. From this we would expect that our returned pairs be less precise, and more false positives to be present, but this is not the case. Upon visual inspection of several hundred of the clone pairs, false matches were very rare, confirming that the choice of metrics does not affect the results.

As in [1, 8] we searched for functions that had identical metric fingerprints. This corresponds to ExactCopy and DistrinctName classes which were defined in [20]. As in [1, 8], we did not use function name as a parameter.

To perform function matching based on metrics, we gathered our metrics using the tool Understand for C/C++. We then wrote a small program that performed the function matching grouping functions together one metric at a time. Function comparisons based on metric fingerprints can be done in $\mathcal{O}(n * m)$ time where n is the number of functions and m is the number of metrics.

3.3 Classifying and Evaluating Clones

To classify the clone pairs, we used the results from the clone detection using CCFinder. Because of the large volume of information presented to us, caused by the large wealth of information given to CCFinder, it was difficult to see any interesting trends that might occur amongst related files. This is because the clones were distributed among many files and many of the clone pairs appeared as blocks of code and it was difficult to get a feel for the cloning activity as a whole within the file system. To remedy this, we researched the file systems included with the Linux kernel to evaluate the relationships between them, and to pinpoint places where cloning activity is likely to have occurred. We also found some interesting relationships between several file systems, which we did not expect, by graphically displaying the amount of clone-pairs occurring between each file-system.

After narrowing down where we would begin to look, we manually viewed a large percentage of the clone pairs found in that area of the system. As we saw trends, we identified types of clones and began classifying many of the clone pairs that fell into these various categories. Once we had created many of the clone categories we have now, we browsed clones within the entire file system to find if there were clone categories we had not yet seen.

When we had a set of clone categories that we were satisfied with, we wrote scripts to place the clone pairs into the categories we had created. The criteria we based these scripts on were as follows: for functions to be classed as clones, 60% of their code must be common between the two. Initialization clones must start within the first 5 lines of a function and end within the first half of the code. Finalization clones must start in the last half of the function and end in the last 5 lines. Blocks of code not in the same function must not be in any of the above clone types. All clone types are exclusive, so a clone pair that is part of a cloned function relationship can not also be an initialization clone.

After categorization, and for any other empirical results we have presented, we performed manual inspection of a large percentage of the clone pairs in the given experiment to ensure that they were within the criteria that we specified and that they were accurately found as clones.

3.4 A Basis for Comparison

As a way to compare the results given to us from the two methods of clone detection, we manipulated the data extracted from one to be close in form to the other. Because full function matches were a smaller subset of CCFinder's returned clone pairs, we used only the function matches found by CCFinder to compare to the function matches found by metric based detection.

In doing so, we defined a criterion for which to decide when a function was matched with another based on the code segments matched between the two. For two functions to match, more than 60% of their individual code must be common between the two. This may be in the form of a single segment of code duplicated between the two, or several individual code segments.

Another issue to consider in comparing the two methods was the minimum size of code segments that could be classed as a clone pair. We found that five lines often found function matches that were too small for CCFinder to find, because we had set CCFinder to find code segments of a minimum size of 30 tokens. Several values of minimum line numbers were tried, five, six, and seven and the results of these are described in Section 5.3.

4 A Taxonomy of Clones

In the following subsections we present a taxonomy of the types of clones we found during this case study using the clone pairs from CCFinder; in the following section we analyze our findings.

The categories of clones are described using the following template: the first paragraph describes the structure of the clone; the second paragraph describes problems caused by this type of clone; the third paragraph describes reasons why these clones may be introduced into the software; and the fourth paragraph describes a possible solution to that form of cloning activity.

4.1 Duplicated blocks within same function

Characterized as repeated blocks of code within the same function, these blocks are of non-trivial size (such as 5 to 127 lines of code) and each copy expresses the same semantic idea, generally with very few variables changed (often only one). We found that this type of clone occurs often in the Linux file-system subsystem.

The major problem that this can cause is increased code size; in particular it can cause functions to grow long and unreadable. In addition, this type of cloning may lead to unintended diverging evolution of the code blocks if a developer changes one block, and not another. A bad initialization or 'value changed' type of error can very easily happen in this type of code, because it is likely variables are shared, intentionally or unintentionally, by the individual blocks.

Situations where this typically occurred was in control structures such as `switch` and `if/else` statements. The cause of this may be that some developers do not anticipate a condition that may require a similar block, so they

do not think to make the block a function from the start. Also, making the function that encapsulates the functionality of this clone block may appear to be too much work, because of the number of local variables involved in the code block. Another reason may be time: it is very fast to just copy and paste the block just a few lines down, and the developer "knows" the code works, so it is a quick and dirty solution. Performance may also be an issue, if many local variables are required to be passed, stack creation and destruction may be time consuming.

A solution to this problem, as with many code clones, would be to create a new function or macro to represent the block, and call the function where these clones occur. Parameters to the function would be the few changed variables that occur in the code block. One would expect this change to be simple and straightforward to implement.

4.2 Similar functions, same file

This type of clone occurs when a programmer has two functions performing very similar tasks, with minor variations. These types of clones are often characterized by changing only a few function calls, variable initializations, constants, or other minor things. We consider any functions which both match 60% of their code to be cloned functions.

Consequences of this type of clone are increased code size. Also fixing bugs may be harder because same error may be spread across several functions, as well as the functions may evolve on separate paths as various maintainers update them.

Developers are likely to do this when the effort required to parameterize the code block and create a more general function appears to be too great when compared to simply copying the code. Also cloning the function may actually make the program conceptually simpler, because the function names can be specific and meaningful. This type of cloning we do not consider extremely harmful because clones are not physically far apart, but it is recommended that such cloning activity should be documented as it may not be apparent to future maintainers which functions are clones of each other.

Solutions for this can be very simple, or quite complex. Possible solutions would be to introduce function pointers to the parameter list, adding more parameters for initialization, etc.

4.3 Functions cloned between files within the same directory

This type of clone occurs when the same functionality is required among multiple files. The majority of code duplication that occurs within a directory (excluding duplication with the same file) is related to duplicated functions, more than 80% of clone pairs that occurred within the same directory (but not in the same file) were related to the duplication of functions. It often occurs with no changes at all to the cloned segment of code, or minor changes such as the function name and some variable or function calls. At times, several constants may be changed, global variables accessed and in these cases a solution is harder to find.

Consequences of this type of clone are code size increase, and increased difficulty in error finding and correcting. The copied code segments are no longer localized in the same file and easily identified, but may be scattered across as many as four or five files. At times, this type of code duplication may contribute to source code that is easier to read. Functions will be easier to understand because they will not include extra logic and flows of control which would be required to restructure a function to encompass the more general functionality required of it to eliminate duplicates. This case is less frequent however, and quite often the use of function pointers or some minor conditional operations would create a function which may perform the desired task.

A simple solution to this is to create a common file to use as a library, and migrate the function definitions and prototypes of the cloned functions to this file. This will work best in the case of exact copies, or clones with minor changes.

4.4 Functions cloned across directories

This type of clone may occur when the same functionality is common among several different components in the software. As with functions cloned within a subsystem, it may entail no changes at all to the cloned segment, or minor changes such as the function name and some variable or function calls. We often saw this type of clone for generic kinds of tasks such as parsing options or outputting errors.

Consequences of this type of clone are code size increase, and may increase labor for error fixing. Also, it may be the case that one developer created one component, and is unaware of the clones existing in the rest of the system. In this case, when an error is found, repairs may not even have a chance to be propagated to the rest of the clones.

This type of cloning may occur when a new subsystem is being created, and the design and implementation is based on previous work of another subsystem.

Creating a set of library function may be the easiest solution, but if the function is cloned only between several files, the effort put into creating a new library, and maintaining it, to be shared by all components may be more work than it is worth.

4.5 Cloned files (possibly with some changes)

This type of clone occurs when a new problem arises with requirements that are very similar to those of an existing software system, and the source code is readily available. For example, when new file system is introduced to the system, it may be possible to copy another's file, and make only minor changes. We saw a very good example of this when we compared `ext2` and `ext3`, in particular `buffer.c` in both systems. This is a very rare occurrence from what we have seen in the file-system subsystem, but in other systems such as this SCSI subsystem this type of cloning activity seems to be much more frequent [10].

Consequences of this type of cloning can be much more severe than function cloning, because the clone has now introduced a large number of lines of code that are common between the two files, and must be changed together, especially when bug fixing. Because it is likely that there will be some alterations to some of the code, it may not be clear where or how to change the cloned file when reflecting changes that have been made to the original code. Also, this is one of the worst-case scenarios for code size increase. In addition, it is possible that side effects (such as inefficient device usage and settings) can occur if the developer does not fully understand the code that he/she has copied. This may lead to inefficiencies in the code and instability. This type of cloning will occur when speed of development may be a factor, or a developer may not completely understand the problem at hand. We have also seen this when drivers are made for related hardware, although not part of this study.

Solutions to this problem may not be as simple as other cloning types. Because the two files are used on different products or include different features, they may need evolve separately from this point on. As well, changes that have been made to the duplicated code may make it difficult to re factor both subsystems completely just to remove to code duplicates. That said, a workable solution may be to try to take the common invariant code and place it into a common library file which both subsystems could use. This solution may lead to a slightly more complex architecture.

4.6 Blocks across files

This type of cloning is similar to the first one but it occurs in different files within the same directories or across file systems. Often, in the case of cloning blocks across directories, we see that the cloned block is in fact the remains of what appears to be a cloned function. The function is often changed to suit the developers own personal style and also to meet the specific needs of his/her own project. Based on our observations, we would argue that most clones that occur across files start out as whole function clones and then are manipulated to fit the current project goals until what remains are scattered blocks of code which can still be captured as code duplicates.

The main problem with this kind of clone is when the developer wants to modify or change these blocks of code or when they find bugs, it will be very difficult to fix and change these blocks everywhere else, and it is possible that the developer may be completely unaware of the other clones. If any logic on which this block depends changes, then all the blocks may be harmed, and it may be difficult to find all the blocks affected.

The solution for this problem is relative to the size and number of clones that occurs across files. In certain contexts it might be proper to leave the clones as it is, such as in the case of `if` or `case` statement, sometimes making function calls may break the understanding of the logic of the code. In other cases a common library should be made.

4.7 Initialization and finalization clones

This type of clone occurs within the same file or across file systems when initializing data parameters or cleaning up at end of function; we have found that the main portion of the function can perform quite different tasks. This usually occurs when using the same data types or when performing the same tasks such as memory allocation and de-allocation or variable initialization. Finalization clones often encompass exit conditions and logging.

Problems with this type of clone are much less severe than other clone types, and in many cases are unavoidable. Certainly increased code size may be an issue, but other problems related to code duplication do not seem as large of a concern.

Solutions to this sort of problem may be the use of macros or functions, but this seems too complex for something that is of such little issue.

5 Case Study Results

In this section, we discuss cloning activity in terms of clone pairs, not numbers of lines cloned. We consider that discussion about the number of lines that have been cloned can be misleading and confusing. In the case of clones within the same file, many clone pairs may overlap each other, in contrast to clone pairs outside of the directory, which in many cases do not intersect. The latter will seemingly have a larger number of cloned lines than the former, but in fact the degree of the cloning activity might actually be higher in the former.

In regards to the total number of lines cloned, allowing for lines to be counted more than once did not prove to be

any more beneficial than discussing clone pairs, so we chose the former for simplicity.

The Linux file system contains 42 different file-system implementations in C. There are 538 `.c` and `.h` files, with a total of 279,118 lines of code. We detected 3116 clone-pairs after filtering, giving us 33,707 unique lines, or 12% of the source code, that were involved in code cloning activity. The average length of the clone pairs is 13.5 lines, with a median of 12 lines, an upper quartile of 15 lines and lower of 8 lines. The minimum length is 1 line and the maximum is 123 lines.

5.1 Families of Systems Based on Duplication

As illustrated in Figure 2, several families of file–systems, or groups where code is similar, become apparent. The most notable is the shared code between `ext2` and `ext3`. Here we see 85 clones common between the two file systems. After investigating the code, the reasons are very obvious. `Ext3` is based on `ext2`, and it appears the development of `ext3` started by copying all of `ext2` into a new folder.

Two unexpected results appeared when viewing this chart. The `intermezzo` file-system seems to be highly related to the main file-system code. By inspecting the code, we see that much of what was cloned involved getting and setting the path, and various navigation codes. The `intermezzo` was inspired by `coda`, although re–engineered and restructured, and we see some significant evidence of this by 11 clones appearing between the two. We also see that the `JFFS` file-system has cloned much from the `inflate_fs`. Here we see that most of the clones in this case are grouped into one file, although they are taken from many files within `inflate_fs`.

5.2 Frequency of Clone Types

As can be seen in by Figure 1, the major cloning trend is to duplicate code from within the same subsystem. 78% of code duplication occurs from within the same directory. Some notable exceptions to this trend are `ext2`/`ext3` and `AUTOFS`/`AUTOFS4`. In these cases, `ext3` was created based on `ext2`, and `AUTOFS4` was created based on `AUTOFS`.

This result is significant. It suggests that problems associated with code duplication such as copying bugs in most cases will be restricted to within a single subsystem. This also gives developers good reason to focus their efforts on eliminating code duplication within subsystems first before doing system wide repair.

Reasons for this are probably the most obvious ones. The developer is most familiar with his/her own code, so is most likely to use code from within their own system. As well, because it is within the same system, it is more likely that relevant and similar code exists in this system.

Table 1 shows the number of clones that occur in the same file, in the same directory but not in the same file, and in different directories. From this table, we can see that the average size of a clone pair (the number of lines of code) is nearly the same, but the number of clones that occur in the same file is more than double the number of clones in the same directory but different files, or in different directories. We saw this again when analyzing the cloning activity on the 3D bar charts as described above.

Table 2 summarizes the frequency of the various types of clones. In this table, one should note that when we say that a function has been cloned, we mean that more than 60% of the code between two functions has been cloned. The number of duplicated functions in this table refers to the number of duplicated function pairs, or in other words pairs of functions that are in a clone relationship. In this table, count refers the number of occurrences of that type of clone, for example in the table we see 589 blocks of code were cloned in the same function, and 244 functions were cloned in the same file.

From Table 2, we see that over 30% of the clone pairs that occur within the same file are blocks of code duplicated within the same function. We also see that 244 function pairs occur within the same file. This number can be decomposed somewhat. From these pairs, there are 293 unique functions that take part in a code clone relationship. 341 clone pairs combined form this group of clones, of which 173 clone pairs encompass more that 60% of the functions which have been cloned.

Within the same directory, we see that there are 653 function pairs that are in a clone relationship. 658 clone pairs contribute to this, making more than 80% of the clone pairs occurring in the same directory but not in the same file part of a function clone relationship. 166 unique functions were cloned, meaning that many of these function pairs are part of larger clone classes.

Outside of a directory, there are 129 function pairs, with 156 unique functions. 175 pairs contribute to these full function matches. From this result, we see that function cloning decreases significantly, even though the actual amount of cloning activity does not drop so dramatically. We also see that functions are less likely to appear in clone classes when cloned across directories.

In Table 3 we see that the metrics–based clone detection validates our results that are based on parameterized string matching. Regardless of the constraint of minimum lines of functions, in all three cases, cloning of functions was most often found within the directory but not the same file, followed distantly by cloning of functions in the same file, and then cloning functions from outside the directory, just as

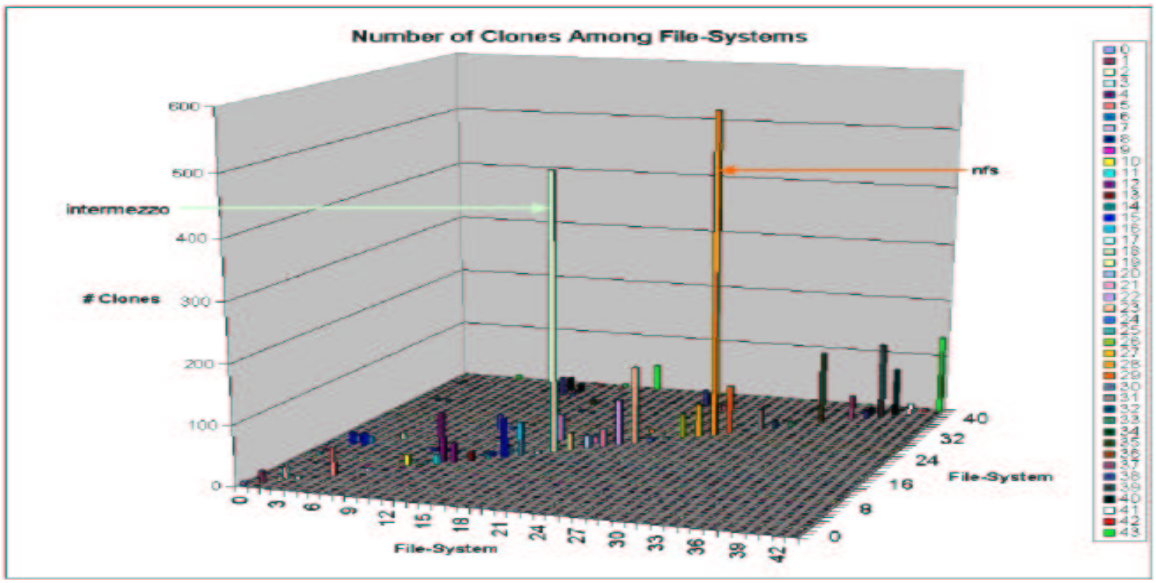


Figure 1: Number of Clone Pairs Between File Systems

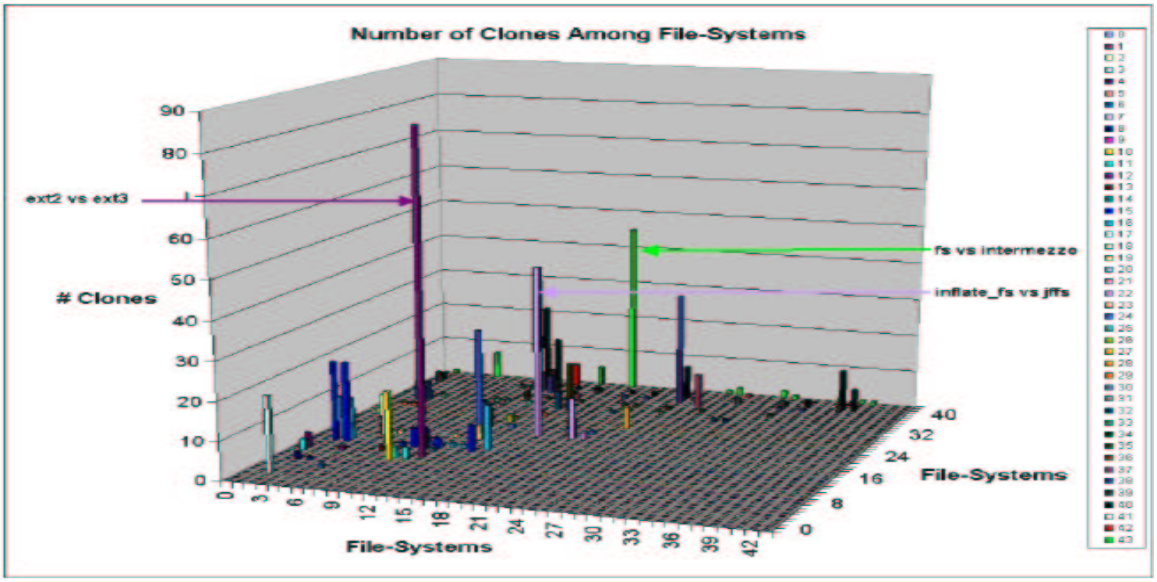


Figure 2: Number of Clone Pairs Between File Systems (excluding themselves)

	Clones in Same File	Clones in Same Directory	Clones in Different Directories
# of clone pairs	1628	806	682
Average LOC	12.7	14.5	14.3
Max LOC	63	71	123
Min LOC	2	4	1

Table 1: Profiles of cloning locality — All clones

Type	Count	Average Length
Same File		
Blocks in Same Function	589	13
Duplicated Functions	**244**	26
Initialization Clones	28	14
Finalization Clones	82	13
Cloned Blocks	588	13
Same Directory		
Duplicated Functions	**658**	16
Initialization Clones	2	14
Finalization Clones	11	10
Cloned Blocks	135	14
Different Directories		
Duplicated Functions	**129**	27
Initialization Clones	6	12
Finalization Clones	45	11
Cloned Blocks	456	14

Table 2: Frequency of various clone categories — Parametric String Match

what can be seen in the previous section. It is interesting to note how quickly the number of functions drops off as the minimum number of lines of a function is increased. A large portion of the functions that we lose are false matches, although some are not.

Initialization clones and finalization clones were not as frequent as were first expected they might be. The clones we did find, however were significant. We expect to find more of these clones in other parts of the Linux kernel, in particular driver source code. A surprising result is that initialization clones appear to occur much less often than finalization clones. After inspection of code block clones, we see that when code for initializing a function is copied, often local variables are added to the cloned list or removed from it. This makes it difficult to classify many of the initialization clones automatically. Therefore, we take the frequency of initialization clones as an underestimate. Better approaches to automatically find this class of clone need to be investigated further.

Cloned blocks of cloned code are difficult to characterize completely, as there are many circumstances leading to the cloning of these blocks. However, we have found that locality does correlate somewhat to the structure and reasons of this type of clone. In the cases of clones in the same file and same directory, these clones are often the product of copying blocks contained within a control structure, such as `if/else` statements. In some cases however, they are what remains of what was once a initialization clone or a finalization clone.

When we inspect clone blocks across directories, it is often the case that the blocks are the remains of copied functions, changed enough that the functions no longer can be classed as cloned functions technically, but by manual inspection these functions are still clearly in a form of clone relation. These blocks raise interesting questions about the evolution of clones.

In many cases, clone blocks represent function pairs where 60% or more of one function has been copied to another, and additional states have been added. These function pairs, which we call partial-match function pairs, represent an interesting form of code cloning. It would be difficult for this form of function cloning to be detected by metrics based clone detection algorithms, but they are certainly function clones. In many cases, one function is entirely copied, and a

	Metric Match			String Match
Minimum Function Length (LOC)	5	6	7	N/A
Same File	141	110	108	244
Same Directory	1157	1152	619	658
Different Directory	116	80	38	129

Table 3: Number of function clones found in metrics based clone detection and parameterized string match

significant number of statements have been appended to the end. In total, these partial function matches accounted for 72 same file clone blocks, 22 same directory clone blocks, and 109 different directory blocks.

This case study shows that the taxonomy is not complete. The presence of so many cloned blocks may be an indication that more categories of clones exist, and further investigation must be done.

5.3 Metrics vs. Parametric String Matching

In Table 4, we see the summary of results in comparing the function pairs found by the metrics method to those found by the parametric method. The first row presents the number of function clones found in both the metrics based clone detection and the string matching algorithm at the same time. Then second and third row show the number of function clones found by each detection method exclusively. In all cases, we see that between 708 and 716 function pairs were found by both methods. These function pairs are in most cases very clearly clones of one another. Also, in all cases, between 353 and 361 function pairs were only found by the parameterized string based approach. In these cases, the functions tend to be longer than average, their average length being 30 lines of code. Often, lines have been added and removed, or fan in or fan out metrics have changed. This shows us that using exact match criteria may not always be sufficient in searching for function pair clones.

In the cases of functions pairs found only by metrics we see two things. First, functions of sizes five LOC and six LOC are often too small to be detected when using a minimum criteria of 30 tokens in the parameterized string approach. Secondly, small functions of sizes five, six and seven LOC are often hard for parameterized string matching to detect clones in when there are enough tokens present. This is because if one token violates the parametric match, then there is little chance that enough tokens were already matched to make a clone that is large enough to report, and there is also little chance of enough tokens remaining in the function to find another clone. Often a function call that takes a different number of parameters or changed mathematical operators can cause the parameterized string matching to miss matches in small functions.

In general, we found when using our metrics-based clone detection tool, it was better at finding small function matches than CCFinder, but CCFinder was better at finding large function matches. When using CCFinder with the Gemini GUI, we found that it was difficult to grasp the total cloning activity in the system, but when clone pairs were grouped by the taxonomy we have presented, interesting cloning activity becomes more evident. We found that metrics-based clone detection finds very close matches, but CCFinder is able find function matches which exhibit more change. As a preliminary result, we found that parameterized string matching presented more interesting and useful clones to use than using ExactMatch metric-based clone detection. Future work will investigate the comparison of these two approaches but allowing more flexibility in the metrics-based matching.

6 Related Work

There are several types of clone detection techniques that have been developed. Metrics-based clone detection tools which detect clones of full blocks of code such as functions based on various metrics extracted from them have been developed by Mayrand et al. [20] and Kontogiannis et al. [18]. Parameterized string matching is discussed by Baker et al. [2, 3] and Kamiya et al. [15]. Baxter et al. [7] have developed a clone detection tool by performing subtree matching on abstract syntax trees. Program dependence graphs have been used by Krinke et al. [19] and Komondoor et al. [16] in detecting duplicated code. Johnson [13, 12] proposed using a fingerprinting algorithm on substrings of the source code. Kontogiannis et al. define two other methods to detect clones in [18]:dynamic pattern matching which finds the best alignment between two code fragments, and statistical matching between abstract code descriptions patterns and source code. Balazinska et al. [4, 6, 5] uses metrics based clone detection to quickly find candidate clones and uses an algorithm based on Kontaogiannis et al.'s dynamic pattern matching algorithm.

Clone detection case studies on the Linux kernel have been reported in [10, 8, 1]. In [8], Casazza et al. use met-

Minimum Number of Lines	5	6	7
Function pairs found by both	716	716	708
Found in Parametric Only	353	353	361
Found in Metrics Only	698	626	57

Table 4: Comparison of # of function clones found by the two clone detection algorithms

rics based clone detection to detect cloned functions within the Linux kernel. They performed analysis across the major subsystems, and then on the architecture dependent code of the memory management subsystem and the kernel core. To evaluate the degree to which cloning occurs, they define a common ratio between two files, which is the percentage of functions in one file which are cloned in another with respect to the number of functions in the first. As noted in [1], this common ratio must be used with great care and absolute values need to used as well. The conclusions of this study were that in general the addition of similar subsystems was done through code reuse rather than code cloning, and more recently introduced subsystems tended to have more cloning activity. Antoniol et al. [1] did a similar study, evaluating the evolution of code cloning in the Linux. They too used function metrics clone detection as their technique and their conclusions were the same, adding that the structure of the Linux kernel did not appear to be degrading due to code cloning activity. In [11] a preliminary investigation of cloning among Linux SCSI drivers was performed.

Kamiya et al. [15] performed tests on JDK to search for clones within the system, and they studied the cloning behavior between Linux, FreeBSD, and NetBSD. While [15] observes that clones in JDK seem to occur in near directories or files based on visual inspection of the scatter plot their tool presents, no quantitative data analysis is discussed concerning this point. None of the above studies have discussed the types of clones they have found, or discussed the locality of code cloning in detail other than comparing the level of cloning amongst subsystems.

A work similar to this also tries to categorize clones for the purpose of software maintenance. In [4], Balazinska et al. create a schema for classifying various cloned methods based on the differences between the two functions which are cloned. The results produced in [4] are used by Balazinska et al. in [6, 5] to produce software aided re engineering systems for code clone elimination. This differs from our work in that our classification scheme is based on locality as well as clone type, and copied functions are only one type in our case, although in [4] they break this down into 18 categories. One of our main research goals is to determine how much developers clone and from where. This question is not answered by the clone classification scheme in [4]. In addition, this work ignores code clones which are not function clones.

7 Summary and Conclusions

This preliminary study began as in-depth evaluation of cloning in a large software system. In this study we found that the Linux file-system subsystem has a significant amount of code duplication within it, the majority being localized within each individual file-system type, or sub-subsystem, similar to the activity in JDK noted in [15]. We also defined a preliminary taxonomy by which non-function and function clones can be categorized. This will be used in future research when characterizing cloning in all of the Linux kernel.

Our first goal, to begin to produce a finely grained analysis of code cloning in a large scale software system has begun, and future work will attempt to characterize more subsystems, in particular the driver subsystem where source code and functionality is vastly different from the Linux file-system subsystem. This work will provide support for generalizing these results, as well as more insight into the growth of the Linux kernel as documented in [11].

During our study, we found that 3D visualization provides much convenient information. From the 3D bar charts we were able to see very quickly related groups of subsystems, and also which subsystems were trouble spots for cloning activity. Further investigation on the scalability of the graph is needed, but at the current time we would suggest that including the ability to visualize clone detection results in such a way may be a very useful addition to maintenance environments involving clone detection.

8 Future Work

Research on this topic is ongoing. We intend to continue this study, to fully characterize the Linux kernel in terms of code clone activity. We will also investigate how other subsystems compare to these results. From preliminary testing, many may be quite similar.

We will also evaluate our taxonomy throughout the course of this study.

Additionally, we will study how code clones evolve over time, in particular, we would like to test the hypothesis that many code block clones start out as function clones, and as time goes on, the functions evolve away from one another. We will also investigate the possibility of using previous releases for detecting clones in current releases.

Acknowledgments

We would like to thank Dr. Ettore Merlo for his on going help and advice.

References

[1] G. Antoniol, U. Villano, E. Merlo, , and M. Di Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology 44(13)*, 2002.

[2] B.S. Baker. A program for identifying duplicated code. In *Proceedings of Computing Science and Statistics: 24th Symp. Interface*, pages 49–57, 1992.

[3] B.S. Baker. On finding duplication and near-duplication in large software system, 1995.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 292–303, 1999.

[5] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *The Proceedings of the 6th. Working Conference on Reverse Engineering*, pages 326–336, 1999.

[6] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, pages 98–107, 2000.

[7] Ira D. Baxter, Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[8] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[10] Michael W. Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in Linux: A case study. A presentation at the 2000 CASCON workshop on 'Detecting duplicated and near duplicated structures in largs software systems: Methods and applications', on November 16, 2000, chaired by Ettore Merlo; available at `http://plg.uwaterloo.ca/~migod/ papers /cascon00-linuxcloning.pdf`.

[11] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, 2000.

[12] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintanence*, pages 120–126, 1994.

[13] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON 93*, pages 171–183, 1993.

[14] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. A token-based code clone detection tool - ccfinder and its empirical evaluation. Technical report, 2000.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[16] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–??, 2001.

[17] K Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of Working Conference on Reverse Engineering*, pages 44–55. IEEE Computer Society Press, 1997.

[18] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection, 1996.

[19] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253. IEEE Computer Society Press, 1996.

[21] Qiang Tu and Michael Godfrey. An integrated approach for studying software architectural evolution. In *Proceedings of 2002 International Workshop on Program Comprehension (IWPC-02)*, 2002.

[22] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE Computer Society Press, 2002.

Using software trails to rebuild the evolution of software

Daniel German
Department of Computer Science
University of Victoria
dmgerman@uvic.ca

Abstract

This paper describes a method to recover the evolution of a software system using its *software trails*: information left behind by the contributors to the development process of the product, such as mailing lists, Web sites, version control logs, software releases, documentation, and the source code. This paper demonstrates the use of this method by recovering the evolution of Ximian Evolution, a mail client for Unix. By extracting useful facts stored in these software trails, and correlating them, it was possible to provide a detailed view of the history of this project.

1 Introduction

Investigating and recovering the evolution of a software project requires a combination of skills: it is necessary to understand the software product, its features, its components and how these have evolved; it is necessary to find, recover, and catalog valuable facts about the history of the project; it is also required to look at the developing team, in order to better understand the software process they have used, their interrelations and communication, their decision taking and their skills; it also required to start piecing together all this information, proposing potential hypothesis that are then proved or disproved. One can equate the work of a "software evolutionist" to the morphing of a software architect, a historian, an ethnologist, an anthropologist, a paleontologist, and a private investigator.

In rare cases, the evolution of a software product is recorded by an insider. This software evolutionist has access, presumably, to all the personnel and available information, and has the potential to accurately record its history as it unfolds. Unfortunately few software projects have this type of resident historian, and it is usually an outsider who has to the work. This external software evolutionist could track the project for some time, looking from the outside at how the project continually evolves. Sometimes her work is done post-mortem, looking at the remnants of the project, like an anthropologist looking for clues of how an ancient civilization functioned.

An outsider evolutionist depends on the available information of a project to tell its history. This paper defines "software trails" as pieces of information left behind by the contributors of a software project. Examples of software trails are

configuration management systems logs, email messages, documentation, recordings of conversations, product releases, and of course, the source code and other required files themselves.

Software trails have the potential of keeping a "community" memory of the software development. Linus Torvalds, for example, has repeatedly said that he would not have a telephone conversation to discuss the development of the Linux kernel, because he wants every decision to be recorded for posterity. The open source community recognizes that, given the volatility of core developers and an unforeseen future, keeping this information can provide important facts critical for the long term survival of a project, taking it from one set of developers to the next one and from one maintainer to another. Arguably, the best open source success stories tend to keep very detailed software trails. These trails can be used for two purposes: to educate future developers on the characteristics of the product, and to assist in recovering the history and evolution of the product. Closed source software projects are also interested in keeping this memory, as they know that their developing teams evolve with time and they cannot be dependent on one person maintaining this information in her head.

From the point of view of the software evolutionist, the software trails left behind by a project are a gold mine, ready to be exploited. This comes at a cost: the amount of information available can be overwhelming. It is necessary to assist the software evolutionist in the process of recovering, cataloging and correlating the information available, in order to help her look at the "big picture", but at the same time, provide enough detail about a particular event in the history of a project.

2 Recovering the evolution of a project from its software trails

This paper proposes a method of recovering the evolution of a software system by analyzing the software trails left behind during its development. For example, how the version control management system logs, messages in its electronic mailing lists and the defect control system logs can be correlated to retrieve important decisions and events, and to trace how the software has evolved since its conception. This method is composed of four steps:

1. **Define Schema**: Create a schema that represents the information available in the software trails, including any relationships between them. For example, that a developer has a list of different email addresses used to post to the mailing lists; that a particular defect was fixed by a given developer with a given set of software changes; that a software change includes a delta of the change, and a version number; etc.

2. **Gather Software Trails**: Retrieve the available software trails and map them into this schema. Often the logs of these trails are not easy to parse nor translate. In some cases heuristics need to be developed and applied.

3. **Extend Information**: The available software trails can be extended by further analysis, enhancing them by extracting new facts or creating new relations as it is found appropriate. For example, many open source developers

do not use configuration management software, and the developer usually states informally (in the version control log) that a given set of changes corresponds to a defect fix; the version control log has to be parsed in order to find something that "might look" like its corresponding defect number.

4. **Analyze**: The final step is to look through this data and try to find interesting events in its development that can tell the history of the project. This is a difficult problem. As the software evolves and grows bigger, its available information grows at the same time, making it difficult for the evolutionist to find the "more" relevant information that tells an interesting fact about the history of the project.

Given the informal nature of some of this trails (and the fact that it is a reverse engineering process, where the evolutionist has no certainty on what were the actual events, and she is just merely trying to reconstruct them from the trails available) the experience and insight of the evolutionist and amount of time that she invests in the analysis of the information available will have an important impact in the quality of the results.

In order to demonstrate the above methodology, this paper uses the software trails of Ximian Evolution to recover its evolution. Evolution is a mail client (similar in scope to Microsoft Outlook) that is starting to gain popularity in the Unix world (Ximian was bought by Novell in August, 2003). Evolution developers have left software trails in mailing lists, Web sites, its CVS repository logs (CVS is one the most widely used version control systems), documentation, inside and outside the code, and Bugzilla, its bug tracking system.

3 Methodology

The following software trails were used in the recovery of the evolution of Evolution:

- Version source code releases. As of May, 2003 there have been 37 different releases. These come in the form of tar files that contain all the necessary files to build and run the product. They are made available for the people who are interested in recompiling the product to suit their particular installation. Five of these releases are considered major (0.0, 1.0, 1.1.1, 1.2.0, and 1.3.1). Evolution has adopted a numbering scheme similar to the Linux kernel, using odd numbers in the second component of a release label (such as 1, or 3 in 1.1.1 and 1.3.1 respectively) to denote "unstable" releases that are considered to be riskier (buggier) than the stable ones (like 1.0 and 1.2.0). Source code releases can be seen as a coarse grain view of the evolution of a project. A collection of scripts and tools such as "exuberant ctags" and "stripcmt" were used for fact extraction (similar to the fact extraction in [GT00]).

- CVS logs. CVS keeps track of who modifies which file, and the corresponding delta associated with the modification. This change is known as a "file revision". CVS keeps information such as who made the revision, when the actual diff of the revision, number of lines added, and number of lines removed. softChange [GM03] was used to recover the information from these

logs and to enhance it. For instance, CVS does not keep track of which files are modified at the same time. softChange analyses the logs, and rebuilds these groups of files, which are then called Modification Requests (MRs). A modification request is a request by a contributor to commit a group of files at the same time. The belief is that if two files are part of the same MR, it is because they are somehow interrelated. Contrary to source code releases, CVS logs provide a very fine grained view to the evolution of the project. A snapshot of the CVS log was taken on May 21, 2003.

- Mailing lists. Evolution maintains at least two mailing list, but some of the information related to it can also be found in the GNOME mailing lists. GNOME (GNU Network Object Model Environment) is a free software collection of libraries and end-user applications that provide a graphical "desktop" for Unix systems. The project started in 1996 as a volunteer effort, and has evolved into a large system that involves paid and non-paid contributors, and it is currently shipped with almost every Linux distribution. GNOME is the parent project of Evolution. Mailing lists tend to serve as a record of important decisions related to a project. Another use of mailing lists is to announce the availability of new releases (including a summary of the new features found in it).

- ChangeLogs. The main source of documentation is the ChangeLog. As the GNU ChangeLog standards indicate, the ChangeLog explains how earlier versions of software were different from the current version.

For the purpose of this paper, softChange was extended to generate relational data, which was then imported into a `postgresql` database (the dump of the database measures 0.5 Gbytes, although some tables contain redundant information to help speed up queries –a copy of the database is available on request and it is available for download at `view.cs.uvic.ca/evolution`, along with the rest of the data used in this analysis). The analysis of the data was done ad-hoc, writing SQL queries. The results of these queries were then plotted using gnuplot.

4 Evolution

During the beginning of 1999, Bertrand Guiheneuf started working on a new mail client for the GNOME project [Gui00]. One of his goals was to create a better mail client than Balsa (then GNOME mail client) and to use Bonobo (GNOME CORBA implementation) to display the different content types in email messages. He decided to start the project by implementing a mail storage library, which he called *camel*. In Guiheneuf's view, Balsa was not good enough. He planned, however, to phase in the development of camel by incorporating its storage library into Balsa (and other potential mail clients) using CORBA [Gui99].

The *GNOME Mailer* project was formally started in April 16, 1999 with a mail message from the GNOME project leader Miguel de Icaza that discussed the need for a more powerful mail client [dI99a]. One important issue that de Icaza addressed in this message was why not to further develop an already started project (such as Balsa). His answer was "there is too much baggage in existing

mail applications that we do not want to carry into the future". This message was probably triggered by Guiheneuf's posting (two weeks before). de Icaza proceeded to outline the main architecture that this client should have (which was further refined in [dI99b]):

- **Storage**. This module was to be composed of two parts: a) it will include a library to understand and interact with a variety of mail storage formats and sending email protocols (imap, pop, spool mail, UNIX mailbox, MH); and b) contain a query engine to filter, move and delete mail.
- The **Folder and Summary Display** would be the main GUI to email messages and folders.
- The **Message Display** would be responsible for displaying a particular mail message.
- The **Message Composition** would implement an editor that would allow the user to create and edit mail messages.
- Interface with the **calendar** and **addressbook** (which in de Icaza's opinion needed to be redesigned).

De Icaza, following Guiheneuf's ideas (and the trend of GNOME in general), proposed to use CORBA for communication between these modules and other applications that would help display different content types in the message display (at the time, there was a move towards making most GNOME applications CORBA aware). This module list would also serve as a way to divide the work into pieces in which different developers could concentrate and work as independently as possible. A mailing list was created for the project, and during the month of April 1999 more than 500 messages were exchanged, most of them related to requirements analysis for the new project.

Guiheneuf would become the first maintainer of the new GNOME Mailer, continuing the development of camel as its storage module. In August 1999, the name Evolution was proposed by him, and it was quickly accepted by the GNOME community[1].

In October 1999, Miguel de Icaza created Helixcode (now Ximian), a commercial venture aimed at continuing the development of GNOME, planning to generate income by selling services around it. Ximian proceeded to take under its wing the development of Evolution and has committed several employees to work on it.

In 4 years Evolution has grown into a powerful product that is starting to be widely used in the open source community. Evolution recently received the "2003 LinuxWorld Open Source Product Excellence Award" in the category of "Best Front Office Solution". One of the objectives of Evolution is to provide a free software product with functionality similar to Microsoft Outlook or Lotus Notes[Per01]. Table 1 lists the main events in the history of the project.

[1] Guiheneuf proposed e-volution, which was quickly altered to evolution. The name was later changed to Evolution and finally to its current official name Ximian Evolution.

Milestones	Date
Coding of camel starts	1999-01-01
Evolution starts	1999-04-16
Ximian is established	1999-10-01
Version 0.0	2000-05-10
Version 1.0	2001-11-21
Version 1.1.1	2002-09-09
Version 1.2.0	2002-11-07
LinuxWorld “Best Front Office Solution” award	2003-01-23
Version 1.3.1	2003-02-28

Table 1: Main milestones of the project

4.1 Releases

Figure 1 shows the growth in the size of the source code releases of Evolution. It was discovered that the total size of the release (sum of the size of all files) and the total size of the source code (sum of the size of all source code files) did not show a clear correlation. Further investigation demonstrated that the main culprit for the increase of the size of the release is its internationalization (translation files with extensions .po and .gmo). The latest version, for example, totals 64 MBytes of which 37 Mbytes (57%) are internationalization files, compared to only 11 Mbytes of source code (17%). Evolution is currently translated into 34 different languages (this does not include regional variants; for example, Evolution includes internationalization files for Portuguese and its Brazilian variant). Another surprise is to discover that the next largest contributor to the size of a release is ChangeLogs: 4.6 Mbytes (7%). ChangeLogs will be revisited in section 4.3.2.

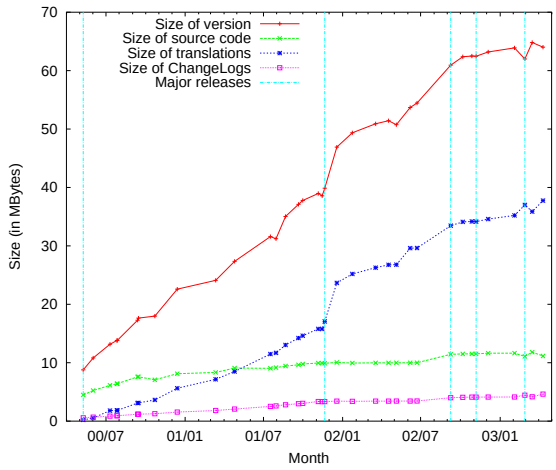


Figure 1: Size of releases over time. The plot shows also the total sizes (in Mbytes) for source code, internationalization files, and ChangeLogs. Together these 3 types of files account for more than 80% of the size of the latest version.

The number of files shows a different picture. The average proportion of source files in the releases is 46% (6.16 std deviation). In contrast, the proportion of

translation files is 2.7% (0.29 stddev), and 1.1% (0.03 stddev) for ChangeLogs. Translation files and ChangeLogs are therefore few, but very large, when compared to source code files.

Figure 2 shows, for a given release, the number of source code files, total LOCS and total cleanLOCS (number of LOCS when comments and empty lines have been removed). The average size of a source file has been stable across versions, at 639 (25 stddev) LOCS per .c and 101 (7.6 stddev) LOCS per .h file. The proportion of cleanLOCS to LOCS has also remained stable across versions, at 72.5% (1.4 stddev) for .c files, and 60% (2.6 stddev) for .h files.

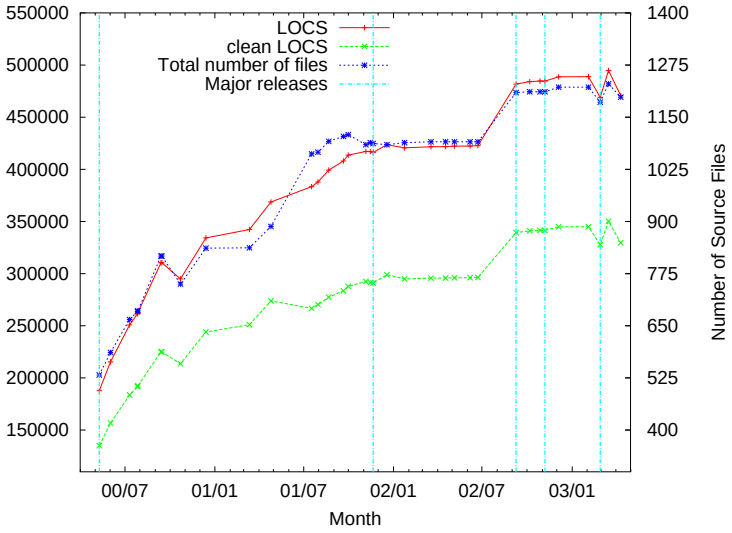


Figure 2: LOCS of releases over time. The plot shows the total number of files, and the number of clean LOCS (LOCS without comments nor empty lines).

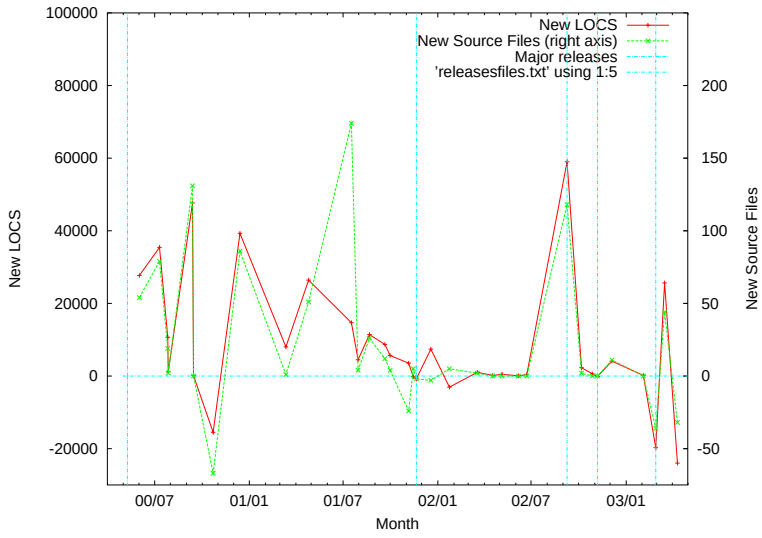


Figure 3: Changes in LOCS and number of files, per version

The actual change in LOCS from one version to another show an interesting

story. Figure 3 shows the increment in the LOCS and number of files over time. Of special interest are the negative increments in either LOCS or source files, suggesting removal of source code. For example, in version 0.6 (released 2000/10/23) 15.5 kLOCS and 67 source code files were removed with respect to the previous version (0.5.1). Between these two releases 157 source code files were deleted and 90 created (45 kLOCS were deleted and 24k LOCS added). Further analysis of the available software trails showed that for this release it was decided to move several widgets (from Evolution's GUI) to the Gal project. Gal, according to its official description is "the GNOME Application Library, a collection of widgets and other helper functions originally extracted from Evolution and gnumeric (GNOME spreedsheet)". In fact, the first version of Gal (0.1) was released in Oct. 5, 2000 [dI00], 5 days before Evolution 0.6 (and the sudden drop in LOCS).

4.2 Development Activity

One important question that arises when looking at the increment in the size of Evolution is how does it correlate to the actual activity of the developers? The CVS logs provides some useful information that can be used to attempt to answer this question.

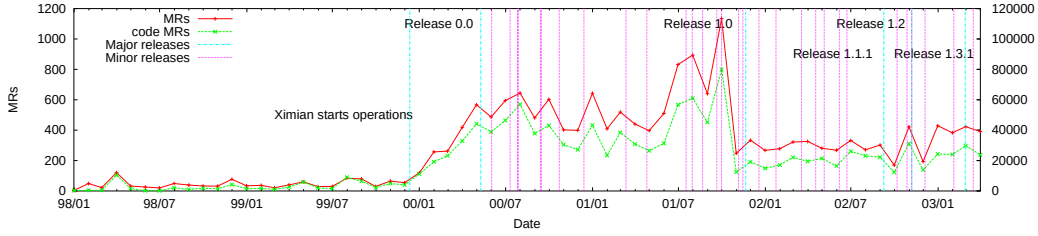


Figure 4: Evolution of the project in number of MRs

Figure 4 shows the number of MRs per month for Evolution. The plot also shows the different releases in the project. There are several interesting observations from this graph. First, the development activity was relatively flat during the first year of the development, and it is not until Ximian is bom that there is a surge in the number of MRs. The number of MRs surges just before release 1.0. After that, the number of MRs remains more stable, but still shows peaks that correspond to releases. Because it is not possible to have access to the actual number of hours spent per developer in the project, it is not possible to determine the development effort spent per MRs, and therefore, if less MRs mean less developer-time, or if some MRs required more time. In the same figure, the number of MRs that involve source code (codeMRs) is also shown. The proportion of codeMRs to MRs has decreased during 2003 (approximately 38% of the MRs do not involve source code).

Why has the proportion of codeMRs dropped? The exploration of the logs drew the following conclusions. From all MRs in 2003, 86% corresponded to changes in source code (61%), translations (13%) and changes to metafiles (files with extension .am and .in, 18%)[2]. Metafiles are used by the automake and autoconf tools to

[2]Some MRs included changes to Metafiles and source code, and some MRs included changes to metafiles and translations

create other files. The most common use of these Metafiles is the creation of Makefiles (the developer creates an .am or .in file, and autoconf and automake create the corresponding Makefile). Metafiles rely heavily on macros (GNOME provides a module called macros with the majority of these definitions).

A surge in the activity related to Metafiles and translations was to blame for the drop in the proportion of codeMRs. The question that followed was, what prompted the surge in Metafile activity? In those MRs 70% of the revisions corresponded to Makefile.am files; and 12% of the revisions corresponded to changes to `configure.in`, the main autoconf file that drives the configuration of Evolution when a user wants to compile it. Inspection of the ChangeLogs seems to suggest a conscious effort to cleanup the Metafiles.

The surge in changes to the translations is attributed to a previous significant change in the UI. Once the development team decides to make a "freeze" in the features of a release, translators start making changes to the corresponding translation files.

Another question prompted by figure 4 is why does it show activity before January, 1999? It appears that some code that was in development previous to Evolution was later incorporated into it (one widget and some calendar related code). It is also suspected that some revisions contain invalid dates, suggesting that during a period of time the machine's clock was set to an incorrect time.

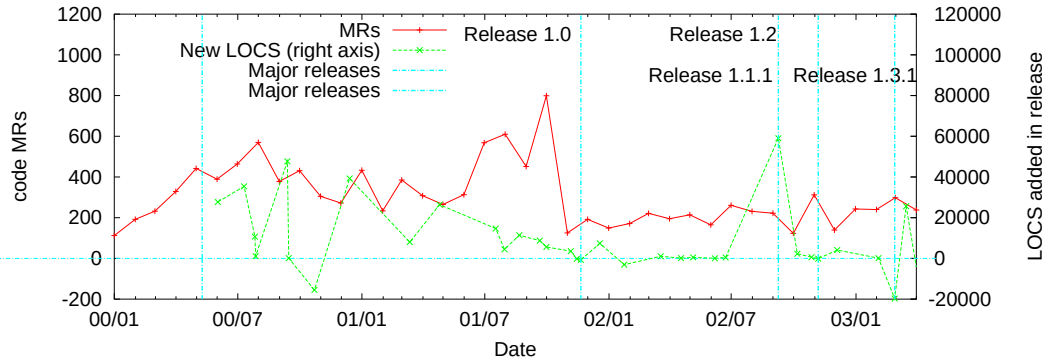


Figure 5: Changes in LOCS and number of files, per version

Figure 5 shows codeMRs and how they relate to the actual growth in the size of the source code in the releases. Even in periods where the code base does not increase (like the first half of 2002) the number of MRs is still large. This suggests a period in which debugging took precedence over development of new features.

4.2.1 Characteristics of MRs

It is also interesting to see the typical characteristics of an MR. Figure 6 shows the number of files per MR. Most of them contain a small number of files, which is a healthy sign. The log for the largest MR (which contains 650 files, in 2001/06/23) reads "Update the copyrights, replacing Helix Code with Ximian and helixcode.com with ximian.com all over the place.". That day a total of 709 files were modified. Similarly, the largest number of files modified in a single day was 1417 (2001/10/27) and the reason was "update the licensing information to require ver-

sion 2 of the GPL (instead of version 2 or any later version)". These two explanations highlight a particular feature of MRs in Evolution: developers take good care of explaining in each MR the reason for the change (CVS allows developers to add a log message to each MR). The average log for an MR is 300 characters (561 stddev, 170 median), with a minimum length of 1 (only 8 MRs) and 18K for the longest log (which involved the merging of a branch to the main CVS tree).

From a total of 18K MRs, only 87% include two or more files in it. A preliminary analysis shows that most of these MRs are of two types: a) files which were overlooked in a previous MR and committed minutes later; and b) minor corrections, such as fixing spelling mistakes. Further analysis is needed to corroborate this hypothesis.

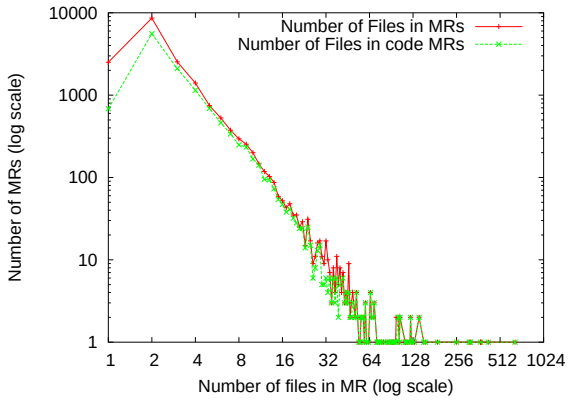


Figure 6: Most MRs contain a small number of files

4.2.2 Contributors

There is a common belief that open source projects are developed by a large number of individuals. While that is true, it is important to recognize that the contribution of the majority of these individuals is very small. In open source projects, contributors can be divided into two main groups: those with write access to the CVS repository (and can make their contributions to the CVS repository themselves) and those who do not have write access to the repository. In GNOME it is not difficult to get write access to the repository. Once somebody has submitted several contributions, this person can apply for CVS write access. In GNOME, more than 500 people have CVS write access[3].

By looking at the changes committed by contributors with CVS write access, we can see that like many other open source projects, the majority of the coding is done by a small number of individuals. Zawinsky, at one time one of the core Mozilla contributors, commented on this phenomenon: "If you have a project that has five people who write 80% of the code, and a hundred people who have contributed bug fixes or a few hundred lines of code here and there, is that a 105-programmer project?"—as cited in [Jon02].

Evolution contains contributions by 201 different userids (to which, this paper will refer as contributors). Few of these, however, contributes a significant portion

[3]The author has write access to the GNOME CVS repository.

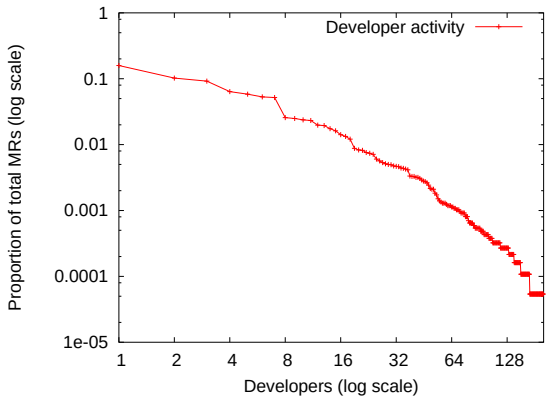


Figure 7: Proportion of MRs per contributor. Each contributor was assigned a number from 1 to 201, which corresponds to the X axis.

of the MRs. Figure 7 shows the proportion of MRs per contributor (each contributor was assigned a number from 1 to 201, which corresponds to the X axis). Only 18 contributors accounted each for more than 1% of the total MRs. The largest contributor is responsible for 16% of the MRs, while at the other side of the spectrum 32 contributors had only one MR only. Furthermore, a total of 48% of the MRs were contributed by only 5 contributors, while 142 contributors contributed just 5% of the MRs (80 contributed a total of 1% of the MRs).

Table 2 shows the 11 most active developers, as a proportion of all MRs. The top 10 appear to be Ximian employees or consultants (see http://primates.ximian.com/). This fact corroborates the hypothesis that private companies (such as RedHat, Ximian, and Eazel) have had a very important effect on the development of the GNOME project [Ger02]. In that respect it is similar to the Mozilla project where core contributors were employees of Netscape (see [MFH02]).

Userid	**Prop.**	**Accum.**
fejj	0.16	0.16
ettore	0.10	0.26
danw	0.09	0.35
zucchi	0.06	0.42
clahey	0.06	0.48
jpr	0.05	0.53
toshok	0.05	0.58
federico	0.03	0.61
peterw	0.02	0.63
iain	0.02	0.65
other	0.35	1.00

Table 2: Most active developers, as a proportion of total MRs

How regularly were contributors participating in the project? The number of different contributors by year is depicted in table 3. After January 2000, in any given month there is an average of 32 contributors (8.3 stddev, minimum 15, maximum 47) per month to the project.

Year	Number of Contributors
1998	37
1999	54
2000	95
2001	98
2002	79
2003	56

Table 3: Contributors to the project by year. It takes into account only those contributors with CVS write access.

4.3 Revisions

Every time a file is modified, CVS creates a record of who modifies it, when, and the "delta" of the modification. This modification is known in CVS lingo as a "revision".

4.3.1 Types of Files

Extension	Prop.	Accum.	Number of files in CVS
.c	0.41	0.41	1195
ChangeLog	0.22	0.62	43
.h	0.13	0.75	1063
.am	0.05	0.81	174
.po	0.04	0.85	71
.ics	0.02	0.87	396
.sgml	0.02	0.90	228
.in	0.02	0.92	136
.png	0.01	0.93	405
other	0.07	1.00	

Table 4: Revisions and number of files per file extension. C files (.h and .c) and ChangeLog modifications account for 75% of total revisions.

Table 4 shows the proportion of revisions per extension (i.e. type of file) and it tells an interesting story. Given that C is the language of choice for Evolution, it is not surprising to see .c and .h files at the top, along with ChangeLogs (ChangeLogs are discussed in detail in section 4.3.2). Metafiles (.am and .in) and translations follow. The next file extension .ics corresponds to files that include information about a particular location in the world, particularly its time zone. There were 1903 revisions made to 396 .ics files, for an average of 4.8 changes per file.

Files with extension .sgml are documentation files. As with many open source projects, the documentation is written in SGML using the docbook DTD. Finally, .png files correspond to artwork.

4.3.2 ChangeLogs

ChangeLog files are an important source of information about the development and evolution of a project. The Evolution developers are fairly consistent in their modifications to the ChangeLog files. From all MRs involving 2 or more files, 93% include a modification to a ChangeLog. Evolution developers seem to make sure that they document their changes in the corresponding ChangeLogs. Table 5 shows the 10 most modified files, 8 of them are ChangeLogs. ChangeLogs (and CVS logs) can provide insight on patches submitted by developers without a CVS account, as developers are expected to be careful to give credit to the patch submitter in the corresponding ChangeLog entry (which are not taken into account for this paper).

4.3.3 Source Code Hot Spots

There have been a total of 41120 revisions to 2258 source code files[4]. Figure 8 shows the proportion of revisions per source code file. 51 files account for 25% of the total number of revisions, while 764 account for only 5% of them.

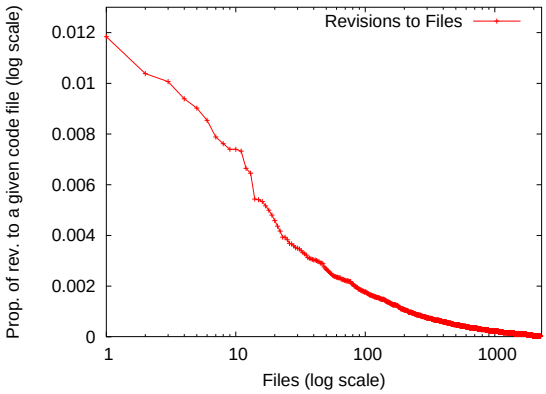


Figure 8: Proportion of revisions per source code file. Each file was assigned a number from 1 to 2258, which corresponds to the X axis.

4.4 Modularization

The success of an open source project depends on the ability of its maintainers to divide it into small parts in which contributors can work with minimal communication between each other and with minimal impact on the work of others [LT00]. From the beginning of the project, there has been a conscious attempt to divide Evolution into modules that fulfill the previous characteristics. Modules are represented in the code base as subdirectories. Figure 9 shows the different modules and the number of MRs for each of them, representing the level of activity in each module.

Figure 10 shows the size of the seven largest modules in Evolution in terms of LOCS. With the exception of libical and widgets, modules tend to grow in size. Before 2002, both libical and widgets show a lot of variability in both their sizes

[4]Many of these files are no longer in the latest release, as they have been removed during the development process. Nonetheless, CVS keeps information about their modification.

File	Prop.	Accum.
mail/ChangeLog	0.04	0.04
calendar/ChangeLog	0.03	0.06
camel/ChangeLog	0.03	0.09
addressbook/ChangeLog	0.02	0.11
shell/ChangeLog	0.02	0.13
ChangeLog	0.02	0.14
po/ChangeLog	0.02	0.16
configure.in	0.01	0.17
composer/ChangeLog	0.01	0.18
mail/mail/callbacks.c	0.01	0.18

Table 5: Top 10 most modified files. ChangeLogs clearly take the lead. As its name implies, mail-callbacks.c contains the callbacks of the mail client, hence the frequency at which it is modified. These 10 files account for a total of 18% of all file revisions.

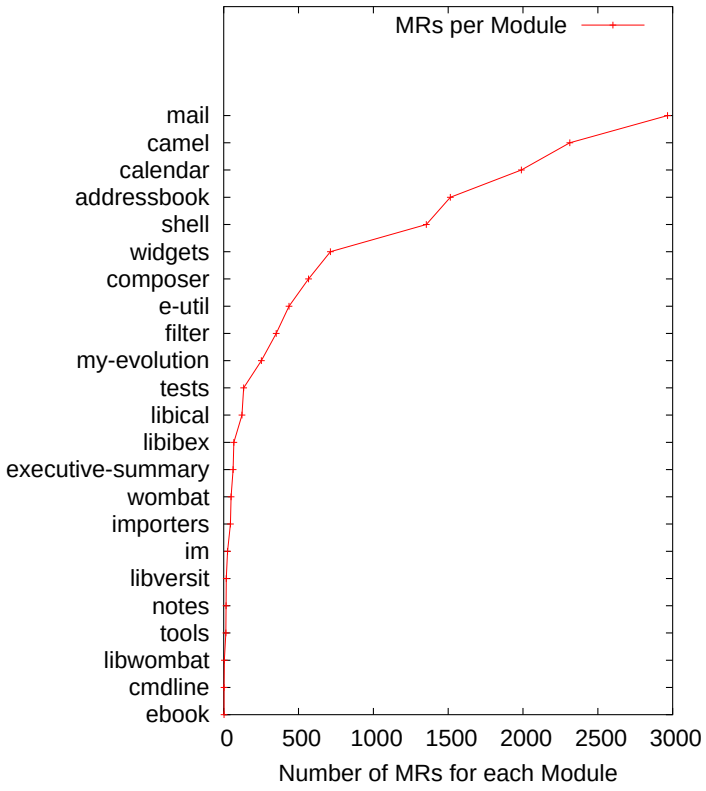


Figure 9: MRs per module. Most of the activity is concentrated in few modules.

and the number of files in them. After Version 1.0, the size of Evolution has been growing at a very small pace.

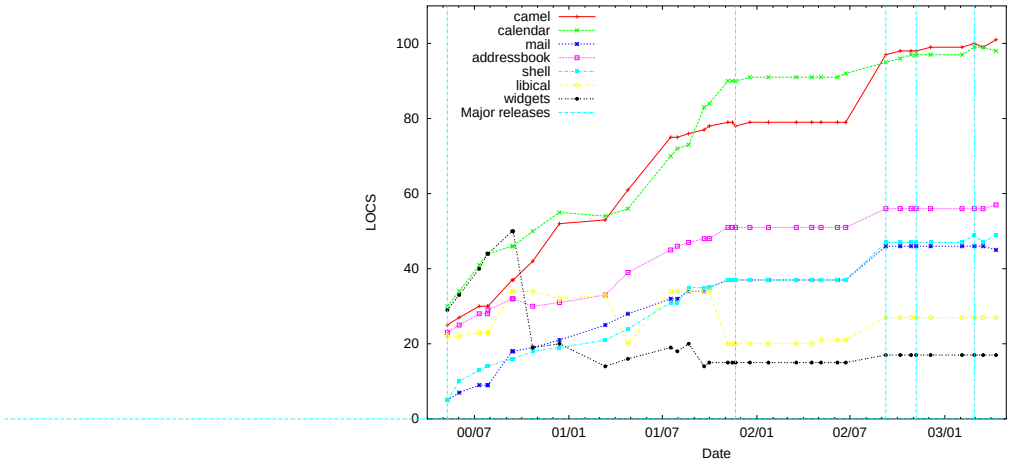


Figure 10: LOCS in selected modules, per version

Other interesting questions are: Do contributors tend to concentrate in one module? How many core contributors does a given module have? Table 6 shows that information for the five most active modules of Evolution. In order to account only for people who are still active in the development, this table only shows data related to MRs which happened in 2002. It is not surprising to see that one or two contributors are responsible for at least two thirds of the MRs in each module.

Finally, how well do modules isolate developers from the complexity of other modules? One potential way to measure this dependency is to analyze the number of codeMRs that require changes in more than one module. Figure 11 shows a compelling story: only 3% of the MRs include more than one module. Further analysis of the changes is required to determine what is the proportion of changes that were actual code changes compared to changes in comments (such as the change in license, described in section 4.2.1).

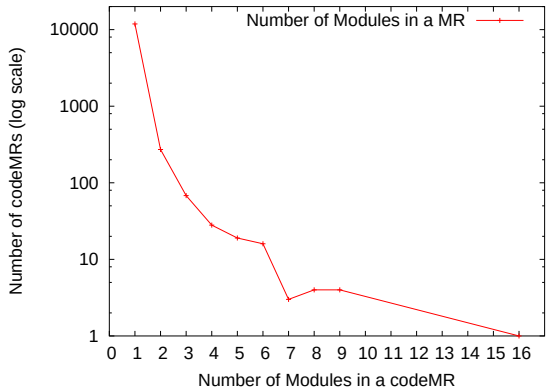


Figure 11: Number of different modules that appear in a codeMRs. The proportion of codeMRs that involve more than one module is very small (3%).

Mod	Progs	Id	Prop	Acc
shell	17	ettore	0.65	0.65
		danw	0.11	0.76
		toshok	0.05	0.81
		clahey	0.04	0.84
		zucchi	0.03	0.87
mail	19	fejj	0.52	0.52
		rodo	0.13	0.65
		zucchi	0.12	0.77
		ettore	0.07	0.83
		danw	0.06	0.89
calendar	17	jpr	0.40	0.40
		rodrigo	0.32	0.72
		ettore	0.07	0.79
		danw	0.06	0.85
		damon	0.03	0.88
camel	9	fejj	0.66	0.66
		zucchi	0.25	0.91
		danw	0.03	0.94
		peterw	0.03	0.97
		ettore	0.01	0.99
addressbook	19	toshok	0.57	0.57
		clahey	0.13	0.70
		ettore	0.09	0.79
		danw	0.07	0.87
		fejj	0.03	0.90

Table 6: Top 5 programmers of some the most active modules during 2002. The first column shows the name of the module, the second shows the total number of programmers who contributed to it in that year, the third shows the userid of the top 5 programmers and the proportion of their MRs with respect to the total during the year.

5 Further observations

The results described in this paper show that the method described in section 2 can be applied to recover the evolution of a software project where the amount of software trails is significant. Several observations can be made about this experience.

- One software trail does not tell the whole story. It is paramount to cross-reference software trails to really understand what they mean in the evolution of the project. For example, the size of software releases in Evolution has been growing in linear fashion, while the growth in the size of the source code is relatively flat; also, many developers have been participating in the project, but most of them with very few contributions.

- Schema definition. The schema used in this study kept changing, in part due to the incorporation of new trails, and in part because new information and relations kept being discovered. It is expected that, as this type of analysis becomes more pervasive, standard schemas can be developed. This will have 2 advantages: a) it will promote the creation of tools that gather software trails, extend them, and analyze them; and b) the evolutionist will better understand the nature and interrelation of the available trails before starting to do her work.

- One of the main challenges of analyzing software trails is that many of them are informal in nature. For example, email messages contain a large amount of information pertaining to the way the project has evolved, but they are difficult to analyze in an automatic fashion. Correlating different trails is also an error prone task, in which heuristics have to be developed and tested. It might be the case that a heuristic performs differently in different projects.

- Information overload and the need for analysis and visualization tools. The amount of available information makes it indispensable to use tools that can filter it and visualize it. Again, as schemas are standardized, different research teams could provide different tools that specialize in mining and visualizing certain types of trails. In this paper, SQL was chosen because it provides a sophisticated query language (further extended in postgreSQL with its support for regular expressions in the `where` clause). SQL was very helpful in filtering and tabulating information, that could then be plotted (our research team has since developed a tool to automatically create many of the plots displayed herein using SVG using the Web as its interface). It is also interesting that Evolution itself proved very useful in analyzing the Evolution mailing lists, given that it provides a powerful query language for email messages.

- Quality of software trails. It is important to state that not all development teams generate “good” software trails. In the experience of the author, there is a point in a software project in which software trails start to “mature” and this point is likely a correlation of the success of the project, the level of interaction that developers have to have, and their maturity, and in the case of commercial projects, the influence of management. For instance, there is very little information about Evolution when only one developer was

contributing to it, but as the developers grew in number (and became more experienced) their trails improved in quality. The Free Software Foundation has an important effect in the quality of trails, as it publishes a collection of guidelines that free software developers should follow.

6 Conclusions and Future Work

This paper demonstrated a methodology to recover the evolution of a software project using its software trails. Software trails, such as version releases, version control logs and mailing lists were used to recover the evolution of Ximian Evolution, a free (as defined by the GPL) mail client for Unix. The analysis of these software trails allowed the discovery of interesting facts about the history of the project: its growth, the interaction between its contributors, the frequency and size of the contributions, and important milestones in its development.

There are several potential avenues for future research. One of them is to create tools that analyze and enhance the facts extracted. For example, CVS's MRs can be analyzed in an attempt to guess the type of modification that the developer intended: a comment, a bug fix, a new feature, or refactoring, for example. This will allow the evolutionist to quickly categorize changes and concentrate on those of interest.

Another area of research is the visualization of this information. As the project grows older, its trails grow in number. It is necessary to create tools that analyze and display the gathered facts to the user and allow its visualization in a highly dynamic manner. Metrics are also an important area of research. It is needed to quantify the information extracted from software trails, so it can be compared with other software projects. For example, how can the "disjointness" of contributors of different modules to different software projects be quantified and compared?

Finally, studies on other software projects (similar to the one done in this paper) are needed. These studies will provide information necessary to better understand the characterization of software trails. Furthermore, these studies will allow researchers to compare the evolution of different software projects; and to a certain extend some of the practices used by their corresponding development teams.

Acknowledgments

This research has been supported by the National Sciences and Engineering Research Council of Canada, and the British Columbia Advanced Systems Institute. The author would like to thank Audris Mockus (co-author of softChange) for his invaluable help in a preliminary analysis of Evolution and the reviewers of this paper for their helpful comments.

References

[dI99a] Miguel de Icaza. Writing a GNOME mail client. http://mail.gnome.org/archives/gnome-announce-list/1999-April/msg00029.html, April 1999.

[dI99b] Miguel de Icaza. Writing a GNOME mail client. http://canvas.gnome.org:65348/mailing-lists/archives/gnome-mailer-list/1999-April/0018.shtml, April 1999.

[dI00] Miguel de Icaza. G Apps Lib 0.1 is out. http://mail.gnome.org/archives/gnome-announce-list/2000-October/msg00005.html, October 2000.

[Ger02] Daniel M. German. The evolution of the GNOME Project. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*, May 2002.

[GM03] Daniel M. German and Audris Mockus. Automating the Measurement of Open Source Projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, May 2003.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proc. of the 2000 Intl. Conference on Software Maintenance*, pages 131–142, 2000.

[Gui99] Bertrand Guiheneuf. Gnome Mail clients (Re: Is Balsa alive?). http://mail.gnome.org/archives/gnome-devel-list/1999-April/msg00042.html, April 1999.

[Gui00] Bertrand Guiheneuf. Candidate (Bertrand Guiheneuf). http://mail.gnome.org/archives/foundation-announce/2000-October/msg00009.html, Oct 2000.

[Jon02] Paul Jones. Brooks' law and open source: The more the merrier? does the open source development method defy the adage about cooks in the kitchen? IBM developerWorks, August 20, 2002.

[LT00] Josh Lerner and Jean Triole. The Simple Economics of Open Source. Working Paper 7600, National Bureau of Economic Research, March 2000.

[MFH02] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[Per01] Ettore Perazzoli. Ximian Evolution: The GNOME Groupware Suite. http://developer.ximian.com/articles/ whitepapers/evolution/, 2001.

Meta-Model and Model Co-evolution within the 3D Software Space

Jean-Marie Favre

Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr/~jmfavre

Abstract

Software evolution-in-the-large is a challenging issue. While most research work concentrates on the evolution of "programs", large scale software evolution should be driven by much higher levels of abstraction. Software architecture is an example of such abstraction. The notion of co-evolution between architecture and implementation has been identified and studied recently. This paper claims that other abstraction dimensions should also be taken into account, leading to what we call the 3D software space. This conceptual framework is used to reason about evolution-in-the-large phenomena occurring in industry. The meta dimension, which constitutes the core of the MDA approach, is considered as fundamental. This paper makes the distinction between appliware and metaware and put the lights on meta-model and model co-evolution. Conversely to the MDA approach which makes the implicit assumption that meta-models are neat, stable and standardized, in this paper meta-models are considered as complex evolving software artefacts that are most often recovered from existing metaware tools rather than engineered from scratch. In fact, we identified the notion of meta-model and model co-evolution in the context of the evolution of a multi-million LOC component-based software developed by one of the largest software companies in Europe.

1. Introduction

Understanding very large software products is a major issue. Understanding their *evolution* is even more difficult since many factors influence software evolution [1][2]. This paper concentrates on *evolution-in-the-large* which is quite different from *evolution-in-the-small*, that is evolution of small programs over rather short periods of time (a few months or years). Evolution-in-the-large is about the evolution of multi-million LOC software over decades.

Evolution-in-the-large is indeed a very complex issue. Considering software evolution at the level of statements and functions is clearly not enough. A much higher level of abstraction is required.

1.1. Architecture/Implementation co-evolution

Software architecture shoutld clearly play a central role in the evolution since it provides a abstraction. However, making explicit the architecture of software is not easy in practice. Architectural Description Languages (ADLs) failed to find their path to industry, in part because of their poor support for software evolution. Software industry is still code-centric. Most of the time the architecture is implicit. To cope with this problem, a increasing amount of research work focuses on architecture recovery and architecture evolution (e.g. [3][4][5][6][7]). Recently the concept of *architecture and implementation co-evolution* has been identified by various authors (e.g.[9][10][11]). Architecture and implementation are two levels of abstraction. They are both subject to evolution. Since they are linked they should ideally evolved in a synchronized way to avoid the so called *architectural drift* and *architectural erosion*. Maintaining some kind of architectural description is useful to ease software understanding, but another important idea is that the architectural description should allow to *drive* or at least to *constrain* the evolution of implementation.

Figure 1 provides a very intuitive view of the relationship between the two abstraction levels. Modifying an entity at a level of abstraction can both have an impact at this same level, but also at the lower (or higher) level of abstraction. In fact *whenever two entities are linked by a relation, changing one entity may have some impact on the other one*.

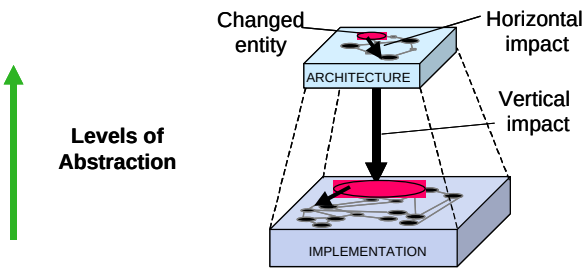


Figure 1. Vertical vs. horizontal impacts

Horizontal impacts refer to impacts within a given abstraction level (i.e. modifying a function may imply to upgrade other functions). By contrast, *vertical impacts* cross abstraction levels. For instance modifying a function may have an *upwards impact* on an architectural component. Removing a dependency between two components may have many *downwards impacts* on implementation entities. The nature of the impact obviously depends on the nature of the entities and the nature of the relation. The same is true for the action to be taken after such an impact is detected.

Horizontal consistency must usually be ensured. For example updating the impacted functions is usually considered of paramount importance to ensure a consistent behaviour of the implementation. *Vertical consistency* could be much more loose leading to a large range of co-evolution policies. For instance, upgrade could sometimes be deferred, taking the risk of a temporary inconsistency and deviation [9]. With no suitable policy, these inconsistencies usually lead to irremediable erosion and the very common situations where architectural artefacts are no longer updated. In fact, the horizontal dimension has been studied for long leading for instance to research on impact analysis either at the implementation level or at the architectural level (e.g. [8]). The term co-evolution is usually used vertically when the evolution of two levels of abstraction can be asynchronous.

1.2. Co-evolution along other dimensions

We discovered over the last years the existence of similar phenomena along other abstraction dimensions. Co-evolution is indeed a very common. This paper introduces two other abstract dimensions and structure the set of software artefact as a *3D software space*. This *conceptual framework* is very useful in the systematic identification of co-evolution processes.

In particular the main objective of this paper is to put the light on meta-model and model co-evolution. Though this phenomenon occurs along the meta-dimension popularized by the UML and MDA standards [12][13][14], the concepts presented in this paper are by no means restricted to software developed using modern techniques such as Model Driven Engineering (MDE) [15][16][17][19] (In this paper MDA refers to the OMG standard while MDE refer to the approach which is more general). This paper shows that meta-model and model co-evolution actually occurs with current and legacy industrial practices.

In fact, the MDA approach assumes that meta-models are neat, stable and standardized. In this paper on the contrary meta-models are considered as complex evolving software artefacts that are most often recovered from existing tools rather than engineered from scratch. Simply put, while the MDA and meta-related technologies are typically oriented towards forward engineering, this paper considers meta-models in the context of reverse engineering.

1.3. Background

In fact, the concepts presented in this paper results from our experience in various industrial settings. In particular, we first identified the meta-model and model co-evolution phenomenon in the context of a collaboration with Dassault Systèmes (DS). DS is the world leader in CAD/CAM and one of the largest software company in Europe. Our collaboration with this company lasted 7 years. During this period we dealt with many issues related with software evolution including configuration management, software architecture and reverse engineering [20][21]. We gained a lot of expertise about evolution-in-the-large. In fact, DS faces a wide range of issues related with very large scale software evolution. More than 1200 software developers work at the same time on the same software product leading to tremendous requirements in configuration management [20]. DS evolves a huge software, CATIA, which is made of more than 70 000 classes, 800 frameworks, and 3000 DLLs. This leads to tremendous requirements on software architecture [22][23]. In fact, Dassault Systèmes is with Microsoft one of the pioneer of component-based software development. In the mid 90's DS started to design and develop an in-house component-technology called the OM and at the same time this technology was used to develop CATIA components [21]. It will be shown in this paper that this is in fact a typical example of meta-model and model co-evolution.

The rest of the paper is structured as following. In Section 2 a simplified explanation of what is meta-model and model co-evolution is provided. Section 3 gives an overview of the conceptual framework referred as the "3D software space". The first dimension, called the meta-dimension, is presented in section 4. Section 5 introduces the "product engineering dimension". Section 6 describes the third dimension, the "representation dimension". Section 7 shows how evolution interacts with this 3D space. Section 8 gives examples of observable meta-model/model co-evolution phenomenon. Finally section 9 concludes the paper.

2. Language/Program/Tool co-evolution

Meta-related notions could be difficult to grasp at the first sight, especially when applied in complex industrial contexts. Before to introduce the 3D software space in a systematic way, let us introduce the issue in terms of much more narrow but much more intuitive concepts. For the sake

of clarity, the illustrating problem is based on well-known programming-in-the-small concepts. Let us consider three kind of entities: *programs*, (programming) *languages*, and (language-dependent) *tools* (e.g. compilers). Three kinds of relation can be considered: (1) language/program, (2) tool/ program, (3) tool/language. All these relations leads to co-evolution issues as suggested below.

2.1. Language / program co-evolution

A program is closely linked with the language it is written in. It is well known that a change in the language could have a strong (downwards) impact on programs. This leads to a wide range of upgrade and migration strategies. When a new version of the language is made available, developers have first to determine which programs are impacted by the language modification. They could then decided to upgrade impacted programs to ensure consistency with the new language. Alternatively they could delay the changes and continue to use the old language version. They might to that for impacted programs while using the new version to develop new programs. This common situation reveals *language and program co-evolution*. This phenomenon is usually not made explicit.

2.2. Tool / program co-evolution

Language dependent tools such as interpreters, compilers also have a great influence on programs. The availability of such *primary tools* is of fundamental importance in practice. *Secondary tools* such as documentation generators, metric and profiling tools are also very appreciated in industrial settings, in particular in the context of quality insurance processes. Developers may have to adapt their programs to use a particular tool. This could be to take advantages of a feature (e.g. adding tags in comments to use a documentation generator like javadoc). Sometimes this is to avoid a bug in the tool (e.g. removing the use of C++ templates in a program because the compiler on a given platform do not handle it properly). Tool evolution leads to *tool and program co-evolution* issues.

2.3. Language / tool co-evolution.

Languages are abstractions. Tools are concrete implementations supporting these languages. A change in a language specification could have many impacts on many tools. This leads to *language and tool co-evolution*. Upgrading primary tools such as compiler and interpreters is usually done first to get synchronized with the language. By contrast, the modification of secondary tools such as browsers are often delayed. Deviation from the language specification is common for such tools.

2.4. Discussion

Summing up, programs, languages and tools are linked by three kinds of relation. Each relation give rises to co-evolution issues. At this point the reader might not be convinced by the relevance of these issues. A few observations must be made to relate the discussion to the context of evolution-in-the-large.

One might argue that changes in languages and tools are seldom when compared to changes in programs. This is quite true but remember that the time scale considered in this paper is expressed in terms of years or decades, not weeks or months. Everything evolve in large companies. Software architecture evolve. Tools evolve. Languages evolve. While small projects apply versionning concepts to programs, large scale projects also deal with *language versionning* and *tool versionning*. Languages and tools are consider as actual part of the software, which is very true.

It is also very important to stress that while the term “language” might evoke to researchers a neat, standardized and stable thing, “real-life” is industry is often quite different. To a large extend, today software industry largely relies on many ill-defined, proprietary and unstable languages. The same is true for tools. The goal of this paper is to model industrial practices as they are.

Taking into account legacy software and legacy practices is an important requirements. In the early decades of computer science, many large companies developed in-house programming languages and made them evolved incrementally while developing programs at the same time. These are real-life language/program co-evolution scenarios in which language evolution is driven by the problems encountered in developing programs.

Note that in this context the language remains most of the time implicit; there is no explicit description of the language. As reported in [24], the exact grammar of programming languages such as COBOL variants is often unknown and has to be recovered from tools. This leads to grammar reverse engineering [24]. Many legacy and proprietary languages have actually evolved mostly through patches in compilers or interpreters to add or remove special features. Many language definitions, if ever existed, deviated from tool evolution and became inconsistent. This is *language erosion*, a real-life example of language/tool co-evolution.

One might argue that this time is over, that modern languages and tools are much more stable and well engineered. This is unfortunately not true. In the last years the boom in internet-based technologies gives rise to the apparition of a very large number of languages such as scripting languages with internet-based features. These languages are more than ever linked with tool evolution such as web servers. Evolution is rapid and chaotic.

Languages are ill-defined and unstable. The future could be soon populated by legacy web applications raising serious language/program co-evolution issues.

Modelling languages also evolve. This includes in particular the continuous evolution the UML standard over the years. Just like other languages, UML greatly evolves (e.g. UML 1.0 to 1.5 and now 2.0) and presents symptoms of language extension and language contraction. Quoting Warmer about UML 2.0: "*The evolution of UML is absolutely required to make sure that UML will stay up to date with the latest developments in the software industry. The direction taken is guided by the user community, but it requires a big effort* "[26]. This evolution is accompanied by strong co-evolution issues, not only with respect to the large amount of UML diagrams that, but also with respect to the production of large amount of commercial CASE tools. In practice these tools are permanently out of sync. They often deviate from the standard and subtle or most often in important ways.

Component-based development is also getting very popular and component technologies such as COM, .NET or EJB are largely used. These technologies are based on "component models" that defines new concepts and rules that must be followed when developing component-based programs. Though no specific syntax is provided component models could be seen as virtual languages [25]. These languages are often ill-defined, unstable and they greatly evolve because the notion of component is in constant evolution. Once again, this leads to language/ program co-evolution issues. This point is illustrated in Section 7 using Dassault Systèmes as a case study.

Co-evolution is a general phenomenon. It can be applied to requirements, modelling languages, software architecture, etc. The term program is therefore inadequate. Since languages do not even need to be explicit, the same apply to the term language. We use instead more general concepts: models, meta-models and metaware. Roughly put programming language are special cases of meta-models, programs are special cases of models, and programming tools are special case of metaware tools. These concepts and their relationships are described in the remainder of this paper in a systematic way using the 3D software space.

3. The 3D software space

The complex nature of software can be represented as a 3D software product space as depicted in Figure 2. Figures 6, 8 and 9 on the next pages zoom on this space and illustrate its content by means of simple examples. The reader is invited to browse these figures paper to get an overall idea of the content of this space.

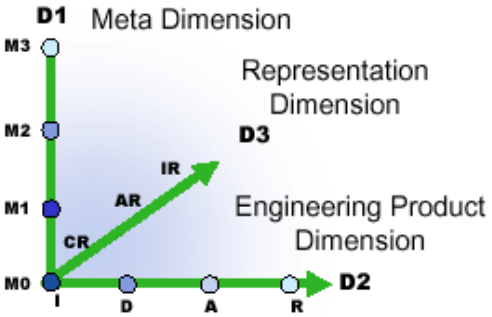
Each dimension corresponds to a different kind of abstraction. All dimensions are orthogonal as it will be show in the next sections.

D1: The meta-dimension. This dimension constitutes the core of the MDA standard. Four levels are distinguished: instances, models, meta-models and meta-meta models. Programs are at the model level (M1), programming-language at the meta-level (M2). The instance level (M0) and the meta-meta level (M3) are included for the sake of completeness. Meta-model/model co-evolution is linked to this dimension.

D2: The engineering product dimension. This dimension aims to structure the software according to each phase in the software life-cycle. It helps for instance to make the distinction between requirement descriptions, architectural documents, and implementation artefacts. Architecture/implementation co-evolution phenomenum is linked to this dimension.

D3: The representation dimension. There are many different ways to represent a given entity ranging from very abstract representations to concrete ones. For instance a programmer might have a mental image of a software architecture. The architecture might also be represented as a boxes-and-arrows graph or as an graph stored in an XML file. Concrete representations heavily depends on the tools that manipulate it. It will be shown that language/tool co-evolution is linked to this dimension.

Each point of this space is represented in the subsequent figures by a cell because it corresponds to a class of software artefacts. Note that the name of each class is conveniently formed by appending the corresponding coordinates in reverse order D3-D2-D1. For instance CR-D-M1 reads "*C*oncrete *R*epresentation of *D*esign *M*odels" and AR-A-M2 stands for *A*bstract *R*epresentation of *A*rchitectural *M*eta-*M*odels.



D1	Meta
M3	Meta-Meta-Model
M2	Meta-Model
M1	Model
M0	Instance

D2	Engineering
R	Requirements
A	Architecture
D	Design
I	Implementation

D3	Representation
IR	Implicit repr.
AR	Abstract repr.
CR	Concrete repr.

Figure 2. The 3D Software Space

In fact, the density of the space is far from uniform. Almost all software artefacts are stuck near to the origin, where programs are. Industry is still code-centric. To illustrate this phenomenon, gray scales are used in most figures. Moreover each dimension will be described as a *pyramid* in the next sections (see Figures 3, 5 and 7). Since the 3 dimensions correspond to a different kind of abstraction, the pyramid structure is well suited to model reality. The *width* of the pyramids represents alternatives or variants, while the *depth* represents the many software entities that constitute each alternative.

4. The meta dimension (D1)

The meta-dimension is surely the most difficult dimension to grasp but it is also the most powerful. It constitutes the core of this paper. In this paper the MDA standard is taken as a reference.

4.1. The meta-pyramid (D1)

The *meta-pyramid* is depicted in Figure 3. A few examples are provided for each level. More examples can be found in Figure 5 in which the meta dimension is represented horizontally.

The most obvious level within the meta pyramid is the model level (**M1**), so let us start by this level. This is the level where regular programs are. This level corresponds to what could be called *appliware*. Entities at this level depends on the particular application domain considered (e.g. banking, nuclear plant design, etc.). For instance the concepts of "account" and "client" might be a part of the banking model, while the concept of "reactor" might be part of the nuclear plant model.

The model level is used to manage the set of all possible real-world situations which are represented at the instance level (**M0**). For instance "Tom" might be a client that owns two accounts "a4099" and "a2394" with a respective balance of $800 and $2000. A point at the instance level describes a particular state of a software at a particular point in time. It corresponds to a program state. Program execution indeed corresponds to the evolution of this state.

Metaware by contrast is independent from application domains. The meta-model level (**M2**) is used to manage the production of software applications. It should describes therefore all software engineering concepts such as "classes", "methods", but also "modules", "frameworks", "configuration", "dynamic libraries", etc. In simple words meta-models capture the set of the concepts used to develop software.

On the top of the pyramid, the meta-meta-model level (**M3**) describes how the meta-models should be described and managed. For instance the MDA standard proposes to use the Meta Object Facilities (MOF) [31]. Simply put, the MOF is a self descriptive subset of UML that allows to describes arbitrary software meta-models (not only the UML meta-model). The MOF is to meta-models what the BNF is to grammars, a standardized way to represent them. Though the meta-meta level is important, this paper concentrates on the meta level for the sake of simplicity. Similarly the term metaware will be used for both level M2 and M3, to avoid introducing the term metametaware.

4.2. Software = Appliware + Metaware

The meta pyramid depicts the realm of software. The next sections will help in making this dimension more concrete, but what is important to understand at this point is that at each level M1, M2, M3 there is some piece of software. Software at the level n+1 is used to build and control software at the level n. Metaware is application-independent software that help producing software applications, that is appliware. A compiler is an example of metaware tools. It is based on the meta-model of the source programming language (e.g. the java meta-model for the javac compiler).

We found distinction between metaware and appliware very important to understand industrial practices. For

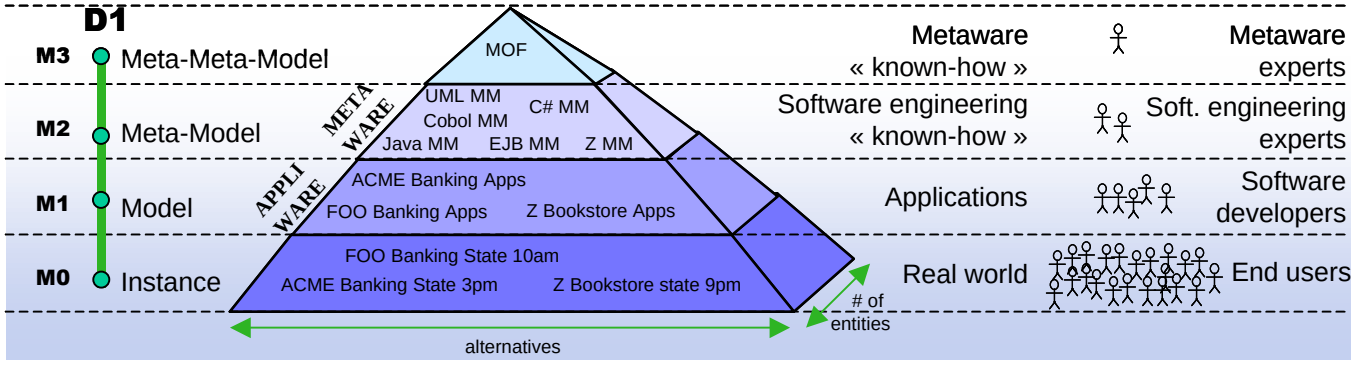


Figure 3. D1: the meta pyramid

Figure 4. D1: the meta-actor pyramid

instance, during our 7-years collaboration with Dassault Systèmes we always stayed at the meta-level. We know much about DS' metaware. By contrast, we never saw DS'appliware [20]. In fact, we never saw a single line of application code in 7 years. Metaware and appliware are distinct parts of software. *Software is metaware plus appliware*. Metaware is software that manage and control software. As it will be shown in the next sections *meta-models are just the visible part of metaware*. Appliware is software that represents applications. Software covers the three higher levels of the pyramid (M1,M2,M3). Level M1 corresponds to appliware, the world of applications. Level M2 and M3 corresponds to metaware. Note that the lower level M0 is about particular states of software execution, which is usually not considered as software.

4.3. The meta/actor pyramid

In fact, one good way to grasp the distinction between the various levels in the meta pyramid is to consider the actors involved at each level. This leads to the *meta/actor pyramid* depicted in Figure 4. The goal of the actors working at the level n+1 is to help actors at the level n to do their job by providing them software. As shown below, raising from a level to the next one decreases the number of people concerned by various orders of magnitude.

End-users are the instance level actors (**M0**). They interact with software applications. They *use* appliware to perform their job. Billions of people around the planet are direct or indirect users of software applications. About 500 000 people use CATIA applications to do their jobs.

Appliware developers are developers of software applications (**M1**). They *produce* appliware for the benefit of M0 actors. They *use* metaware tools to do their job. The number of developers is estimated to be about 6 millions [16]. The great majority of them work at the M1 level. Within the context of DS, more than 1200 software engineers work on developing CATIA applications.

Metaware developers are the meta level actors (**M2**). They *produce* metaware for the benefit of the M1 actors. In practice, each large company includes a separated group of people that define processes, work on quality and build/ integrate tools to manage applications development. They are referred as "know-how providers" in [16]. Their number is estimated to be around 100 000 for the globe [16]. In the context of Dassault Systèmes, these tasks are handled by the Tool Support Team (TST) [20]. This team, made of a few dozens of people, work on metaware and build in-house tools to support CATIA development.

At the higher level of the pyramid, the number of people dealing with meta-meta models (**M3**) is obviously even lower. It might be something around a few thousands for the whole planet because most of the time the meta-meta level is not expressed. However, this could change in the future in particular if the MDA and MOF find their place in industry.

5. The engineering dimension (D2)

Though it is an over-simplification, the waterfall lifecycle clearly shows that software products are not only made of programs: software also includes requirement specifications, global design, detailed design, etc. This leads to dimension D2.

5.1. The engineering pyramid (D2)

Dimension D2 aims at structuring software artefacts following the a very basic engineering process. For the sake of simplicity, only 4 levels are distinguished in the context of this paper, namely the requirement level (**R**), the architectural level (**A**), the design level (**D**), and the implementation level (**I**). This view is obviously a huge simplification of the software realm. The purpose is just to cross this dimension with the other ones, so the model must be simple enough to get understandable results. The next figure shows the *engineering product pyramid*. Each level is illustrated by different examples. More detailed examples are provided in next sections.

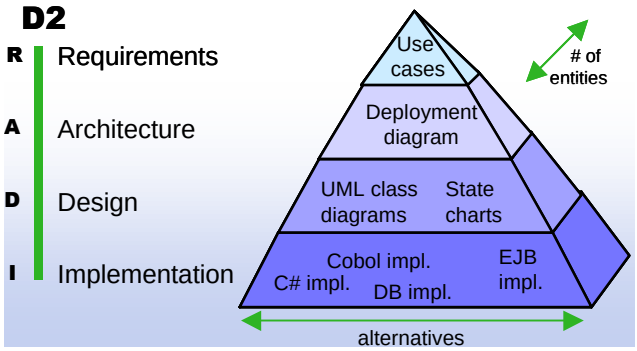


Figure 5. D2: the engineering pyramid

5.2. The engineering / actor pyramid (D2)

Software lifecycles like the waterfall model not only help in identifying the variety of software artefacts. They also make it clear that various actors with different skills are involved in the production of software. Though the engineering/actor pyramid is not depicted, each level of the D2 pyramid involves different actors: requirement engineers, software architects, designers, and developers. A typical project is formed by many developers, but only few architects.

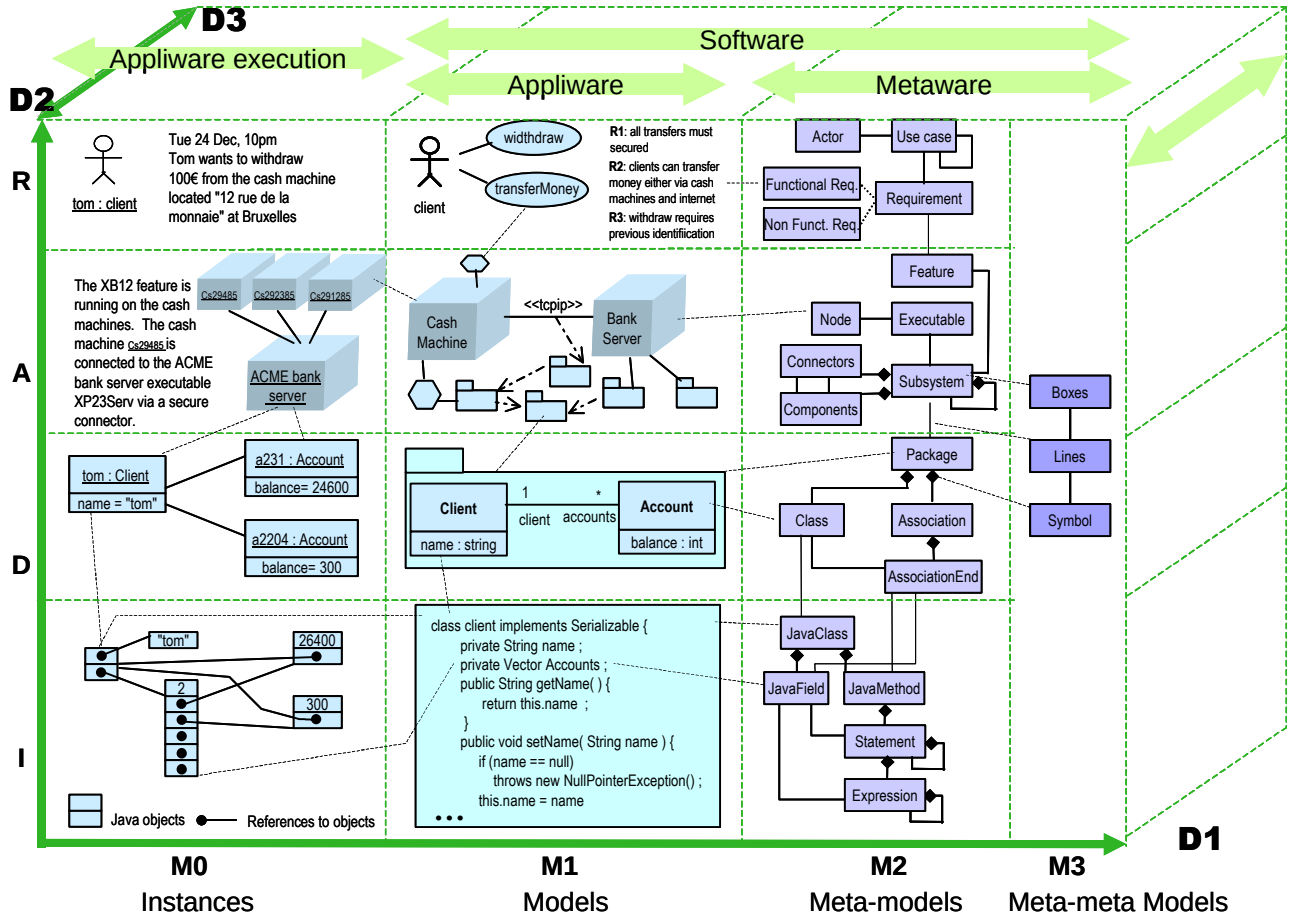


Figure 6. Crossing the meta-dimension (D1) with the engineering dimension (D2)

5.3. Crossing D1 and D2

Confusing the meta dimension and the engineering dimension is quite common, especially when considering the higher levels of abstractions. These two dimensions are however truly orthogonal. For instance, there are architectural models (A-M1), architectural meta-models (A-M2), design model (D-M1), design meta-models (D-M2) and so on. Figure 6 illustrates this property by means of a very simplified yet consistent example. The banking application of the virtual ACME company is considered.

In fact, Figure 6 is centred around column M1 (models) because the engineering process we speak about is defined on models. The reading of the figure should therefore start from that column: other columns are derived from M1. Column M2 acts as the key for the concepts instantiated in column M1. That is, column M2 describes the various meta-models in an informal way, using a UML-like class diagram notation. In fact, boxes, lines and symbols are used to described the meta-models (see column M3). On the opposite side column I describes a particular state of the real world as modelled for the purpose of the ACME banking software. The reader is invited to carefully read Figure 6 which is expected to provided enough intuitive material to grasp the idea.

5.4. Cross-links between levels and co-evolution

All the concepts presented in Figure 6 are connected. However, for the sake of readability only a few links have been drawn between the different cells. The nature of the cross-links depends on the dimension considered.

Vertical cross-links correspond to tracability links between the artefacts produced during the software life-cycle. At the level of meta-model tracability links can just be modelled as regular associations. We first applied this approach to link software architecture and source code in the context of java beans [18], and then in the context of CATIA [22]. Maintaining these links is fundamental to support co-evolution along the engineering dimension (D2), and in particular architecture/implementation(A/I) co-evolution. Tracability between models is considered as an important issue in the MDA approach [14].

Horizontal cross links are different in nature: they relate an entity to its model and conversely a model to its instances. The modelling of these cross-links constitutes the basis to support co-evolution along the meta-dimension and in particular meta-model/model (M2/M1) co-evolution.

6. The representation dimension (D3)

The reader might have noticed that all the examples in Figure 6, do not correspond to the same kind of representations. In fact one can imagine many other alternative representations for each cell. What is needed is an additional dimension to represent these variations. This leads to D3, the "representation dimension" (D3).

6.1. The representation pyramid (D3)

It is important to recognize that a single piece of information can be represented in many different ways ranging from implicit representations to very concrete ones. The fact that a piece of data is not explicitly represented as a sequence of bits does not mean that it does not exist. For instance, most of the time, software architecture is not explicitly represented. Software architects maintains some mental images and this might be enough. To communicate, box-and-arrows diagrams are often used. Other information is also represented by means of natural languages. Or it is simply part of the "implicit knowledge" of a given company. These kind of representations are obviously not adapted to automated processing. Very concrete representations are required when tools support is needed.

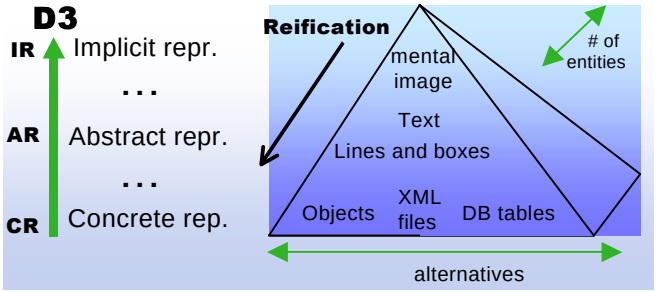


Figure 7. D3: the representation pyramid

The figure above depicts the *representation pyramid*. Though there is a continuum of abstraction levels, only three levels are named for the sake of simplicity: implicit representation (**IR**), abstract representation (**AR**) and concrete representation (**CR**). The lowest level is oriented towards tool processing, while the highest level represents implicit knowledge. Though the actor pyramid is not depicted human actors would be at the top of the pyramid while the many tools that process concrete representations would be at the lower level.

The shape of the pyramid is justified by the fact that a very large set of representation techniques can be used to represent a particular software entity. This includes for instance graph of objects in memory, tuples in a database, XML files, etc. Concrete representations greatly vary depending on the purpose of the tool considered. For instance a compiler, a syntax editor and a test coverage tool might represent the same program with very different internal structures. That's just a matter of concrete representation.

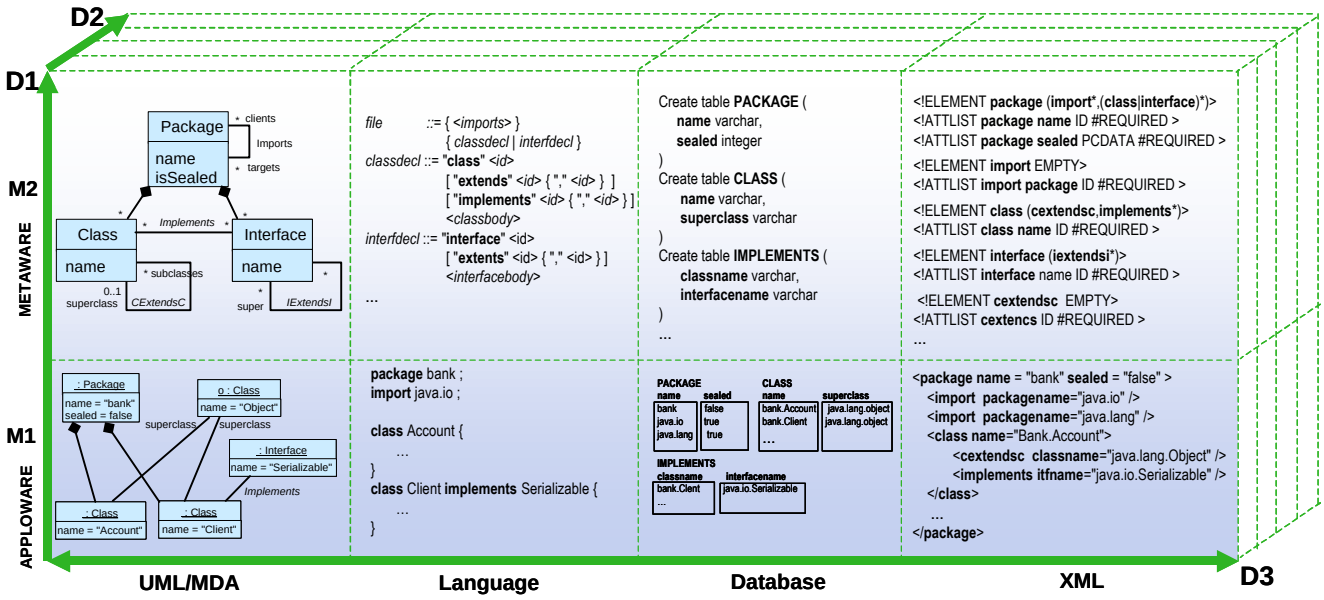


Figure 8. D1+D3: Alternatives representations of a Java program (I-M) and a Java meta-model (I-MM)

6.2. Crossing D3 and D1-D2

Though this might not be obvious, the representation dimension D3 is orthogonal both to the engineering dimension D2 and the meta dimension D1. Due to space constraint only small slices of the space could be provided.

Figure 8 on the previous page illustrates the variety of concrete representations both for a java program (M1) and a java meta-model (M2). At both levels, the *same* information is represented in different ways. Since all alternative representations are more or less at the same level of abstraction with respect to D3, this example illustrates the width of the representation pyramid but not its height.

The continuum from implicit representations to very concrete ones is illustrated in Figure 9 for the metaware column (M2). This small slice of the software space illustrates in particular the notion of *conceptual meta-model*, *specification meta-model* and *implementation meta-model* as well as *metaware tool*. Due to limitation space, Figure 9 illustrates the height but not the width of D3 pyramid: only one possible representation is selected when going down from one level to the next one.

A very simple example of meta-model (a small subset of the java language) has been selected for the sake of clarity. In practice the approach has to be applied on much more complex meta-models, such as proprietary architectural meta-model (e.g. [20][27])[1].

The implicit knowledge a java programmer could have about the java language would fit on the top of the pyramid. A programmer might know for instance that java provides simple inheritance between “classes”, yet a “class” may implement multiple “interfaces”. This is the implicit part of the metaware. Just implicit knowledge.

At the other extremity of the spectrum, we found very concrete metaware artefacts managed by metaware tools. In the case of a programming language, metaware tools include all tools that parse, analyse, interpret, and manipulate programs: interpreters, compilers, browsers, etc. Obviously, each tool have an embedded knowledge of the language it manipulates. This knowledge is represented somehow in the code of the tool. This observation is consistent with what Lammel and Verhoef report in [24].

As show in Figure 9, the role of meta-models is central to metaware. *Meta-models makes the bridge between concrete metaware items and informal metaware knowledge*. Meta-models constitute the conceptual part of the metaware. In the academic world, the term meta-level usually evokes this part, because meta-models are neat abstractions to reason about. Unfortunately our experience

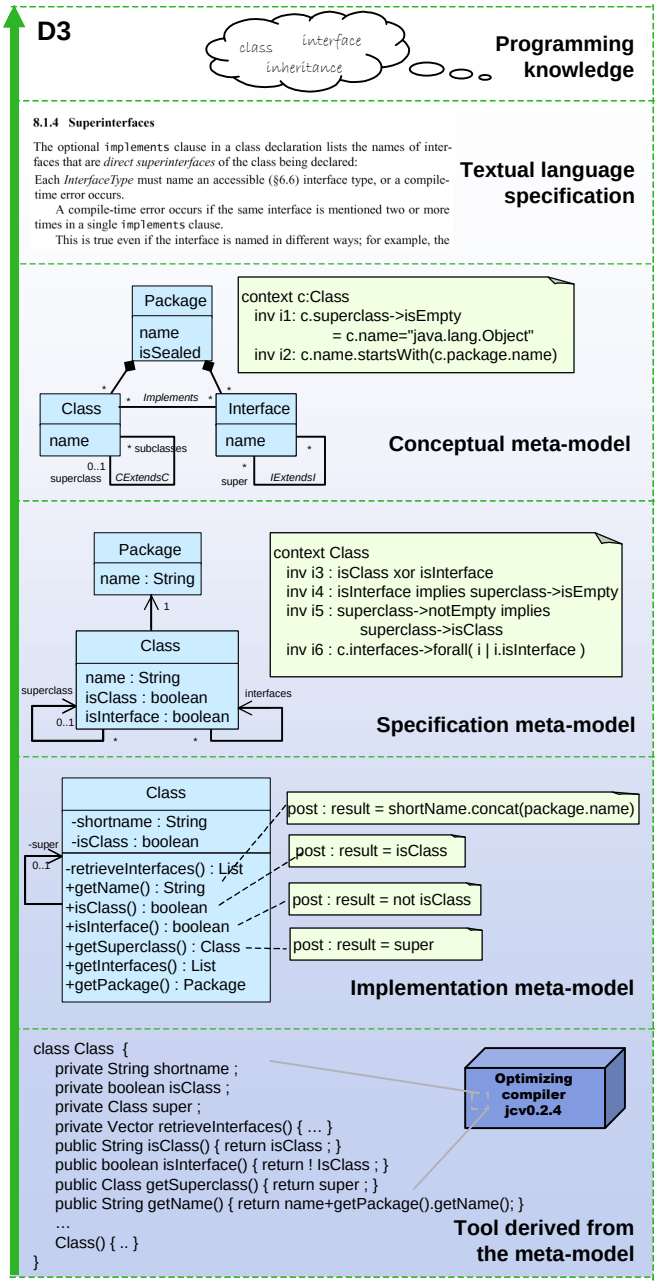


Figure 9. D3: Metaware

shows that this part is often missing in industry. Though the term meta-level might not evoke anything in large companies, metaware *does* exist. It takes however the form of software development tools. Many of these tools are complex and often proprietary [20]. Recovering meta-models is important in particular since meta-models capture the application independent part of the company know-how [32].

Finally it should be noted that the distinction made between conceptual meta-model, specification meta-model and implementation meta-model is indeed based on the

1. For a full understanding of the various steps described in Figure 9 it is assumed that the reader has both a knowledge of java and the understanding of the refinement process as described in [30]. .

application of the principles introduced by Fowler [30]. These levels are usually applied on models to develop appliware through successive refinement. We found however these concepts very useful to categorize existing meta-models. In fact, reading Figure 9 from bottom to top clearly suggests a forward engineering process, while reading the figure from top to bottom leads to a reverse engineering process. While D2 is centred around appliware engineering, D3 is centred around *metaware engineering*.

7. Evolution: entering the fourth dimension

As depicted in Figure 10, evolution can be introduced in the conceptual framework by adding a fourth orthogonal dimension representing time.

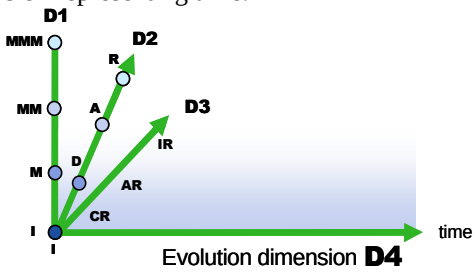


Figure 10. Entering the fourth dimension

This modelling put emphasis on the fact that every classes of software artefacts evolve. Everything evolve or will evolve soon or later. Large companies with a long background about software development know that. Along the years and decades they accumulate know-how about evolution-in-the-large. However, stability is still a very common yet implicit assumption made in many research projects. We are not aware for instance of much research work concerning meta-model evolution.

Co-evolution phenomena can easily represented by crossing one abstraction dimension with the time dimension. A pair of cells X and Y leads to co-evolution that will be noted X/Y-CoE. Figure 11 depicts how evolution interact with the engineering dimension revealing for example architecture/implementation co-evolution (i.e. A/I-CoE). The figure also suggests that the rate of changes greatly vary between software artefacts. For instance, the architecture of a software is expected to be much more stable that its implementation.

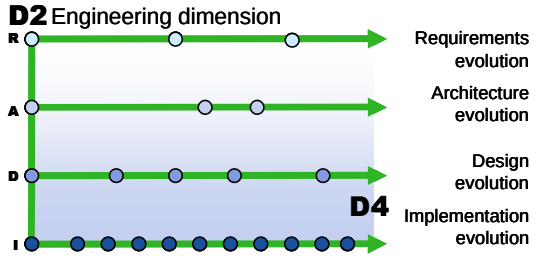


Figure 11. Crossing D2 and D4

Figure 12 add time to the meta-dimension. Notice that instance evolution (M0-E) corresponds to program execution. The rate of change is therefore extremely high, especially when compared with higher level of abstractions. Similarly models (e.g. programs) are much more unstable than meta-models (e.g. languages).

The conceptual framework is useful to structure ideas but it is sometimes too abstract to get a real feeling of what happen in practice. Let us illustrate the concept of meta-model and model co-evolution (M2/M1-CoE).

8. Example of M2/M1 co-evolution in industry

From our collaboration with Dassault Systèmes we can draw various conclusions about large scale software development and evolution. Most conclusions could also apply to other industrial contexts as well.

(1) Evolution-in-the-large is often achieved through ad-hoc processes, tools and concepts. This should not be surprising because many problems are discovered on the run. Pragmatic solutions are incrementally elaborated in a "as-needed" mode and sometime in "panic" mode to solve unexpected issues. For instance, in [20] we describe how ADELE, the configuration management tool developed by our team, was adopted at large by Dassault Systèmes and how the huge requirements in collaborative development lead to "hot" periods. Large companies where thousands of developers work on the same software cannot stop their development process when they find problems. They have to find solutions.

(2) Architecture is fundamental to evolution-in-the-large but its explicit representation with ADLs raises more problems than it solves. Industry is code centric and most architectural facts should be extracted from the code. Initially one of our goals was to study what kind of ADLs could be applied to support the evolution of CATIA [23][28]. We soon discover however that a much better approach was to provide architectural recovery and software exploration tools [22][29].

(3) The notion of software architecture is really more complex than the academic vision tend to explain and its exact nature really depends on the company culture and know-how [23]. In particular, we didn't found a single definition of architecture really helpful in practice. We

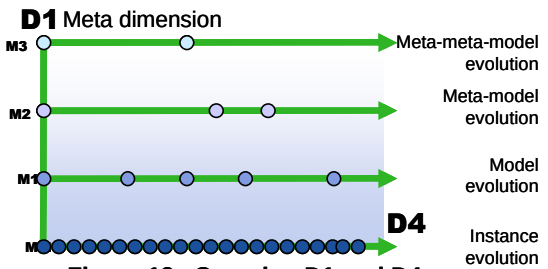


Figure 12. Crossing D1 and D4

found on the contrary that many of the architectural concepts used at-large within DS were beyond traditional concepts. For instance we identified the business architecture. It describes how software can be sold in parts. This structure is quite complex at DS.

(4) The company "know-how" is in part immaterial knowledge shared among the company, but in part materialized by in-house metaware tools. These tools support appliware development and constrain appliware evolution by enforcing specific processes and in-house quality standards. These tools are either bought and then customized, or developed internally by the tool support team. Over the last decades DS has developed a huge amount of metaware to support for instance configuration management, testing, component-based-development, etc. This range from sophisticated tools to hand-craft tools .

(5) The distinction must be made between metaware and appliware, between M2 and M1, between models and meta-models, especially in the context of software architecture. Too often these levels are confused in this context; in large part because architecture if a fuzzy notion. The distinction must be made between architectural models (A-M1), which are application dependent, and architectural meta-models which represent reusable know-how about building software (A-M2). This difference is illustrated in Figure 6 in the respective cells. Hofmeister and her colleagues describe reusable architectural know-how resulting from Siemens experience in [20]. This book is organized around 4 meta-models presented on the front and back covers of the book. Extracts of the architectural meta-models we recovered from DS' metaware can be found in [22][23].

(6) Everything evolve in large companies. In particular the notion of architecture and the architecture of applications. To be more precise both architectural meta-model evolution (A-M2-E) and architectural model evolution (A-M1-E) take place. In the first case this is the architectural know-how which evolves, in the second case this is the architecture of particular applications. This leads to architectural meta-model/model co-evolution issues (A-M2/M1-E). That is, without entering into the implementation details (I), one can observe that both architectural concepts and their occurences in software applications evolve.

(7) The evolution of the architectural concepts can have a strong impact on the architecture of the application, but also the other way around. For instance in the mid 90's DS decided to develop a component technology similar to Microsoft' COM but with also a set of unique features to cope with DS specific needs [21]. This technology evolved at the same time as the applications built using it. DS know-how about component-based architectures greatly evolved with the experience gained in developing large set of components (about 8000 today). Note that when the concepts underlying the component technology change, component-based applications may or may not be impacted. For instance sometimes an architectural concept reveals to be harmeful after a long period of use without noticable problem. This was the case for instance for a feature included in the DS component technology. This feature greatly simplified component developement but it later revealed to be responsible of significant decreases in performance when used at-large. DS then decided to remove it from the set of features available to develop software. From a conceptual point of view this corresponds to a removal of an element from the architectural meta-model. Components using this feature had to be identified to be upgraded. Sometimes, external events make it necessary to improve the architectural meta-model. For instance a few years ago DS decided to make its component technology available to partners such as Boeing. Before this DS used a visibility model based on the traditional public/ private distinction to control dependencies between software entities. This was enough within the context of DS, but not enough for externalisation because more levels had to be added to better control external dependencies. From a conceptual view, this modification just imply at the level of meta-model to change the type of the "visibility" meta attribute, as well as to update the constraints associated with this attribute in the meta-model. From a concrete point of view, the metaware tool that control dependency management was modified and the level of visibilty had to be assigned for each software entity concerned. From a conceptual point of view, this modification consiste in updating in architectural models the value of the visibility attribute of each entity concerned.

(10) *Metaware tools developed within large companies are often built in an incremental and in ad-hoc way, following the needs of the company*. These tools are often hand-craft using for example unix scripts to automate transformations. As shown in the previous example some transformations occur only a few time and building a tool from scratch could be too costly. One important issue in this context is to facilitate the production of metaware. Declarative meta-progamming or using meta-model driven environment are very promizing approaches in this context.

9. Conclusion

Software evolution is too often confused with program evolution. Software is much more than programs. Just like programs, languages follow Lehman's laws of continuing changes: in order for a *language* to continue to be useful (and used) in the real world it must change continuously. Languages are integral part of software. Languages, tools and programs evolve in parallel.

While architecture and implementation co-evolution has

been identified as a natural process during evolution-in-the-large, this paper has unveiled the existence of meta-model and model co-evolution, which is a generalisation of language/program co-evolution. We shortly described for instance the architectural meta-model/architectural model co-evolution problem as it occurs in industry.

These complex issues has been studied thanks to the provision of conceptual framework. The framework is based on the fact that software artefacts can be classified along a three abstraction dimensions. The meta-dimension is based on the four layers and includes models and meta-models. The engineering dimension distinguishes software artefacts according to the phase in which they are produced. The representation dimension makes it possible to model artefacts that range from implicit and fuzzy knowledge to very concrete representations used by tools.

Emphasis has been put on the need to make the distinction between metaware and appliware. Appliware is the set of applications, while metaware is software that help in developing and controlling appliware. In fact, software is metaware plus appliware. Software evolution should not be restricted to appliware evolution, metaware also evolves.

Some experts predict that the MDA standard could have a strong influence on the future of software engineering [17][14]. However, failure is still possible. Historically, most approaches looking only towards the future have failed. Roughly put, while ADLs were designed to make the architecture explicit (and failed), Model Driven Engineering is designed to make explicit models and meta-models. This is certainly the way to go, but this raises some questions. Will software engineers accept to draw UML models if evolution is not supported in a very effective way? What about extracting models and meta-models from existing software? What about model and meta-model co-evolution in the context of the MDE? Could we assume that "standard" meta-model will not evolve in the long run? What about reverse engineering of meta-models from legacy and proprietary metaware?

We see Model Driven Engineering as a very promising approach. But we also believe that this approach will fail if evolution is poorly supported and if legacy software is not taken into account. Metaware evolution and metaware reverse engineering are open research issues as well as effective tool support for meta-model/model co-evolution.

10. References

[1] T. Mens, J. Buckley, M. Zenger, A. Rashid, "Towards a Taxonomy of Software Evolution", USE 2003

[2] M. Felici, "Taxonomy of Evolution and Dependability", Workshop on Unanticipated Software Evolution, USE'2003.

[3] L. O'Brien, C. Stoermer, C. Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", SEI Technical Report CMU/SEI-2002-TR-024, 2002

[4] A.E. Hassan, R.C. Holt, "Architecture Recovery of Web Applications", ICSE 2002

[5] S. Boucetta, H. Hadjami, F. Kamoun, "Architectural Recovery and Evolution of Large Legacy Systems", IWPSE 1999

[6] Q. Tu, M.W. Godfrey, "An Integrated Approach for Studying Architectural Evolution", IWPC 2002

[7] J.B. Tran, M.W. Godfrey, E.H.S. Lee, R.C. Holt, "Architectural Repair of Open Source Software", IWPC 2002

[8] J. Zhao, H. Yang, L. Xiang, B. Xu, "Change impact analysis to support architectural evolution", Journal of Software Maintenance and Evolution, 14:317–333, 2002

[9] R. Wuyts, "A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation", PhD, Vrije yiversity of Brussel, 2001.

[10] K. Mens, T. Mens, M. Wermelinger, "Supporting unanticipated software evolution through intentional software views", USE 2002

[11] T. D'Hondt, K. De Volder, K. Mens, R. Wuyts, "Co-evolution of Object-Oriented Software Design and Implementation", Proc. Int'l Symp. Software Architectures and Component Technology: The State of the Art in Research and Practice, Kluwer, 2000

[12] OMG, "MDA: the OMG Model Driven Architecture", http://www.omg.org/mda/

[13] OMG, "Model Driven Architecture - A Technical Perspective", ormsc/01-07-01, 2001

[14] A. Kepple, J. Warmer, W. Bast, "MDA Explained - The Model Driven Architecture: Practice and Promise", Addison Wesley, 2003

[15] S. Kent, "Model Driven Engineering", LNCS 2335, 2002

[16] X. Blanc, P. Desfray, "Model Driven Engineering", in french, to appear in 2003

[17] J. Bézivin, X. Blanc, "MDA: Towards an Important Paradigm Change in Software Engineering", in french, Développeur Référence, http://www.devreference.net/Develop, July 2002

[18] V. Marangozova, "Linking the Software Architecture with Source Code", Master, in french, University of Grenoble, June 1998

[19] Softeam, "Guarantee permanent Model/Code consistency: Model driven Engineering versus "Roundtrip engineering", 2000

[20] J.M. Favre, J. Estublier, R. Sanlaville, "Tool Adoption Issues in Very Large Software Company", 3rd Workshop on Adoption Centric Software Engineering, ACSE 2003

[21] J. Estublier, J.M. Favre, R. Sanlaville, "An Industrial Experience with Dassault Systèmes' Component Model", Book chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, 2002

[22] J.M.Favre *and al.*, "Reverse Engineering a Large Component-based Software Product", CSMR'2001

[23] R. Sanlaville, "Software Architecture: An Industrial Case Study within Dassault Systèmes", PhD dissertation, in french, Univeristy of Grenoble, 2002

[24] R. Lammel, C. Verhoef, "Semi-automatic grammar recovery", Software Practice and Experience, 2001

[25] J. Estublier, J.M. Favre, "Component Models and Component Technology", Book chapter in Builiding Reliable Component-Based Systems, Archtech House publishers, 2002

[26] J. Warmer, "The Future of UML", available from www.klasse.nl

[27] C. Hofmeister, R. Nord and D. Soni. Applied Software Architecture. Addison-Wesley Publisher, 2000.

[28] Y. Ledru, R. Sanlaville, J Estublier, "Defining an Architecture Description Language for Dassault Systèmes", 4th International Software Architecture Workshop, 2000.

[29] J.M. Favre, "GSEE: a Generic Software Exploration Environment", 9th International Workshop on Program Comprehension, IWPC'2001

[30] M. Fowler, "UML distilled: A brief guide to the standard modelling language", Addison Wesley, 1999

[31] OMG, "Meta Object Facilities (MOF) Specification, Version 1.4", April 2002

[32] P. Desfray, "MDA – When a major software industry trend meets our toolset, implemented since 1994", Softeam white paper, 2001

MDS-Views: Visualizing Problem Report Data of Large Scale Software using Multidimensional Scaling*

Michael Fischer and Harald Gall
Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{fischer,gall}@infosys.tuwien.ac.at

Abstract

Gaining higher level evolutionary information about large software systems is key a in validating past and adjusting future development processes. In this paper we address the visualization of problem reports by taking advantage of the proximity introduced by changes required to fix a problem. Our analyses are based on modification and problem report data representing the system's history. For computation of proximity data we applied a standard technique called multidimensional scaling (MDS). Visualization of feature evolution is enabled through the exploitation of proximity among problem reports. Two different views support the assessment of a system design based on historical data. Our approach uncovers hidden dependencies between features and presents them in easy-to-evaluate visual form. Regions of interest can be selected interactively enabling the assessment of feature evolution on an arbitrary level of detail. A visualization of interwoven features can indicate locations of design erosion in the architectural evolution of a software system.

1. Introduction

Our work is focused on the evolution of software features since they are a natural unit to describe software from the perspective of the application user and the software developer as well. A software design may erode during the project's lifetime and features which were initially implemented in separated modules become more and more interwoven [8]. Our goal is to reveal this degeneration through the use of information collected during the implementation and maintenance phase.

In this paper we describe an analysis and visualization method for uncovering dependencies between sub-modules of software products source tree. Our approach is based on standard techniques and tools and any future project can be adapted easily to meet the data requirements for analysis. Existing projects often have no bug tracking system at all or the integration with the revision control system does not track relevant data.

Modification Reports (MR) and *Problem Reports* (PR) contain an overwhelming amount of information about the reasons for small or large changes to a software system. Groups of reports can be related to provide a clearer picture about the problems concerning a single feature or a set of features. Hidden dependencies of structurally unrelated but over time logically coupled files (i.e. files that most often are changed together although residing in separate modules or subsystems) further exhibit potential to illustrate feature evolution and possible architectural shortcomings.

The approach presented here addresses this problem by grouping feature related PRs and presenting the results in visual form. The input data for this process are selected from a *Release History Database* (RHDB) [12] which contains MR, PR and feature data. For every feature which shall be inspected the related PR information is selected from the RHDB. Now, the distance between two PRs can be expressed as the number of files commonly modified to fix both problems. The more files they have in common the shorter is the distance between the PRs. On the basis of this proximity data, groups of related reports are formed by applying a technique called *Multidimensional Scaling* (MDS) [14] on these data. Results of the MDS process are visualized in a two or optional higher dimensional space. PRs belonging to the same feature are indicated by

*This work is partially funded by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT) and the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

the same symbol which has been selected to represent the feature. Groups of PRs can be selected interactively and saved for further processing.

Grouping of PRs can reveal hidden dependences between features but can be also used to identify groups of commonly modified program code. Results from this analysis can be used as evidence for a poor system architecture or as indication of design erosion.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of related work in the area of visualization of large large scale software evolution. In Section 3 we introduce the data sources used. Section 4 describes the feature extraction process and its results. Section 5 gives short introduction in grouping using multidimensional scaling. Visualization of problem report data using different views to emphasize different relationships is discussed in Section 6. We conclude in Section 7 with an outlook to future work.

2. Related Work

In [17] Taylor and Munro describe an approach based on revision data to visualize aspects of large software such as active areas, changes made, or sharing of workspace between developers across a project by using animated revision towers and detail views. Since their approach is purely base on revision history, additional important information such as problem reports or feature data are not considered for visualization.

Similar to the our working environment used to produce the results presented in this paper, Draheim and Pekacki propose a framework for accessing and processing revision data via predefined queries and interfaces [9]. Linkage of their data model with other evolutionary project information – such as problem report data as required for our analysis – and making them accessible for external queries is beyond the scope of their work.

Mozilla has been already addressed, for example, by Mockus, Fielding and Herbsleb in a case-study about Open Source Software projects [15]. They also used data from CVS and the Bugzilla bug-tracking system but, in contrast to our work, focused on the overall community and development process such as code contribution, problem reporting, code ownership, and code quality including defect density in final programs, and problem resolution capacity as well.

Bieman et. al [5] used change log information of a small program to detect change-prone classes in object oriented software. The focus was on visualization of classes which experienced frequent changes together which they called *pair change coupling*. Instead of grouping coupled objects they used for visualization standard UML diagram together with a graph showing the number of pair change couplings between change-prone classes.

3. Building a Release History Database

Our analysis is based on three main sources: (a) modification reports (MR); (b) problem reports (PR); and (c) basically the executable program to extract feature information. Data in sufficient quantity and quality is offered via publicly accessible resource from the *Mozilla* project. The MR data are available via the projects Concurrent Versions System (CVS) [7], whereas the PR information is offered via the *Bugzilla* [1] bug tracking system. The source code package for building the executable are released frequently and contain all necessary files to compile the program.

3.1. RHDB

We have downloaded the relevant MR and PR, filtered, validated and stored the these data in our Release History Database (RHDB) [11, 12]. The nucleus of our RHDB is depicted in Figure 1. General information about objects - items or artefacts - of the software's program source and modification reports are stored in *cvsitem* and *cvsitemlog*, respectively. The relation between them was easy to establish, since the relevant information is contained in consistent form within the logs from the CVS repository. Problem reports from the bug tracking system are stored in the *bugreport* entity. Crucial in the reconstruction of the RHDB was the re-establishment of the linkage between modification reports and problem reports since no formal mechanisms are provided by CVS to link this information. In the rebuild process we used the problem report IDs found in the modification reports of CVS to link modification reports and problem reports. These IDs have been entered by the authors of source code modifications as free text. Natural problems were: context not clear in which an ID was used, incorrect report IDs (typos) or no ID at all. Whilst we were able to solve the first two problems with more or less effort, we didn't find a practicable solution for the third problem to reconstruct this information, e.g., from patch data. IDs in modification reports are detected using a set of regular expressions. A match is rated according to the confidence value we have assigned

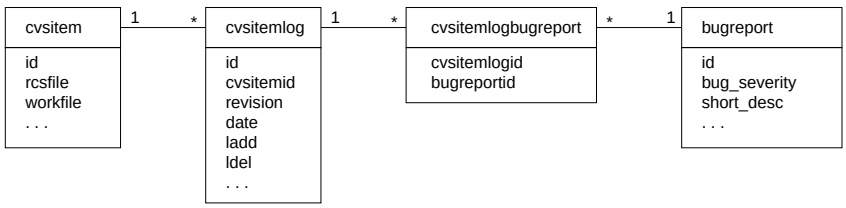


Figure 1. "Nucleus" of the RHDB

to the expression and can be *high* (h), *medium* (m), or *low* (l). The confidence is considered high if expressions such as <*keyword*><*ID*> can be detected whilst a five digit number just appearing somewhere in the text of a modification report without preceding keyword is considered low.

To verify the results from the reconstruction process we used patch information which are sometimes attached to problem reports. These patches contain file name information which can be used to validate the results from the linkage reconstruction. If a patch was found which confirms the linkage, the rating value is changed from (h) to (H), (m) to (M) and (l) to (L), respectively – details can be found in [11].

3.2. Selection of problem reports

To improve results of this analysis process, we need to reduce the impact of PRs not directly related with fixing a specific functional problem. We inspected the description of the largest reports and tried to find a criterion for the selection of our data sets. Reports such as "license foo" (98089,7961), "printfs and console window info needs to be boiled away for release builds" (47207,1135), or "Clean up SDK includes" (166917,888) - the numbers in parenthesis indicate the problem report number and number of referenced files respectively - are considered as not relevant since they primarily concern administrative problems. When the problem reports become smaller they are getting more interesting for the evalution of coupling between features. While the following two reports are still concerned with administrative issues "libtimer_gtk_s is causing link problems" (11159, 300) and "repackage resources into jar files" (18433, 289), the remaining reports begin to focus on programming and bug fix problems: "[meta] necko api revision bugs for embedding" (74221, 254), "[CSS] Rule matching performance improvements" (78695, 234), "Investigate switching output to use DOM Serializer" (50742, 234), "change nsCRT::mem* to mem*" (118135, 233). Thus, we have decided to use 255 as limit for the size of bug reports we accept. This is an arbitrary limit and specific to our configuration. Fortunately, no "major" or "critical" classified PRs are affected by this criterion.

4. Features

Since features are used in communication between users and developers it is important to know which features are affected by (future) functional modifications of a software system. According to [13] a feature is *a prominent or distinctive aspect, quality, or characteristic* [3, 4] of a software system or systems. For our purposes we will refer to the more practical definition of a feature as *an observable and relatively closed behavior or characteristic of a (software) part* [16].

Goal of the feature extraction process is to gain the necessary information to map the abstract concept of features onto a concrete set of files which implement a certain feature. To extract the required feature data we applied the software reconnaissance technique [18, 19] within our *Linux* (*RedHat 8.0* & *SuSE 8.1*) development environment. GNU tools [2] were already used sucessfully in [10] to extract feature data.

We first created a single statically linked version of *Mozilla* with profiling support enabled. From several test-runs where the defined scenarios (see Table 1) were executed, we created the call graph information using the GNU profiler. The call graph information again was used to retrieve all functions and methods visited during the execution of a single scenario. Since our analysis process works on the file level, we mapped function and method names onto this higher abstraction level. In the next processing step, "feature data" were extracted from file name mappings using set operations, e.g., the *Xml* feature using the following expression:

$$\mathrm{Xml} = (\mathrm{MathML} \cap \mathrm{XML}) / (\mathrm{Core} \cup \mathrm{HTTP} \cup \mathrm{PNG} \cup \mathrm{fBlank} \cup \mathrm{hBlank} \cup \mathrm{ChromeGIF}).$$

Table 1. Scenario definitions with features

Scenario	Description	Feature	Color	Files
Core	mozilla start / blank window / stop	Core	White	705
HTTP	TrustCenter.de via HTTP[1]	Http	DeepPink	28
HTTPS	TrusterCenter.de via SSL/HTTP[2]	Https	MediumGreen	6
File	read TrustCenter.de from file	-	-	-
MathML	mathematic in Web pages[3]	MathMlExtension	YellowGreen	13
About	"about:" protocol	About	Gold	3
PNG	sample image[4]	ImagePNG	DarkOrange	10
XML	XML Base[5]	Xml	MediumOrchid	65
JPG	JPEG Karlskirche[6]	ImageJPG	Cyan	16
fBlank	read blank html page from file[7]	Html	DeepSkeyBlue	76
hBlank	blank html page via HTTP[8]	-	-	-
ChromeGIF	Mozilla logo[9]	ImageGIF	SlateBlue1	4
Image	-	Image	OrangeRed1	3

where the names represent the set of files extracted in the previous steps from the executed scenarios. Table 1 also lists the names assigned to the features, the colors which are used later, and the number of files retrieved. E.g., for the *About* feature we determined to following set of files: `content/base/src/nsTextContentChangeData.cpp`, `xpfe/appshell/src/nsAbout.cpp`, and `xpfe/appshell/src/nsAbout.h`. Finally, we imported these filename information into the RHDB along with the release number of the program from which the data were retrieved. In our case it was *Mozilla* version *1.3a* with the official freeze date 2002-12-10.

5. Introduction to multidimensional scaling

The goal of multidimensional scaling (MDS) is to map objects $i = 1, \ldots, N$ to points $\|\mathbf{x}_i - \mathbf{x}_j\| \in \mathbb{R}^k$ in such a way that the given dissimilarities $D_{i,j}$ are well-approximated by the distances $\|\mathbf{x}_i - \mathbf{x}_j\|$ whereas k is the dimension of the solution space. MDS is defined in terms of minimization of a cost function called *Stress*, which is simply a measure of lack of fit between dissimilarities $D_{i,j}$ and distances $\|\mathbf{x}_i - \mathbf{x}_j\|$. In its simplest case, *Stress* is a residual sum of squares:

$$\mathrm{Stress}_D(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \left(\sum_{i \neq j} (D_{i,j} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2 \right)^{\frac{1}{2}}$$

where the outer square root is just a convenience that gives greater spread to small values [6].

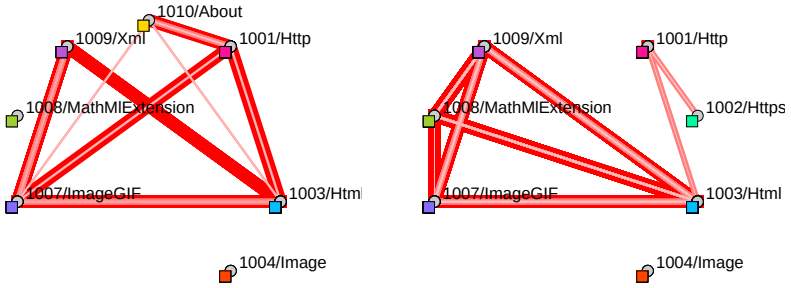
For our experiments we used *metric distance scaling* which is a combination of *Kruskal-Shepard distance scaling* and *metric scaling*. *Kruskal-Shepard distance scaling* is good at achieving compromises in lower dimensions (compared to *classical scaling*) and *metric scaling* uses the actual values of the dissimilarities in contrast to *non-metric scaling* which considers only their ranks [6].

The generation process of the dissimilarity matrix can be formally described as follows. A problem report descriptor d_i of a problem report p_i is built of all artefacts a_n which refer to a particular problem report via their modification reports m_k (linkage MR – PR; see Section 3):

$$d_i = \{a_n | a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i\}.$$

The distance data for every pair of problem report descriptor d_i, d_j are computed according to the formula below and fed

[1] `http://www.trustcenter.de/`
[2] `https://www.trustcenter.de/`
[3] `http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.xml`
[4] `http://www.w3.org/Math/testsuite/testsuite/General/Math/math3.png`
[5] `http://www.w3.org/TR/2001/REC-xmlbase-20010627/Overview.xml`
[6] `http://www.infosys.tuwien.ac.at/img/karlskirche.jpg`
[7] `file:///home/eu/robinson/WWW/blank.html`
[8] `http://intra.infosys.tuwien.ac.at:8092/~robinson/blank.html`
[9] `chrome://global/content/logo.gif`



(a) 1999 (b) 2000

Figure 2. Dependencies between features introduced by large problem reports

---

into the *Dissimilarity Matrix*.

$$\mathrm{dist}(d_i, d_j) = \begin{cases} 1 & \text{if } p_i \not\mathsf{R} p_j, \\ \frac{1}{2}(1 - \frac{n}{\min(s_i, s_j)}) & \text{if } p_i \mathsf{R} p_j \end{cases} \quad (1)$$

where s_i and s_j denote the size of the descriptors d_i and d_j respectively. The fraction $\frac{1}{2}$ is used to emphasize the distance between unrelated objects and "weakly" linked objects. All values are scaled according to the maximum number of elements the descriptors can have in common, i.e., they are scaled to the size of the smaller one.

An alternative way of specifying distances is to use edges and weights. We use a logarithmic function to emphasize the connections with higher values, while connections with lower values are weakened. This has the effect that the nodes with stronger connections are moved closer to each other than nodes with only a few connections. The weight for an edge between two nodes v_i and v_j are computed by the following formula, whereas n specifies the current number of connection between the two nodes and n_{max} the maximum number of connections between two nodes of this graph:

$$\mathrm{weight}(v_i, v_j) = 10 - \lfloor \frac{\ln(n)}{\ln(n_{max})} * 8 + 1.5 \rfloor \quad (2)$$

All weights are mapped by the above formula onto a range of $[1..9]$ where 9 means the closest distance. Other integer values not covered by the given range cause the visualization program *xgvis* [6] to hang or crash. Now we just need to define when two problem reports are linked: p_i and p_j in the RHDB are linked via a software artefact a_n if a modification report m_k exists such that

$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge m_k \mathsf{R} p_j$$

or two modification reports m_k, m_l exist such that

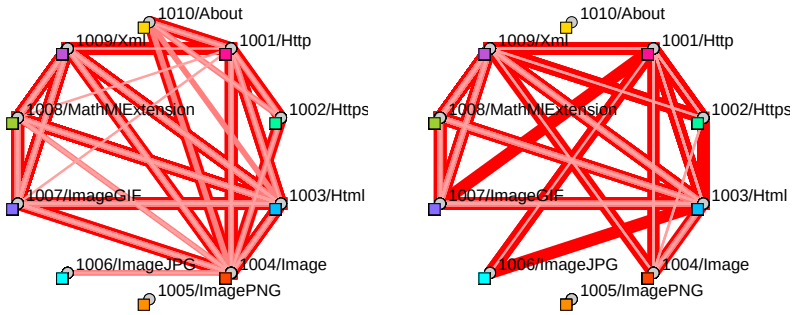
$$a_n \mathsf{R} m_k \wedge m_k \mathsf{R} p_i \wedge a_n \mathsf{R} m_l \wedge m_l \mathsf{R} p_j.$$

6. Views

Visualization is a useful technique to present complex interrelationships. We use two different types of views to facilitate the understanding of evolutionary processes in large software: a) *feature-view* focuses on the problem report based coupling between the selected features; and b) *project-view* depicts the reflection of problem reports onto the structure of the project-tree.

6.1. Feature view – projecting PRs onto features

This view focuses on the visualization of features and their dependencies via problem reports. The coupling between two features is symbolized via edges whereas the number of references is expressed as line-width. Since a feature comprises



(a) 2001 (b) 2002

Figure 3. Dependencies between features introduced by large problem reports

several items (i.e., object of the project tree), a connection between two features may consists of several lines of different width. They indicate the coupling of files through problem reports on feature level rather than file level. In fact, all entities contributing to a feature are drawn on the same position, which supports the impression that features are compared.

To reduce the amount of visible edges and to visualize only the most important ones, we used a threshold criterion of 10% of the number of references of the topmost problem report or 20 edges if the number of references is greater than 200. Likewise, in the graphical representation the number of problem reports two features have in common are not depicted. These numbers are listed in Table 2 for all features. This means that every edge in the following figures (except for Figure 2.a) represents at least 20 references. Due to the small set of files we identified for the features *ImageGIF*, *Image*, and *About* and the 10% limit, the coupling for this features may be not very indicative.

Figures 2 and 3 depict the results for the observation periods 1999, 2000, 2001, and 2002, whereas features are aligned on a circle and colored according to Table 1. From 1998 till 1999 – the situation is depicted in 2.a – we found virtually no coupling between features. The situation changed in the subsequent observation periods not dramatically but constantly. In Figure 2.b the situation for the year 2000 is depicted and indicates that the focus shifted from *Http* to other features such as *Html*, *Xml*, and *MathMlExtension*.

Consecutively, the situation changed substantially – as depicted in Figure 3.a – in year 2001. The partially connected graph has turned into an almost fully connected graph and also the number of reported and fixed problems have more than doubled (from 290 to 657). Except feature *ImagePNG*, all features are affected by system-wide changes. In 2002 (see Figure 3.b) the situation improved slightly since the number of reported problems dropped to 580.

6.2. Project view – projecting PRs onto project-tree structure

The goal of this view is the visualization of the reflection of problem reports onto the structure of the project-tree. Our method is to assign weights - to both, the edges of the tree structure and the edges introduced through the coupling of problem reports - and to search for groups in the resulting data set. Output of the optimization process is visualization using a conventional drawing program, whereas the resulting graph is enhanced with feature information and distance data.

6.2.1 General description

Input data for *xgvis* and the drawing program, e.g., *xgvis*, used for visualization are generated by a Java program. This program accepts some arguments which allows to control the data selection and generation process. A critical step in the data generation process is the selection of parameters for the weights since this has a direct impact on the final layout. For edges of the project-tree we use a *weight* of 10. Connections via problem reports are weighted 1 for every connection between two objects - a single report can affect several objects of different directories. This scheme gives more emphasize on the directory structure compared to connections with 1 or 2 problem reports between nodes. In a first phase of the data generation process, the objects of the project tree are assigned the respective nodes of the graph. The minimum child size parameter *minchildsize*

Table 2. Total number of "couplings" between features

	Feature										Period				
Feature	...1	...2	...3	...4	...5	...6	...7	...8	...9	..10	<2000	2000	2001	2002	Total
...1/Http	0	20	10	7	0	2	2	0	7	4	24	63	155	129	367
...2/Https		0	2	1	0	0	0	0	2	0	0	1	69	61	130
...3/Html			0	6	0	1	46	16	29	1	87	116	140	122	463
...4/Image				0	0	15	5	0	3	2	5	5	54	31	95
...5/ImagePNG					0	0	0	0	0	0	0	0	3	5	8
...6/ImageJPG						0	0	0	0	0	1	0	22	11	34
...7/ImageGIF							0	17	36	0	15	61	75	38	188
...8/MathMLExtension								0	67	0	2	9	32	72	114
...9/XML									0	1	7	34	105	110	249
..10/About										0	2	1	2	1	6

specifies which nodes remain expanded or will be collapsed. Collapsing means that the object information is moved to the next higher level. We used as default setting a value of 10. To simplify the resulting graph – the complete *Mozilla* project tree consists of more than 2500 subdirectories – we cut off unreferenced directories. With the compactification parameter *compact* it is possible to determine the limit for callapsing directories entries. A limit of 1 means that only unreferenced directories are collapsed/removed. The effect on the graph is that unreferenced leafes are suppressed.

In the following figures the project-tree, i.e., directory structure, is shown as gray nodes connected by black lines. The root node is indicated with the name "ROOT" and features are indicated by colored boxes according to the colors given in Table 1. Coupling between nodes as result of a common problem reports are indicated by pink lines. Broader lines and a darker coloring means that the number of problem reports two nodes have in common is higher. Black lines indicate the structure of the project tree. Since the optimization algorithm tries to place connected nodes close to each other, stronger dependencies can be grasped easily.

One marginal problem is the limited layout area in the two dimensional solution space - all nodes must be placed at least somewhere within a single plane - the placement of nodes after the optimization is only indicative within a certain radius. Naturally, this radius depends on the total number of nodes. By zooming-in, it is possible to give a better picture of otherwise overlayed areas. An n-dimensional solution space could yield better results but is very difficult to visualize. It is also possible that nodes are placed side by side, not because they share a common problem report, rather they have nothing in common with other nodes so they could be attracted by them and moved to a different location. In general, the layout after the optimization is one possible solutions. It also does not mean that a global minimum for the given distances has been achieved.

6.3. Project view results

The initial layout without any optimizationed clustering of the graph for three features – *HTTP* (DeepPink), *HTTPS* (MediumGreen), and *HTML* (DeepSkyBlue) – after data extraction from the RHDB is depicted in Figure 4. Black lines indicate the structure of the project tree, whereas pink lines indicate the coupling via problem reports.

A minimal set of three features - *HTTP* (DeepPink), *HTTPS* (MediumGreen), and *HTML* (DeepSkyBlue) - is depicted in Figure 5. The root of the project tree is indicated by "ROOT". Easy to see is the placement of HTML features on the right side, and *HTTP*, *HTTPS* on the opposite side. One exception is *.xpcom*: it can be shifted to the left side during the optimization process, but it is automatically moved back to its original position by the optimization program. This means the achieved result is stable and this is an optimal solution for the given weights. Interesting to see is the arrangement of `.security` and `.netwerk` since they were placed close together. In the original layout they were placed in oppositely directions (see Figure 4). What can be deduced from the placement, is the coupling between certain very close together placed nodes such as `.layout.html.base` and `.intl.lwbrk`, or `.netwerk.base` and `.netwerk.protocol.http`.

Figure 6 depicts all features we extracted from *Mozilla*. The root of the project tree has been shifted away from the center and its structure has been completely mangled by the optimization procedure on the basis of the given weights. As the previous figure already shows, it is also possible to assign distinct areas for the features, e.g., the *Http* and *Https* area by the colors DeepPink and MediumGreen. Features in other areas are not that distinguishable as desired.

Figure 7 depicts a detailed view of the cluttered area of Figure 6 with its coupling between different levels of the project structure. The positions of the nodes are slightly modified to have a better "viewing position" on the coupling. Since the coupling lines connect nodes of the project tree not along the project tree path, it may indicate unstable interfaces or

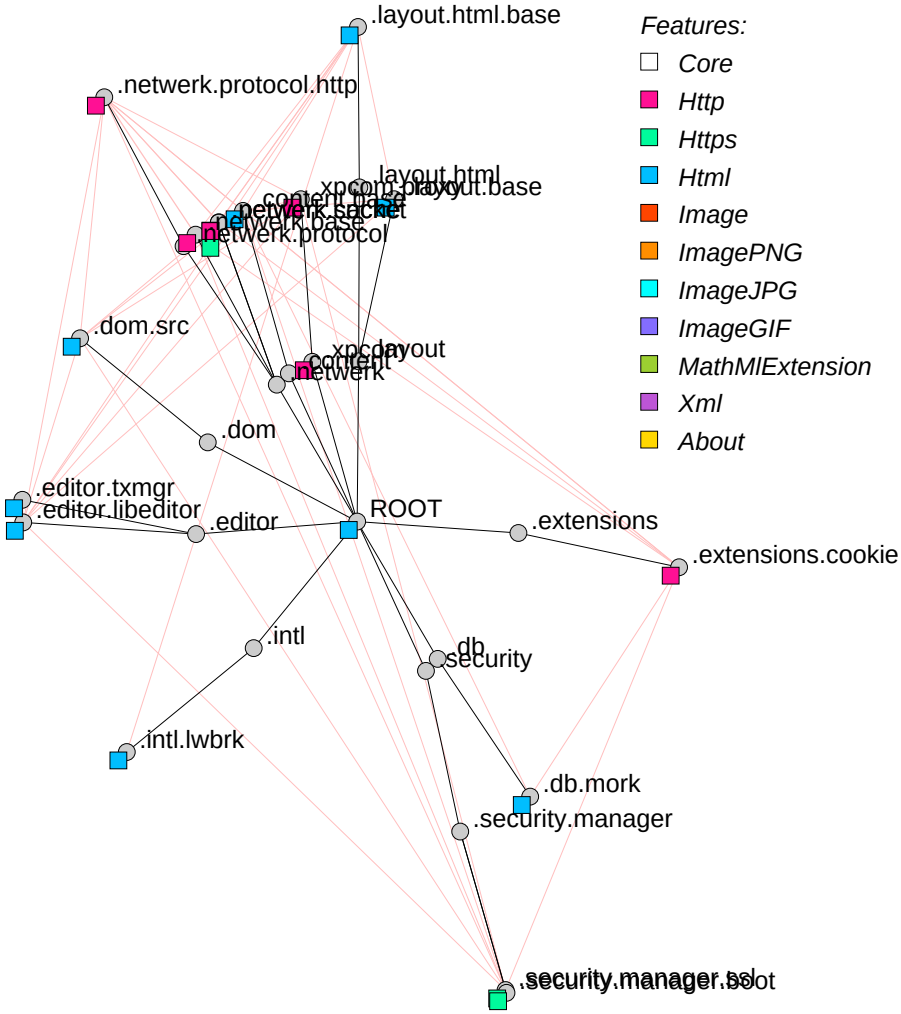
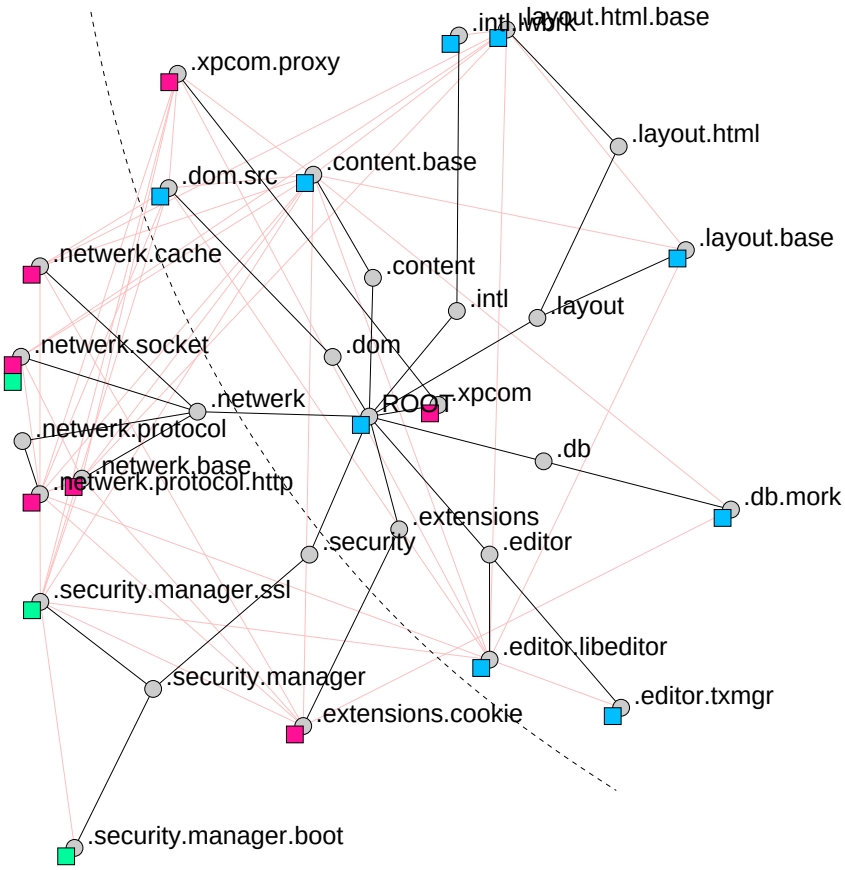


Figure 4. Initial layout for features: Http, Https, Html

Feature Html



Feature Http & Https

Figure 5. Features: Http, Https, Html

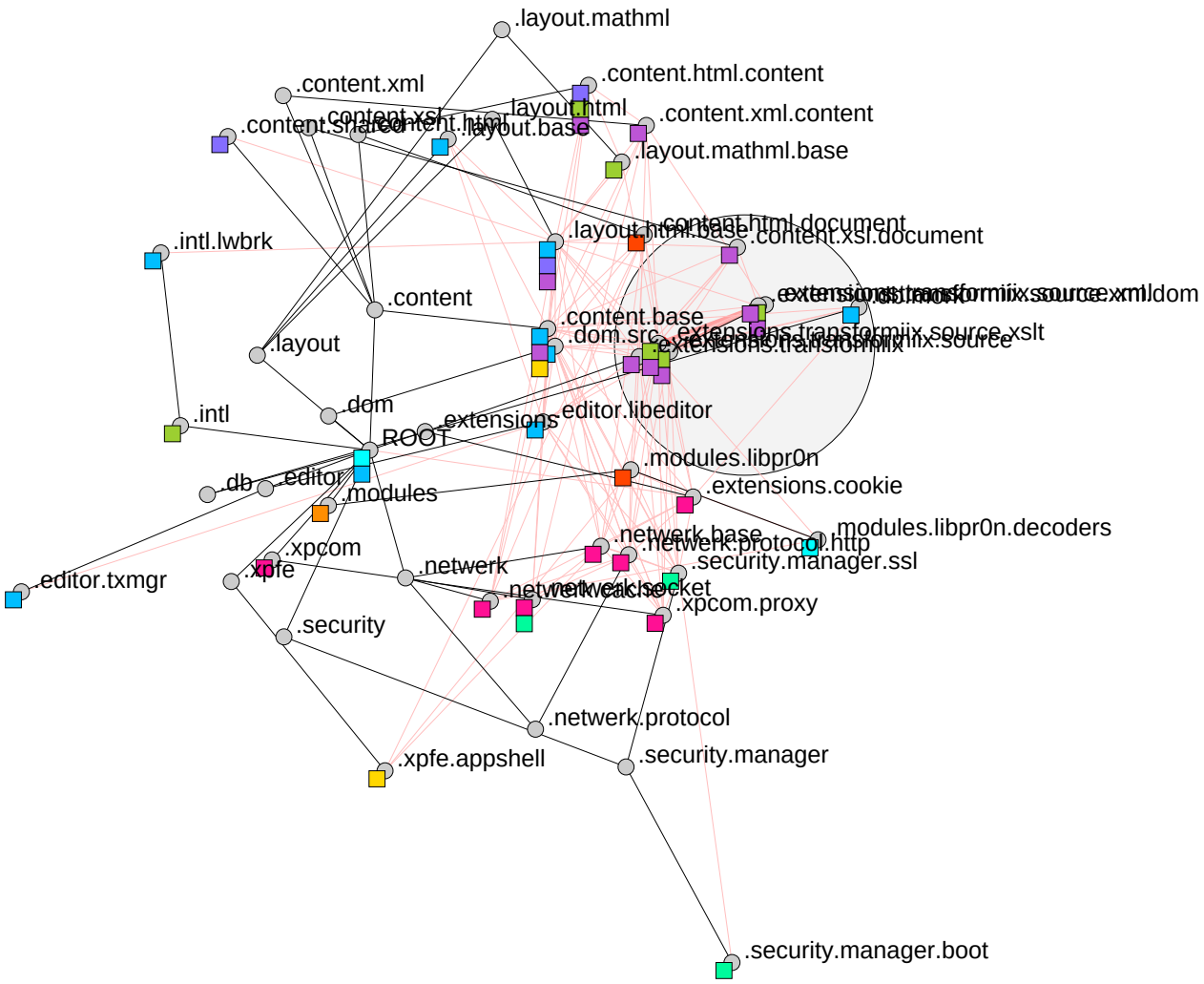


Figure 6. All features but no core

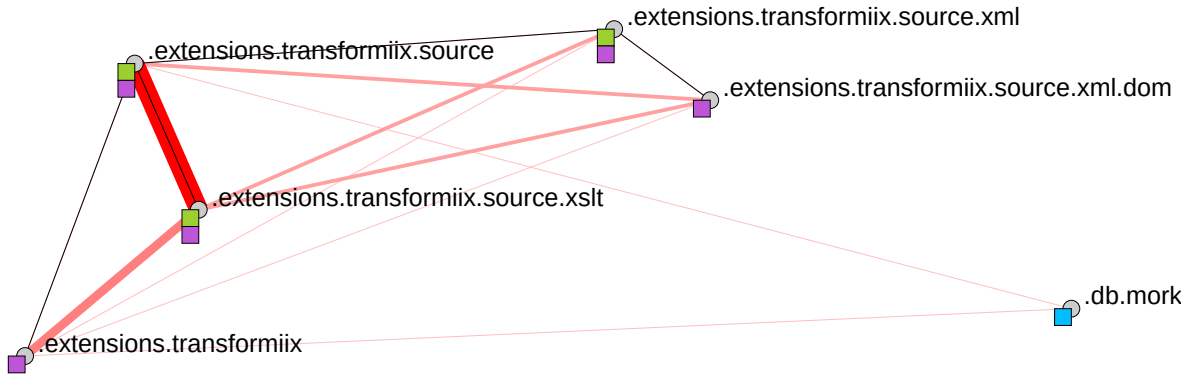


Figure 7. Detailed view of Figure 6

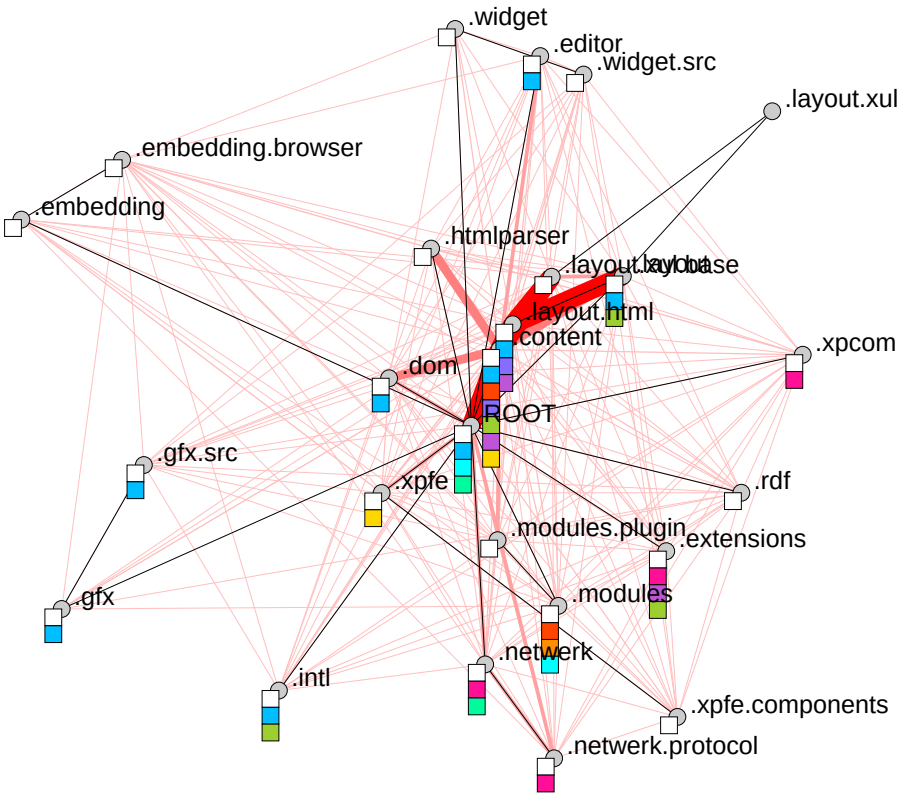


Figure 8. Core and features

weaknesses in the system design.

The core - indicated by white boxes - and all features are depicted in Figure 8 on a very coarse level. For this configuration we selected all reports which were rated *major* or *critical*. We also set the minimum sub-tree size to 250 (*minchildsize*) entities and the minimum number of problem report references to 50 (*compact*). As result we received a graph with 25 nodes, 215 edges induced by problem reports. By changing the values for *minchildsize* and *compact* it is possible to generate an arbitrary detailed graph of the while project. Since *Mozilla* has more than 2500 sub-directories a complete graph representation of the whole project tree is far beyond the illustration capabilities of this medium. It is intuitive that the most critical sub-systems in *Mozilla* are related with visualization which is also supported by our findings. The nodes with the highest density in severe problem reports are `.content` (with 595 references), `.layout.html` (438), `ROOT` (366), `.layout.xul.base` (220), and `.layout` (210). In fact, `ROOT` does not have any entities, but through the compactification of the project tree entities from lower levels have been moved to higher levels. The same is true for the other nodes such as `.com` or `.htmlparser`, since all the entities are moved to higher levels until the *compact* criterion is met. Nodes with fewer connections are `.dom` (161), `.xpfe` (121), `.netwerk` (119), `.modules` (118), `.modules.plugin` (106), `.network.protocol` (100), `.rdf` (100), `.htmlparser` (96). Another interesting aspect is the spread of edges. In total 215 connections between nodes are depicted. While some nodes are more focused on a single partner node, e.g., `.htmlparser` with 15 edges, others have a wider spread, e.g., `.xpfe` with 21 edges. Since `.layout.xul` does not share a problem with any other node, we can neglect this node and have 24 nodes sharing connections with others. For instance, `.xpfe` shares connections with 87.5%, or `.content` with 95.8% of the possible nodes.

7. Conclusions and future work

The graphical representation of dependencies between features based on problem report data opens a new perspective on the evolution of software systems through retrospective analysis by visualization. By intuition problem reports should have a minimal impact on different features. Situations where this is not the case can be grasped easily through the graphical representation, e.g., in case of overlapping or feature spreading. We have applied multi-dimensional scaling of problem reports linked with files and directory structures for the visualization of features of Mozilla for the years 1999 until 2002. The tool that we developed allows a domain expert to generate two specific views of relationships and dependencies of a large software system: (1) the *feature view* enables a projection of problem reports onto the files that realize a particular feature, thereby indicating otherwise hidden feature dependencies that have evolved over time (intentionally or unintentionally); (2) the *project view* enables a projection of problem reports onto the directory and project structure of a system and, as a result, depicts the logical coupling between modules, sub-modules, etc. introduced through changes over time.

First results using MDS are promising, thus we want to further explore this approach and test other large software systems to compare, for instance, the spread of features in commercial and other open source software. Of further interest are the exploration of higher dimensional solution spaces which should yield more optimized solutions. With *xgvis* this is quite difficult since the interactive selection and visualization works optimal only on two-dimensional data.

An interesting perspective for future work is the coupling of this visualization approach with architecture recovery systems. One possible application could be to gain insight into the impact of problem reports on architectural styles and patterns. A pattern search process might identify all implementations of a socket connection. The location information is augmented with information from the RHDB and visualized using MDS. More work will be devoted to visualization capabilities such as highlighting or selection of diverse areas for detailed inspection of problem reports. Furthermore, we will investigate the optimization algorithms as to allow placing related features as close to each other as their proportional computed strength indicates.

8. Acknowledgments

We thank the *Mozilla* developers for providing all their data for this case study to analyze the evolution of an large Open Source Software project.

References

[1] Bugzilla Bug Tracking System.
`http://www.bugzilla.org/`.

[2] GNU's Not Unix! - the GNU Project and the Free Software Foundation (FSF). http://www.gnu.org/.
[3] *Merriam Webster's Collegiate Dictionary*. Merriam-Webster, Incorporated, 10th edition edition, 1996.
[4] *The American Heritage Dictionary of the English Language*. Houghton Mifflin Co, 4th edition edition, 2000.
[5] J. Bieman, A. Andrews, and H. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of 11th International Workshop on Program Comprehension*. IEEE, 2003.
[6] A. Buja, D. F. Swayne, M. Littman, N. Dean, and H. Hofmann. XGvis: Interactive Data Visualization with Multidimensional Scaling. *Tentatively accepted for publication in the Journal of Computational and Graphical Statistics*, 2001. http://www.research.att.com/areas/stat/xgobi/papers/xgvis.pdf.
[7] P. Cederqvist et al. *Version Management with CVS*, 1992. http://www.cvshome.org/docs/manual/.
[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
[9] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland*. IEEE Computer Society Press, September 2003.
[10] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, November 2001.
[11] M. Fischer, M. Pinzger, and H. Gall. Analyzing and Relating Bug Report Data for Feature Tracking. In *10th Working Conference on Reverse Engineering (WCRE), Victoria, Canada*, November 2003.
[12] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 2003 International Conference on Software Maintenance (ICSM 2003), Amsterdam, Netherlands*, September 2003.
[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda). Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.
[14] J. B. Kruskal and M. Wish. Multidimensional Scaling. *Quantitative Applications in the Social Sciences*, 11, 1978.
[15] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.
[16] E. Pulvermüller, A. Speck, J. O. Coplien, M. D'Hondt, and W. DeMeuter. Feature Interaction in Composed Systems. In *Feature Interaction in Composed System*, 2001.
[17] C. M. B. Taylor and M. Munro. Revision towers. In *1st International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2002.
[18] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *International Conference on Software Maintenance*, pages 200–205, 1992.
[19] N. Wilde and M. C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.

Linking the Effect of Typographical Style to the Evolvabililty of Software

Position Paper

Andrew Mohan **Nicolas Gold**

Information Systems Group
Department of Computation
UMIST
UK

a.mohan@postgrad.umist.ac.uk
n.e.gold@co.umist.ac.uk

1. Introduction

The fact that comprehending software is a costly business is not in question. The question is whether this cost can be controlled, even reduced, as the software evolves. To answer that question one must analyse the source code, the key artefact in this evolutionary software lifecycle, to discover and maintain its evolvable quality.

This paper presents the problem of program comprehension and its relationship with programming style. It defines programming style as those characteristics of source code associated with formatting and commenting (i.e. typographical style [1]). The paper also outlines a position whose ultimate aim is to support the software evolution process through maintaining comprehensibility. This aim could be achieved by managing the cost of increasing code comprehensibility (in terms of deviation from a base programming style), through the application of groomative maintenance [2].

2. The Problem

In the software life cycle, maintenance is the final stage, taking place once the developed software has been incorporated into the business. However it is the most costly activity taking up between 40 and 70 percent of the cost of any software system [3, 4].

Software evolution is the result of the application of maintenance to software over time. To apply maintenance to existing code, the maintainer firstly requires a sufficient level of comprehension of that code [5, 6, 7]. This process of program comprehension is the most costly activity of software maintenance [8]. The key artefact in this process is the source code itself [9]. The ease of comprehending this source code is strongly influenced by the programming style (e.g. use of comments, variable naming, indentation) employed by the original developer and subsequent maintainers [1]. Therefore determining software's stylistic quality, by learning (or discovering) the base programming style, is a desirable activity.

Groomative maintenance is the activity in which software is changed, without changing its functionality, to improve its maintainability [2]. This maintenance activity is applied because as a program evolves it becomes more complex, and thus more difficult to comprehend and maintain [10]. If groomative maintenance is applied to increase the stylistic quality of the program, this should improve its comprehensibility and consequently maintainability (and evolvability). There are several associated problems:

1. What is program style?
2. Why and how does program style affect program comprehension?
3. When does this effect become problematic?
4. How can program style be learnt?
5. Who will benefit from improving the stylistic quality of a program?

To define program style (and metrics to record and subsequently compare program style), one must consider that these must reflect the quality of the software in terms of comprehensibility (as opposed to identifying its author for instance [11]). The difficulty is to identify what programming style attributes need to be measured that affect the quality attribute of comprehensibility. A major part of this difficulty is that each programmer is individual. The styles they prefer and that are easier for them to comprehend are individual to them [12]. This individual learning (or constructivist learning) implies that a coding standard is wholly effective only for that particular individual (although the imposition of organisational coding standards may facilitate a "middle ground" position that is sufficiently effective for everyone). Our work is focused upon the learning (or discovery) of a coding standard to use as a quality benchmark. This can then be used to measure the degradation of code quality in respect to comprehensibility for that individual, whether they be an individual person, group or company.

To clarify that any changes in comprehensibility result from changes in the coding style, a method of rating the comprehensibility of the program needs to be identified. This could be achieved either manually or automatically with some kind of tool. The problem with a manual method is that this is very subjective and costly (although accurate in terms of judging comprehensibility for that particular maintainer). An automatic method is objective but may be limited in terms of the capabilities of the evaluation tool.

3. Proposed Solution

The solution to the above problem can be expressed as proving, or otherwise, the following hypotheses:

1. A quality attribute of programming style can be learnt (or discovered) and related to program comprehension.
2. The degradation in the stylistic quality of a program is associated with an increased cost in comprehending and therefore maintaining it.
3. A degradation "boundary" for stylistic quality can be determined in the evolution of a program which can indicate the need for the application of groomative maintenance to improve this quality aspect.

The style used when writing or maintaining a program has a direct impact upon the quality of the software and consequently upon a program's comprehensibility and

maintainability [13]. Furthermore, as an evolving program changes its complexity increases unless maintenance is undertaken to reverse this [10]. This leads to the possibility of using programming style as a stylistic coding quality standard. An example standard (based upon [14, 15], but not exhaustive) would consider:

- Module Length - average of non blank lines
- Identifier Length - average
- Comments - percentage of program
- Indentation - ratio of initial spaces to chars
- Blank Lines - percentage of program
- Line Length - average of non blank lines
- Embedded Spaces - average number per lines
- Constant Definition - percentage of user identifiers that are constants
- Reserved Words - number of different reserved words and standard functions used
- Included Files – number of occurrences.

If we look at indentation as an example, here a stylistic standard should indicate a level from 2 to 8 (the normal upper limit). However specifying an exact level is more problematic. This is because overly indented programs hinder comprehension, due to the associated increase in both the horizontal and vertical costs of reading the program [13, 16]. Miara et al discovered that indentation is needed in a program, as no indentation makes a program difficult to comprehend. Indentation is therefore desirable in a program and should be at a moderate level, i.e. 2 or 4 spaces. However an important factor regarding comprehensibility, is that whatever indentation style is used it should be consistent throughout [17].

The degradation of a program's stylistic quality, derived by the measurement of deviance from the standard used in version X+1 against version X, could be used to predict when groomative maintenance should be applied to the code to improve its falling stylistic quality. However there is a requirement to demonstrate that the degradation in quality, measured through this deviance, is related to increased difficulty in comprehending the code. This would also provide the necessary business benefits to undertake the work.

To model the changes in comprehensibility we are exploring the use of an automated method: hypothesis-based concept assignment (HB-CA) [18]. This is a method for automatically recognising concepts (descriptive terms nominated by the programmer, e.g. updating a policy), within a program and matching them to sections of the code to help the maintainer rapidly build an initial understanding of the program [19]. The number of concepts identified or, in particular, the number that are not, could be used as a measure of program comprehension, i.e. a degree of difficulty modeller. HB-CA is particularly suitable for this task because it uses those clues in the source code (e.g. comments and identifiers) that maintainers use when forming hypotheses about a program [18]. HB-CA has a knowledge-base defined by the maintainer containing concepts of interest from the application and software engineering domains, and possible source code clues to these (e.g. words that might be used in identifiers or comments). The method creates hypotheses for the appropriate concepts when it finds a clue, identifies areas of source code where hypotheses for similar concepts are found (using a flexible concept-oriented rather than location-

oriented approach, see [20]), and assesses the evidence in each area to provide the most likely description for that code.

The establishment of the relationship between programming style and comprehensibility via concept assignment is to be achieved by measuring the deviance in stylistic coding quality of version X from version X+1 and relating this to the concepts with each version. This evolutionary measurement could then predict an effect upon program comprehension using the stylistic quality or at least highlight offending areas of code that have caused the effect (we have detailed a framework on concepts and the comprehensibility of evolving programs using a version of HB-CAS with initial case studies, see [21]).

4. Final Remarks

The ability to predict an effect upon program comprehension using the degradation of code quality may indicate the need for groomative maintenance to reinforce the quality standard upon the software. This is analogous with the process of rejuvenating software to prevent or reverse the effects of software aging [22, 23]. If concept assignment can be used to model the effect of program style upon program comprehension, during the evolution of that program, then HB-CAS can be used as a modeller of certain aspects of software quality. Indeed the evolution of the concepts themselves could be capable of indicating to maintainers a way of producing more comprehensible code by, for example, indicating candidates for refactoring [24].

The position presented in this paper is that the automatic modelling of software quality, given both a measurable stylistic coding standard and a relationship to comprehensibility, has the potential to contribute to reducing the program comprehension burden associated with evolving software. This is achieved by ensuring that by adherence to its stylistic quality standard, the evolvability of the code is maintained or even improved.

References

1. Oman, P. & Cook, C. (1990). Typographic style is more than cosmetic. Communications of the ACM, vol.33, nos.5, pp506-520.
2. Chapin, N., Hale, J.E., Khan, K.Md., Ramil, J.F. & Tan, W. (2001). Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, vol13, pp 3-30.
3. Lientz, B.P. & Swanson, E.B. (1980). Software Maintenance Management; Addison-Wesley, Reading M.A..
4. Takang, A.A. & Grubb, P.A. (1996). Software Maintenance – Concepts and Practise. Int. Thomson Computer Press.
5. Brooks, R. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, pp543-554, June 1983.
6. Mayrhauser, A.von & Vans, A.M. (1995). Program Comprehension during Software Maintenance and Evolution; IEEE Computer, vol.28, pp44-55.
7. Pennington, N. (1987). Stimulus Structure and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology, 19, pp295-341.

8. O'Brien, M.P. & Buckley, J. (2001). Inference-based and Expectation based Processing in Program Comprehension. Proceedings 9th International Workshop on Program Comprehension, IEEE Computer Society, pp71-78, Toronto, Ont., Canada, 12-13 May, 2001.
9. Dromey, R.G. (1995). A Model of Software Product Quality; IEEE Trans. of Software Engineering, vol.21, nos.2, pp146-162.
10. Lehman, M.M. & Belady, L.A. (1985). Program Evolution, Processes of Software Change, Academic Press Inc. Ltd..
11. Krsul, I. & Spafford, E. (1997). Authorship analysis: Identifying the author of a program. Computers & Security, vol.16, nos.3, pp248-259.
12. Exton, C. (2002). Constructivism and Program Comprehension Strategies: Proc. 10th International Workshop on Program Comprehension, Paris, France, 27th-29th June 2002, pp281-284.
13. Shneiderman, B. (1980). Software Psychology - Human Factors in Computer and Information Systems; Little, Brown and Company.
14. Berry, R.E. & Meekings, B.A.E. (1985). A style analysis of C programs; Communications of the ACM, vol.28, nos.1, pp80-88.
15. Oman, P. & Cook, C. (1990). A taxonomy for programming style. 18th ACM Computer Science Conference Proceedings, pp244-247.
16. Kernighan, B. & Plauger, P.J. (1978). The Elements of Programming Style; McGraw Hill.
17. Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B. (1983). Program Indentation and Comprehensibility; Communications of the ACM, vol.26, nos.11, pp861-867.
18. Gold, N.E., Bennett, K.H. (2002). Hypothesis-Based Concept Assignment in Software Maintenance, IEE Proceedings – Software, vol. 149, no. 4, pp103-110.
19. Biggerstaff, T.J., Mitbander, B.G. & Webster, D.E. (1994). Programming Understanding and the Concept Assignment Problem; Communications of the ACM, vol.37, nos.5, pp72-83.
20. Gold, N.E. & Bennett, K.H. (2001). Flexible Method for Segmentation in Concept Assignment. Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC) 2001, pp. 135-144, 12-13 May 2001, Toronto, Canada.
21. Gold, N.E. & Mohan, A.M. (2003). A Framework for Understanding Conceptual Changes in Evolving Source Code. To appear in Proc. International Conference of Software Maintenance, IEEE Computer Society, Amsterdam, Netherlands, 22nd-26th September, 2003.
22. Castelli, V., Harper, R., Heidelberger, P., Hunter, S., Trivedi, K, Vaidyanathan, K. & Zeggert, W. (2001). Proactive management of software aging; IBM Journal of Research & Development, vol.45, nos.2.
23. Bobbio, A., Sereno, M. & Anglano, C. (2001). Fine grained software degradation models for optimal rejuvenation policies. Performance Evaluation, vol.46, pp45-62.
24. Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Inc.

Challenges of Highly Adaptable Information Systems

Stephen Cook, Rachel Harrison, Timothy Millea & Lily Sun
Applied Software Engineering Research Group
University of Reading

18th August 2003

Keywords: architecture description, autonomic computing, design pattern language, e-learning, information system architecture, simulation, software evolution, system dynamics

1 Introduction

The success of personal, networked computing (most obviously in the form of the World Wide Web) has encouraged computerisation in application domains that were previously found (or assumed) to be unsuitable for it. Some of these domains are characterised by:

- imprecise and volatile requirements;
- frequent reconfigurations of processes, strategies and objectives;
- complex rules with innumerable exceptions;
- high (and often rising) user expectations (e.g. for usability, customisation).

In other words, information systems in these domains must be highly adaptable if they are to satisfy users' complex and rapidly evolving requirements.

This position paper identifies four current research areas in software engineering that are critical success factors for the development of highly adaptable information systems. The e-learning domain is used as a running example. Section 2 introduces some background material and related work in the areas of information system architecture, software evolution and e-learning. Section 3 outlines the issues that define this research programme. Section 4 relates these concerns to current research in the Applied Software Engineering Research Group at the University of Reading.

2 Background and Related Work

2.1 Architecture and evolution in information systems

The architecture and the evolution of any information system are closely related, as illustrated by their definitions. IEEE Standard 1471-2000 defines architecture as:

> 'The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.' [12]

The phenomenom of software evolution, first identified by Lehman and Belady [15], refers to a process of continual change in software systems, particularly in the growth of their functionality and complexity. Implicitly, each incremental step in the evolution of a software system involves the adaptation of some of its architectural properties and the preservation of others.

The architectural properties of any particular system are not equally adaptable. Some may be so difficult (and therefore expensive) to change, that they are effectively invariants of a system, or of a product line of systems, or even of an entire enterprise. This kind of architectural property can be thought of as an investment that is intended, in part, to reduce the costs of future adaptations to the system. (Arguably, the return on investment of architectural work should be measured in terms of reducing the system's maintenance costs.) However, over the lifetime of a system, its evolution may expose either weaknesses or inflexibility in its architecture (especially if the evolution was unanticipated), which in turn may raise the costs of ownership and even threaten the viability of the system.

The behaviour of these relationships is not yet well understood; case studies of real-world domains that require highly adaptable information systems can improve our knowledge.

2.2 Flexible e-learning systems

Computer-based training (CBT) and distance-learning are well-established, niche alternatives to traditional ('chalk-and-talk') models of education. The e-learning [19] concept builds on these traditions but also adds powerful new ingredients drawn from network-centric computing, computer-supported co-operative work (CSCW), adaptive environments, flexible processes and component-based software reuse. Effectively, e-learning may be regarded as a new paradigm of education that could improve flexibility, quality and participation in education and training [17, 13]. Ambitious plans are already being made for e-learning to play a major role in expanding higher education [5]; the University of Reading is directly involved in these initiatives through its leadership of the Thames Valley New Technology Institute[1].

Considerable resources have been applied to e-learning by industrial trainers, educational institutes and software producers. Several COTS products (e.g. Lotus LearningSpace[2], Blackboard[3], Oracle iLearning[4]) are available and have established a baseline of functionality that enables a tutor to publish teaching materials online, create discussion forums, organise assessments, and link to other resources [2, 10].

However current e-learning products have been less successful so far in providing more advanced functionality. For example, although many learning systems can provide simple customisation (e.g. a choice of font families), richer forms of personalisation currently depend on personal, usually face-to-face, interaction between teacher and learner. Consequently, if current e-learning systems were to simply replace traditional educational models, there would be a significant risk that the quality of the learning experience would deteriorate.

The challenges facing the next generation of e-learning systems include the provision of:

- improved ability to adapt rapidly and transparently to changes in a learner's profile and his/her progress through a learning package;
- better mechanisms for discovering and comparing relevant learning resources;
- the ability to specify the requirements of an instructional component and delegate the discovery of a resource that satisfies it to another process (e.g. a software agent);
- low-maintenance systems that are easy to inter-operate with both external resources and other education management systems.

This implies that e-learning systems face the challenge of how to evolve rapidly to become semantically rich, highly dynamic, distributed and personalised to the needs of individual users.

3 Architectural Challenges of Highly Adaptable Systems

3.1 Assessing information system evolvability

The IEEE definition of architecture cited in section 2.1 assigns a major role to architecture in defining the evolutionary principles of software-intensive systems. This role is poorly supported by existing modelling languages and tools, which tend to focus on system structure, operational behaviour and communications. However, the architectural properties of a system have to be considered at various levels of abstraction, from policies and principles (the 'Contextual' level) down to servers and programs (the 'Components' level) [20]. Consequently, architects and other stakeholders are often hampered in assessing whether the architecture of a system supports its expected evolution across the range of their concerns.

Some support for assessing architectural adaptability is provided by scenario-based methods (e.g. ATAM [8]). The explicit identification of architectural commonalities and variability has been recognised as particularly important in product-line engineering [9]. However, these approaches may not

[1] http://www.hefce.ac.uk/News/hefce/2002/NTIs.htm
[2] http://www.lotus.com/products/learnspace.nsf/wdocs/
[3] http://www.blackboard.com/
[4] http://ilearning.oracle.com/

scale up gracefully in domains such as e-learning that are characterised by complex and rapidly changing concepts. The variabilities in, for example, 'teaching resources' cover a potentially vast range of cross-cutting concerns (e.g. teaching methods and technologies, language and culture of learning milieu, students' level of education and prior experience, applied *vs.* theoretical focus of course). Compared with physical products, it is much more difficult to identify a stable, core 'chassis' that could be adapted using standardised, bolt-on components.

3.2 Architectures for low-maintenance information systems

Highly adaptable information systems are implicitly expected to adapt intelligently to a continuous stream of events, both from within the system and from its environment. Currently, complex adaptations of software usually require manual intervention by skilled (hence expensive and often scarce) personnel. Unless intelligent adaptive processes can be largely automated, it will be impossible to prevent highly adaptable systems from becoming 'support-bound' as they increase in scale.

3.3 Using design patterns in rapidly evolving domains

The design of highly adaptable systems should make use of design patterns [11] to achieve the following benefits:

- simplified software maintenance (assumption: the explicit use of well-known design patterns makes systems easier to understand);
- more adaptable systems (many 'classic' design patterns are directed at solving problems of system evolution).

However, it is unclear how design patterns should be used in domains that are evolving rapidly. For example, some approaches (e.g. [4]) have chosen to add explicit and detailed domain knowledge to individual patterns but to leave implicit any relationship to 'deeper', domain-independent patterns such as those catalogued in [11]. This approach could lead to inflexible, rather than evolvable, designs if there is a high risk that the knowledge will be modified in the future.

3.4 Assessing the dynamics of architecturally complex systems

It will not be possible to accurately predict either the dynamic behaviour or the evolution of highly adaptable systems by purely static analysis of their programs; some form of behavioural modelling or simulation will be essential. These models will need to take account of:

- the technological environment in which the system operates;
- the social and business processes that the system is intended to support;
- the 'global software process' [16] in which the evolution of the system is managed.

Models will also need to take account of a wide range of timescales, from very short-term (as services vary dynamically during a user's online session) to much longer-term (as services, agents and resources evolve through both technological and business life cycles).

4 Proposed Research Programme

4.1 Architecture description languages for evolutionary properties

A case study of the e-learning domain can explore practical approaches to assessing system evolvability (section 3.1) by investigating:

- which concepts of evolution are most relevant to highly adaptable information systems,
- how the concepts could be represented as a simple grammar, and
- how to anchor them to software engineering theory.

The results could assist the design of a structured language for unambiguously describing the evolutionary requirements and capabilities of a system in architectural terms (i.e. defining an evolution *viewpoint* and its *model*, to use the terminology of IEEE 1471-2000 [12]).

4.2 Architectures for autonomic information system services

One approach that could mitigate the risk of highly adaptable systems becoming support-bound (section 3.2) is autonomic computing. The term autonomic takes its meaning from the self-regulation of the central nervous system, in which functions such as heart beat rate, blood sugar levels and perspiration are adjusted without conscious thought and according to changing external conditions. By analogy, autonomic computing systems should regulate and maintain themselves to provide an optimal level of service without the conscious intervention of either the user or maintenance staff.

The e-learning domain provides an opportunity to assess the emerging results of our 'Autonomic Computing – Creating self-Evolving Software Systems' (ACCESS) project. ACCESS[5] introduces a model in which the evolution of a software system is guided by resolving the expressed concerns of its stakeholders. The resolution process operates within a space of possibilities defined by a software component market. This approach to automated 'just-in-time' system evolution develops ideas on ultra-rapid evolution that were proposed by Bennett *et al.* [6]. A schematic diagram of ACCESS's proposed architecture is shown in figure 1.

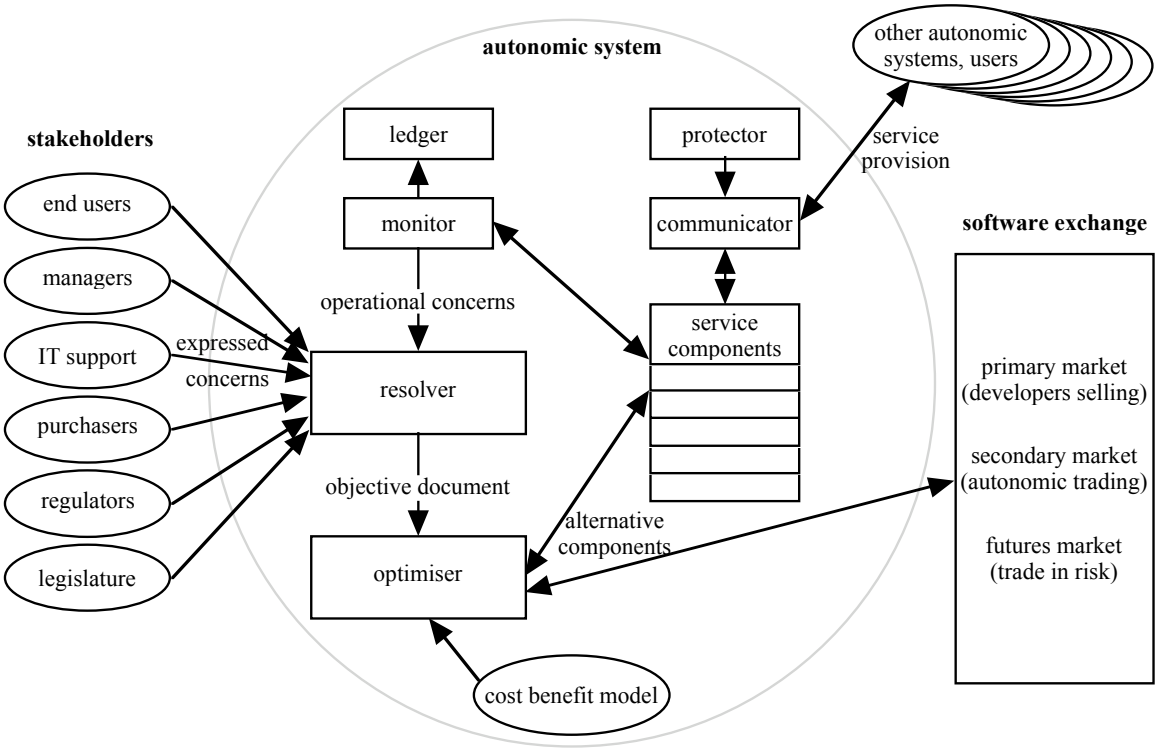


Figure 1: ACCESS schematic architecture

The ACCESS approach is by intention domain-independent and its architecture is highly abstract. Applying it to a specific rapidly evolving domain such as e-learning will raise many questions, including:

- is a market-based metaphor suitable for an activity such as education that has multiple, potentially conflicting, objectives?
- most competitive social situations, including markets, require a regulatory function that is independent of the broking function (i.e. the 'autonomic system' in figure 1); how should this be provided in an e-learning context?
- fairness in market-based allocation systems depends critically on all stakeholders having similar access to reliable information, but feedback to stakeholders is only implicit in the ACCESS

[5] EPSRC grant no. GR/S19066/01

architecture; what additional, possibly domain-specific, feedback channels are needed to ensure fair access for stakeholders to information?

4.3 Design pattern languages for rapidly evolving domains

The issue of how to relate design patterns to domain knowledge (section 3.3) can be addressed by investigating whether the concepts of system evolution provide an effective rationale for structuring design patterns in rapidly evolving domains. For example, Simon [18] suggested that the qualities of *hierarchical* and *nearly decomposable* organisation make it easier for a system to evolve. This may imply that when systems are required to be highly adaptable, their atomic design patterns should be as domain-independent as possible, and the binding to a specific domain should be achieved at a higher level, i.e. through an arrangment of selected patterns into a *pattern language* [1].

This approach could be seen as a generalisation of the coordination patterns that Andrade *et al.* [3] proposed as a mechanism for allowing business rules to evolve independently of core business concepts. It is also implicitly related to Lehman's SPE taxonomy [14]; the concept of patterns as reusable, domain-independent solutions seems similar to Lehman's *P-type* components (which are less likely to evolve), while pattern languages seem closer to his *E-type* components (which inevitably evolve).

Case studies and experiments are needed to explore which of these concepts are both relevant and scalable to the demands of e-learning systems. The development of new IT courses at Reading University provides an opportunity to conduct pilot studies, e.g. to compare the effectiveness of different approaches to the design and use of pattern languages for the e-learning domain.

4.4 Simulation of architectural evolution

Previous simulation studies of software evolution (e.g. [7]) have usually treated a software system as a black-box component and have not attempted to consider the effects of the system's architecture. On the other hand, simulation models of computer networks do take account of network architecture but often model the architecture as a simple, recursive structure. These approaches, if taken separately, may not be sufficient to produce accurate predictions of the dynamics of highly adaptable systems on either short- or long-term timescales.

One of the questions that we plan to investigate is whether models of software evolution can be improved by introducing selected information about the system's architecture. For example, referring again to Lehman's SPE classification of software components, does knowing the proportions of *E-* and *P-type* components in a system improve predictions of the course of its evolution? The ultimate goal would be to discover which architectural properties (i.e. fundamental design choices) of highly adaptable systems are most important in determining the shape of a system's subsequent evolution.

5 Conclusions

The demands of highly adaptable information systems provide a challenge for many aspects of software engineering, especially those related to architecture and evolution. The e-learning domain is very suitable for investigating these problems because it is entering a phase of rapid change. Furthermore, this rapidly evolving domain has a direct impact on many higher education institutions, which creates opportunities for researchers to also explore the practicality of candidate solutions to the problems that this paper has identified.

References

[1] Alexander, C., Ishikawa, S. and Silverstein, M., 1977. *A Pattern Language: Towns, Buildings, Construction.* New York: Oxford University Press.

[2] Anderson, M.D., 1997. Critical elements of an Internet based asynchronous distance education course. *Journal of Educational Technology Systems,* **26**(4), 383–388.

[3] Andrade, L., Fiadeiro, J., *et al.*, 2000. Patterns for coordination. *In*: Catalin-Roman, G. and Porto, A., ed. *Coordination Languages and Models*. Springer-Verlag (Lecture Notes in Computer Science, 1906), 317–322.

[4] Avgeriou, P., Papasalouros, A., *et al.*, 2003. Towards a pattern language for Learning Management Systems. *Educational Technology & Society,* **6**(2), 11–24.

[5] Beller, M. and Or, E., 1998. The crossroads between lifelong learning and information technology: a challenge facing leading universities. *Journal of Computer Mediated Communication,* **4**(2) December, .

[6] Bennett, K., Munro, M., *et al.*, 2001. An architectural model for service-based software with ultra rapid evolution. *In*: *Proceedings of the IEEE International Conference On Software Maintenance (ICSM 2001): Systems and Software Evolution in the Era of the Internet*, Florence, Italy, 7–9 November 2001. Los Alamitos, CA: IEEE Computer Society, 292–300.

[7] Chatters, B.W., Lehman, M.M., *et al.*, 2000. Modelling a software evolution process: a long-term case study. *Journal of Software Process: Improvement and Practice,* **5**(2–3), 95–102.

[8] Clements, P., Kazman, R. and Klein, M., 2002. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison- Wesley (Software Engineering Institute series).

[9] Coplien, J., Hoffman, D. and Weiss, D., 1998. Commonality and variability in software engineering. *IEEE Software,* **15**(6) November/December, 37–45.

[10] El-Tigi, M. and Branch, R.M., 1997. Designing for interaction, learner control, and feedback during Web-based learning. *Educational Technology,* **37**(3), 23–29.

[11] Gamma, E., Helm, R., *et al.*, 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley (Professional Computing series).

[12] IEEE Computer Society, 2000. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE-Std-1471- 2000. New York: IEEE.

[13] Learning Systems Architecture Lab, 2002. *SCORM Best Practices Guide for Content Developers*. Pittsburgh, PA: Carnegie Mellon University.

[14] Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE,* **68**(9), 1060–1076.

[15] Lehman, M.M. and Belady, L.A. (eds.), 1985. *Program Evolution: Processes of Software Change*. London: Academic Press (A.P.I.C. Studies in Data Processing, 27).

[16] Lehman, M.M. and Kahen, G., 2000. A brief review of feedback dimensions in the global software process. *In*: Ramil, J.F., ed. *FEAST 2000 Workshop: Feedback and Evolution in Software and Business Processes*, London, UK, 10–12 July 2000. London: Imperial College of Science, Technology and Medicine, 44–49.

[17] Schweizer, H., 1999. *Designing and Teaching an Online Course: Spinning Your Web Classroom*. Needham Heights, MA: Allyn and Bacon.

[18] Simon, H.A., 1969. *The Sciences of the Artificial*. Cambridge, MA: M.I.T. Press.

[19] Sloman, M., 2001. *The E-Learning Revolution: From Propositions to Reality*. London: CIPD.

[20] Zachman, J.A., 1987. A framework for information systems architecture. *IBM Systems Journal,* **26**(3), 276–292.

Design Erosion in Evolving Software Products

Jilles van Gurp, Jan Bosch, Sjaak Brinkkemper

University of Groningen, Vrije Universiteit Amsterdam

{jilles|jan.bosch}@cs.rug.nl, Sjaak@cs.vu.nl

Abstract. *Design erosion affects most, if not all, software systems. As these systems age, it becomes ever more difficult to make new changes until eventually it is more feasible to replace (or at least refactor) the software than it is to continue to the regular maintenance. In earlier work we have already identified a number of potential causes for this phenomenon. The case study presented in this paper, examines two eroded subsystems of a large software product. We look at various aspects of how the company involved has identified that the systems were eroded and how they managed to recover from that situation.*

1. Introduction

In this paper, we present the preliminary results of two case studies, which were conducted on two subsystems within the same company. Due to the preliminary and confidential nature of the case study and its results, we will not elaborate any further on the domain of the software or nature of the company involved in this paper. However, a full paper that will include these details is nearing completion.

For the moment, it is enough to specify that the company involved is a large multinational that, for the past few decades, has developed a large software product, which has been deployed on numerous (thousands) of customer sites worldwide. The software product, this company makes, consists of a number of application modules and an infrastructure layer that is common to these application modules. In the first case study, we examined the evolution of a component in the infrastructure layer. In the second case study, one of the application modules was examined. The purpose of the case studies was to explore the problems and issues encountered in large software developing organizations, such as the company involved in this study, with respect to design erosion.

Design erosion is a problem that affects most, if not all, large software systems. The phenomenon is also known as architectural drift [5], software aging [6] or architecture erosion [4]. Essentially the problem is that as software evolves, the software is incrementally changed to meet new requirements, fix defects or optimize quality attributes (adaptive, corrective and perfective maintenance [8]). However, these requirements may conflict with requirements in earlier iterations or may change the assumptions under which design decisions in earlier iterations were made. When faced with such requirement conflicts, there are two strategies for adapting the system to incorporate the changes:

- **An optimal design strategy.** No compromises are made with respect to design quality and the design of the software is enhanced in such a way that the new requirements can be incorporated without compromising the design integrity. While this strategy typically results in a good design, the associated cost may make it infeasible for some changes.
- **A minimal effort strategy.** Often complicated design changes can be avoided by stretching the design rules of the existing design a bit. While this may have consequences for the quality of the design, this strategy can be very effective in meeting the requirements on short notice.

In [3], we concluded that it is inevitable that in real world systems the first strategy is not always feasible. Consequently, cost considerations or time constraints sometimes force developers to take less than ideal design decisions. Over time, these less than ideal design decisions accumulate, resulting in what we call design erosion. Eroded software systems are typically hard to understand due to the many sub-optimal design solutions that have accumulated and complicated the design. Consequently, additional changes become harder and eventually may even become infeasible. When this happens, the only ways to resolve the situation are to either repair (e.g. using refactoring techniques) or replace the software. Both types of resolutions typically require a significant effort. In [3] we list a number of real-world projects that were affected by design erosion. In these examples, the subsequent effort to repair/replace the software spanned several years.

In a world that is increasingly relying on a growing quantity of ever-larger software, design erosion presents a serious problem. Affected software cannot be easily replaced or repaired. Failing to do so, however, may cause maintenance cost to rise and limits the flexibility of the affected software. Ultimately, eroded software may threaten the existence of the company that produces it as well as the existence of companies that use the software.

The cases we report on in this case-study, concern software subsystems that are part of a large software system that have both been affected by design erosion to such an extent that in both cases, the company chose to undertake an effort to address the issues, which in both cases implied several person-years of work. In one of the cases, this effort involved the refactoring of tens of thousands of lines of code. In the other case, the affected component had to be replaced by a new one to address the issues. The old version, representing a decade of evolutionary development and refinements, had to be discarded.

In the remainder of this paper, we will first discuss the research questions of this case study and our research method. After that, we will present some preliminary conclusions of the case-study. As outlined above, at this point, we cannot go into detail on the case studies themselves, however.

2. Research questions

The focus of our study is to explore how design erosion issues are identified, resolved and prevented in software developing organizations. Specifically, our study addresses the following research questions:

- **Symptoms.** What are the effects of design erosion on a system?
- **Identification.** How does an organization decide that their software is eroding and needs to be repaired? How does the decision process work?
- **Causes.** What are common causes for erosion?
- **Resolution.** What kinds of solutions are applied to fix an eroded system? How and when are decisions with respect to preservation and repair taken?
- **Prevention.** What practices help prevent erosion?

3. Methodology

In this section, we will outline the empirical research approach we have applied in the case studies and discuss its strengths and weaknesses. In his editorial for the journal of empirical software engineering [1], Victor Basili makes a plea for the use of empirical studies to validate theories and models that are the result of software engineering research. In a more recent publication, [2], Basili presents an overview of how empirical research has benefited NASA's Software Engineering Lab. When doing empirical research, a distinction can be made between qualitative empirical studies and quantitative studies. The approach advocated by Basili in [1] and [2], can be characterized as mostly quantitative. As can be seen in [2], collecting quantitative data is a labor-intensive process that needs to be tightly integrated with the development process. In a setting like NASA, where reliable, dependable software is required this is feasible. The results of the quantitative empirical research are used to optimize the development processes. However, in many other contexts this is much less feasible.

Qualitative data, on the other hand, is relatively easy to obtain and has the advantage of providing more explanatory information [7], which in an exploratory case study such as ours is very desirable. As is noted in [7], neither quantitative nor qualitative empirical research can prove a given hypothesis. Empirical research can only be used to support or refute a given hypothesis. A combination of both quantitative and qualitative studies is the best way of supporting a hypothesis [7].

In this exploratory case study, we use interviews as the primary tool of retrieving information. Consequently, our research is mostly of a qualitative nature. However, where possible, we complement the qualitative data with quantitative data provided by the interviewees (e.g. estimated defect rates, number of lines of code, etc.). Due to the confidentiality of such metrics within the company, a full quantitative study was not feasible. We have found that in general, software development organizations are very reluctant in providing or publishing such data.

In both case studies, the interviews followed the same pattern. We first met with the interviewees (software engineers, product architects, project managers) in a group for an introductory meeting. During this meeting, the purpose of the case study was communicated and a brainstorm session was held to select appropriate modules/components for further study. This meeting was also used for planning subsequent interviews. In the following meetings, both group and individual interviews were held during which more specific questions about the design and evolution of the system were asked.

In addition to interviews, we were given access to various documents including for example functional designs and requirements documentation. Using these documents, we were able to both verify/clarify certain statements of the interviewees as well as prepare specific questions in advance.

3.1 Case selection

Throughout both case studies, we have cooperated with the company's R&D department who were very much interested in the results of the case study for the sake of (a) providing an outsider analysis on the architecting and engineering practices, and (b) educating the product architects and software

engineers with the results. Using their expertise and knowledge of the company's product portfolio, two representative sub-systems were selected for further study and contacts with staff working on these sub-systems were initiated. Before selecting the cases, we had several meetings with the R&D department during which we discussed the organizational structure, the company's product architecture and the goals for the case study. In addition, an estimate of the time that was needed for both cases was made.

We used the following criteria for the selection of the cases:

- The systems had to be old enough to have endured design evolution.
- During the evolution, there must have been significant changes in the requirements.
- It should be possible to interview both people who were involved in the initial development of the system and people who were involved in restructuring the system for new requirements.

3.2 Validity

To ensure the correctness of our data and conclusions, we have used two methods:

- **Cross-checking.** In both cases, we interviewed multiple developers. This allowed us to compare their answers and verify whether there were any contradictions. In both cases we were also given limited access to software documentation, which allowed us further validate the information we received.
- **Feedback.** An important part of qualitative research is feedback. The data presented in this article consists mostly of our interpretation of interviews. Verifying whether this interpretation is correct is therefore an essential part of ensuring the validity of our case study. After each meeting, a report detailing our conclusions and interpretation was communicated back without the interviewees for feedback. The feedback has made us confident that the interviewees share our interpretation and conclusions.

However, there are a number of problems with our research approach that may affect the validity of our findings:

- **Representativeness of the cases.** By limiting ourselves to one company and one software product, we risk that this case study's conclusions may not be applicable to other domains and companies. Both the corporate culture and the domain this particular company is operating in affect our conclusions. However, based on our experience with case-studies in other companies, the corporate culture in this company is representative for many software developing companies. In addition, despite coming from the same company, the two cases we selected are dissimilar, so, any conclusions that can be generalized for these two cases may be applicable to other domains as well.
- **Quantitative data.** As explained earlier, we use a (mostly) qualitative approach. Complementing our data with quantitative metrics would certainly strengthen our conclusions. However, there are a few reasons why this study does so only to a limited extent. First of all, many relevant metrics that would need to be collected are generally considered as sensitive information in software development organizations. Consequently, we did not have access to raw quantitative data. However, the company does collect metrics and provided some qualitative information regarding e.g. defects to us that was based on this data. Additionally, this is an exploratory study. A quantitative study requires a more precise formulation of hypotheses, relevant quantifiable parameters and a model for the interpretation of values for these parameters. A study such as presented here may provide the necessary input formulating hypotheses and parameters for future quantitative studies.
- **Cases are not comparable.** We have deliberately chosen to research two cases from different domains to show that identification, resolution and prevention of design erosion works the same across domains. Therefore, both cases use different types of technology and involve people with different skills and training. On the other hand, both teams operate in a centrally managed release development project to design and build the sub-systems as part of one product. This makes it possible to compare the results of both case studies, notwithstanding some limitations.

4. Results & observations

In this section, we present the answers we found to the five research questions in the introduction in both our case studies. While we cannot go into much detail on either of the case studies, it is worthwhile to outline them in an abstract fashion.

- Case 1 examined the evolution of an infrastructure component that had evolved in a number of versions. In each version, significant architectural changes were made to this component. Recently, based on an internal evaluation it was decided to replace this component with a new component because the old one had eroded so much that repairing it

and adding new features was no longer feasible.
- Case 2 concerns an application module that was originally designed at the request of a particular customer. After an initial design project, the realization phase was handled by a relatively inexperienced development team. However, the resulting software had all sorts of problems. Eventually, the development was transferred to a more experienced team. This team subsequently decided to refactor and restructure the software.

As mentioned in the introduction, a full paper with much more detail is pending. In the remainder of this section, we will simply refer to them as case 1 and case 2.

4.1 Symptoms

A first step in preserving the design of a software system is to recognize the symptoms of an eroding system. Both cases we examined, exhibited similar symptoms of deterioration:

- **Low quality code.** In both cases, the developers working with the system were unhappy with the quality of the source code. They complained about misuse of language constructs, the lack of structure, inconsistent use of code standards, etc.
- **Uncertainty about specifications.** There was a great deal of uncertainty about the specification of the system in both cases. The designs were sketchy and incomplete. In the case 1, application developers actually depended on unspecified and even incorrect behavior of the infrastructure component. In case 2, changes were not properly documented (as prescribed in the companies development processes), effectively making the existing designs obsolete.
- **Regressions.** In both cases, fixes for defects often introduced new problems. Particularly in case 1, where at one point there were about 100 known defects, this was an important reason for discarding the old software. The estimated cost of fixing these 100 defects in combination with the near certainty of additional defects provided enough motivation for doing so.
- **Deployment problems.** In both cases, there were problems with respect to the usage of the system. In case 1, developers of application modules were relying on the unspecified, arguably incorrect, behavior of the component whereas in case 2, the functional design was no longer accurate because design changes were not documented.
- **Defect rates & cost.** An interesting aspect about the development process in the company is that it includes a fine-grained process for measuring defect rates and relating defects to particular development artifacts. In both cases, the developers we interviewed indicated that the amount of defects that needed to be fixed was substantially higher than in comparable systems.

4.2 Identification

In order to repair an eroded system, it has to be recognized first that the system is eroded and that it is worthwhile to undertake an effort to repair it. Obviously, in the systems we examined, the developers came to this conclusion. A number of factors may play a role in identifying erosion:

- **Evaluation**. In both cases, the decision to redevelop/redesign the system was taken after an internal evaluation of the software. In both cases these evaluations were prompted by problems with the existing software and a general feeling the software was not in a good condition (e.g. because of the symptoms outlined above). Additionally, in both cases, the defect rates that are routinely collected within this company were abnormally high, which provided additional evidence that both software systems had quality problems.
- **New requirements**. New requirements may call for enhancements that, given the quality of the system at that point, are infeasible. In both cases, it was the case that there were new requirements that were proving to be hard to realize in the existing systems.
- **Change of staff**. Developers, like most human beings, may be reluctant in admitting their own faults. In both cases, the developers that identified the erosion and took the initiative for the redevelopment of the software had not been involved in the original development of the software.
- **Defect Metrics.** In both cases, defect metrics played an important role. The development process includes a fine-grained process for collecting such metrics and the decision to redesign (case 1) or refactor (case 2) was partially based on these metrics.

4.3 Causes

In order to effectively repair an eroded system, the causes of the issues that are responsible for the erosion need to be understood. . We have found that both cases had a number of common issues. Consequently, these issues are also likely to share the same causes:

- **Vaporized design decisions**. In both cases, all or most of the original developers were either

no longer working on the system or had left the company entirely. Consequently, many of the design decisions taken early in the evolution of both systems were poorly understood. Particularly the maintenance of case 1 became more problematic after the person who designed this component left the company. In the other case, the designers were on a different continent than the people who were involved in the realization phase.

- **Too little attention to design during evolution**. During the evolution of a system, changes may occur that require that the software design is altered. In both cases, we found that little attention to the design was paid during the evolution. In case 1 several, major design changes had taken place during its evolution. The resulting software had become extremely complex. In case 2, time-pressure had caused developers to bypass the proper process for defect fixing (which includes documenting the changes and designing a fix).
- **Quick fixes**. During the evolution of a system, defects are found and fixed (in [8] this is called corrective maintenance). The proper way to fix a defect is to analyze the defect, design a solution, implement and test the solution. Unfortunately, time-pressure or cost considerations may prevent developers to properly follow this process. Often this results in quick fixes that addresses the issues but that may also introduce additional issues. Especially in case 2, it was identified that the existing process for processing change requests (which is the common way for fixing defects) had not always been followed. In case 1 the design was so out of date that developers did not bother to update it anymore.
- **Experience.** An interesting aspect in case 2 was that the development of the software was transferred a number of times. One of the first development groups was relatively inexperienced and consequently, the quality of their work was relatively low. The lack of experience with development and the internal development processes probably was an important reason for the problems that surfaced once the development was transferred to a more experienced team.
- **Time pressure.** In the two cases we examined, two components of the same software product. While the components of this product are developed separately, their development must be synchronized with the release cycles of the product. Consequently, if a particular change cannot be realized in the timeframe between two product releases, problems may arise. The time-pressure associated with these releases was an important factor in the initial development of case 2. In order to make the release, certain things were rushed an parts of the code were incomplete.

4.4 Resolution

Once it has been determined that a system is eroded, and once causes have been identified, an attempt can be made to repair the system and prevent further damage. The obvious things that can be done and that we have observed in both cases are:

- **Redevelopment**. Redevelopment of the software is often the only real option in fixing an eroded system. This approach was chosen in case 1. Interestingly this decision was taken based on an estimate of the cost of fixing all the known defects (about 100).
- **Restructuring**. The people working on case 2, on the other hand, chose to restructure the existing system and reserved a significant amount of time for it. As in case 1, this decision was based on a cost estimation.
- **Strong focus on design**. As pointed out earlier, the lack of up to date designs is usually one of the problems with eroded systems. In both cases, recovering/updating the designs was an integral part of the attempt to address the problems and key to the success of the whole operation.
- **Modularization and object orientation**. In both cases, the developers complained about the fact that the source code was in bad shape and that there were many dependencies between the various modules and components in the system. In both cases object oriented like mechanisms such as encapsulation, information hiding and delegation were applied to improve the structure of the system.
- **Take product release cycles into account.** As argued earlier, the development of individual subsystems, such as the two cases we are dealing with, must be synchronized with the product release cycles. Typically, changes are projected at a particular release and there is little room for delays. Consequently, it must be possible to make the necessary changes within that timeframe. If not, an option is to break down the work. This happened in case 1 where the new component was planned and delivered in two releases. In case 2, one of the problems was that the developers adopted some quick fixes in order to be able to integrate their software in the product in time for the product release.

4.5 Prevention

The developers of the systems we examined in this paper have experienced first hand what it takes to

recover a deteriorated system. Naturally, they made an effort to learn from the experience to adapt the way they develop software in such away that future problems can be avoided. In the cases we examined a number of practices were adopted that appear to be successful:

- **Automatic regression testing**. In order to prevent that new defects are introduced during defect fixing, automated tests can be used to verify that the system still works. Regressions were particularly a problem in case 1. Therefore, the developers adopted the practice of creating automatic tests while they were redeveloping their component. By the time this component was finished, a test suite of 800 tests was available. Also, the defect metrics showed that there were almost no regressions during the maintenance of the new system.
- **No undocumented fixes**. Both cases shared the problem that in the past there had been undocumented changes. This both makes it hard to test the software and to use it correctly (this was a problem in case 1). To address this, all changes are now documented properly. Also, in case 1, test cases are made to ensure that the software works as advertised in the documentation. Any deviation from the specified behavior is considered as a defect now.
- **Stronger focus on process**. Part of the problems in case 2, and to a lesser extent case 1, can be attributed to the fact that the existing development process was not enforced. This caused all sorts of problems the processes were designed to prevent.

5. Concluding remarks

In this position paper, we have briefly discussed the results of two case-studies. As discussed in the introduction, a full paper including details on how and where the results outlined here were obtained is pending.

An important conclusion of our earlier work was that design erosion is inevitable. Consequently, our case study did not focus on how to prevent design erosion but on effective strategies for dealing with design erosion. Both software systems in the two cases we discuss in this paper are part of a software product, which has existed in several versions. The company involved identified that there were problems with these subsystems and successfully addressed these issues without causing any delays in the product release schedule. In other words, the process of identifying and resolving design erosion works reasonably well in this company.

An interesting aspect of this case study is that, in addition to the technical factors identified in our earlier study [3], there are also a number of non-technical factors that contribute to design erosion. For example, in case 2, an important factor was that the existing development process was not enforced. Consequently, any measures for resolving or preventing design erosion also have to consider these non-technical factors.

Based on what we have seen in this case study and in other software systems, we are strengthened in our belief that design erosion is inevitable. Software developing organizations should not be judged by how effective they are in preventing design erosion but in how effective they are in identifying and resolving eroded software components.

In future work we will present more details about the case study presented here. Additionally, we intend to write a 'best practices' paper.

6. References

[1] V. Basili, "Editorial", Journal of Empirical Software Engineering, Vol 1. no. 2, 1996.

[2] V. Basili, F. E. McGarry, R. Pajerski, M. V. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Lab", proceedings of ICSE 2002, pp. 69-79, 2002.

[3] J. van Gurp, J. Bosch, "Design Erosion: Problems & Causes", Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

[4] C. B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", in Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), 1999.

[5] D. L. Parnas, "Software Aging", in Proceedings of ICSE 1994, 1994.

[6] D.E. Perry, A. L. Wolf, "Foundations for the Study of Software Architecture", in ACM SIGSOFT Software Engineering Notes, vol 17 no 4, 1992..

[7] C.B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering", IEEE Transactions of Software Engineering, 25(4), pp. 557-572, 1999.

[8] E. B. Swanson, "The dimensions of maintenance", proceedings of the 2nd international conference on software engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976.

Observations on automation in cross-platform migration

Ben Wilson, Tony Van der Beken
Anubex
Veldkant 35C, Kontich Belgium

E-mail: ben dot wilson at anubex dot com, tony dot vanderbeken at anubex dot com

Abstract

There are numerous ways for organisations to migrate an operational information system from one deployment platform to another. This paper relates a number of experiences of applying automated techniques to cross-platform migrations of larger (> .5MLOC) information systems in real world projects. The paper examines these experiences, considering factors influencing the organisations' decision for the approach, the project-specific features and limitations of the approach, and the effects of the approach on the organisational context. This paper does not attempt to provide an exhaustive comparison of the advantages and characteristics of the different approaches that may be used, but rather to consider a single approach in more detail, based on the experience of the authors. It is our position that the use of automated methods will increase as the risk-elimination effects of the technique will ensure its rise in popularity, and information systems increasingly outlive the platforms for which they were developed.

1. Introduction

Calculating the actual number of time-proven, mission-critical information systems with over a half million lines of program code currently in operation worldwide is a nearly impossible task. The term 'legacy' is often used to describe these systems, and with the term a number of negative and subjectively sensitive attributes typically spring to mind: ([3], [4], [9], [10]) hard to understand, insufficiently documented, difficult to maintain, and unintuitively structured are some of them. In the typical industry jargon, the term 'quality' ([5], [7], [8], [11]) is often quoted as a blanket term, and can be used to refer to a combination of any number of these attributes.

The notion of 'poor quality' can be used as an argument to convince the organisation owning the information systems into undertaking a revolutionary reengineering effort ([7], [8], [9]). Typically such an effort involves understanding a program's functionality (perhaps aided with a 'legacy understanding tool,' a 'business rules extraction tool,' or a 'code slicing tool'); storing this information in some form of a repository or representing the code in an easier-to-understand format or a modelling tool; and rewriting or generating new code with a 4GL or in a modern, object-oriented architecture such as J2EE or .NET.

There are, however, many mature systems for which not all or even none of the above attributes apply, and indeed the only observation that can be made is that the systems were built with leading-edge technology. And that the leading-edge technology is old. Host-based, monolithic, character-based systems on proprietary platforms that are hard to integrate are also called 'legacy systems.' Too often, however, simply because a system is built to run on technology in its teens, it gets stigmatised with the same subjectively sensitive, negative attribute of being 'difficult to maintain.'

Organisations who either lack the monetary resources to re-engineer or rewrite large amounts of code, or who see no business benefit in doing so, typically look to the alternative of replacing the system (or parts thereof) with COTS[1]. If no suitable COTS alternative can be found to replace the system's functionality (or the remaining parts thereof) then the organisation will be stuck with finding a solution that remains their own.

This, in a nutshell, is where the industry for cross-platform software migration tools lies. These tools serve to make the transition from endangered or undesired component technologies to the ones of the organisation's choosing optimally automated and cost-effective, simultaneously enabling the organisation to retain what they see as an asset, namely the functionality of their systems. While there are many variants, the better tools enable this simultaneous transition and retention without introducing runtimes foreign to the technology being implemented and proprietary to the tool vendor. Some vendors make this possible by coupling their tools with a service to generate a 100 % functionally identical and 100 % visually equivalent copy of the original system that furthermore retains its ease of maintenance.

Anubex is a Belgian IT company, and has specialised for the past ten years in building and deploying application transformation and migration tools. During this time, we have advised over fifty organisations on migration projects of larger (.5-10 MLOC) information systems, and built over

[1] COTS: Acronym for Commercial Off The Shelf software, or a packaged system. The 'approach' that this acronym refers to can be applied to the implementation of any packaged software, however, regardless whether it is publicly or commercially available.

forty tools that automate various aspects of software transformation for specific platforms and languages. In this paper, we relate, based on our experience, our view on how the overall perception of cross-platform migration is evolving, how automated translation works in the context of a manageable project, and how semi-automated transformation and migration techniques impact organisations and developers.

2. Definitions and scope

The discourse on software transformation and migration is made difficult by the lack of a clear consensus regarding the use of terms, sometimes intermingled, to describe, variously, the business goals, the technical deliverables, and the methodologies used. The terms 'legacy transformation,' 'legacy modernisation,' 'legacy renovation,' and 'legacy reengineering' are used to describe families of these 'approaches' in which redevelopment [3], re-writing [5], EAI ([1] [11]), retro-documentation (more used in French-speaking regions of the world) [6], replacement ([1], [5], [11]), migration ([3], [10]), consolidation [1], wrapping ([3][11]), web-enablement [1], re-use [5], screen-scraping [11], domain engineering [8], and componentisation [8] (to give a few common examples) fall.

The narrower term 'migration' also suffers from a similar lack of clarity. Migration can be either a business goal in itself or a technical deliverable of a larger business goal (for example, consolidation), and is furthermore embodied in multiple reengineering methodologies that rely on automation in different ways ([3], [10]). The term is also sometimes avoided, with synonyms such as retargeting, replatforming, and rehosting [7] being used.

For the purposes of this paper, we consider migration as the *restoration of value to a software application by removing its dependency on undesired technologies or architectures, through the conversion of the application's pieces from one technology to another, creating an otherwise identical working system that uses new technologies in a native way*. This approach makes use of platform or language-specific 'models' that represent the application before the migration and afterwards.

We define a platform-specific model as one where the *bi-directional transformation between it and the source code it represents can occur an infinite number of times, without the loss of any information*. Migration is automated, then, through the transformation of one platform-specific model into another. It can rely on the use of either one or two meta-models, traditionally referred to as grammars [2], depending on whether the migration of the legacy application involves a conversion of the code from one language to another or not. This approach has the characteristics of a black box, meaning that any manual alterations made to the code occur either before the application is parsed or after the migrated code is generated.

With the exception of any additional application tuning, which may be necessitated to ensure the retention of the performance of the system in its new environment, this is where migration, as used in this paper, stops. Many legacy strategies that bear the label 'migration' attempt to go further ([3], [10]), incorporating additional steps such as the restructuring of 'spaghetti' code, the re-architecturing of an application's entities, the manual development of GUI interfaces, or the full exploitation of object-orientation. While it is not our intention to ignore the value of such additional services, our observation is that the ROI argument to follow the shortest possible path to achieve a clearly defined business goal (in this case, the decommissioning of an undesired technology component) is growing in importance. This is especially the case as concerns decisions made for larger software applications, and the approach may be pursued even when the 'quality' of the code may be questionable.

The overwhelming majority of organisations we advise (100 % of them) consider automated migration for administrative software applications. These applications run the 'core business' of banks, insurance firms, government institutions, or services companies; or the 'back office' applications for companies in the aerospace, telecommunications, or manufacturing industries. Most of these organisations started developing the applications between fifteen and twenty years before their migration, and in all cases except for one, used COBOL. (The sole exception regarded a 3MLOC application written in BASIC for a European airline company.)

At a technical level, the transformation projects being pursued fall into one of two categories:

- Actual cross-platform migration from endangered or proprietary hardware platforms to 'open' distributed systems (mainly Unix or NT), or from endangered development environments to modern development tools and deployment platforms (mainly application servers such as WebSphere, etc.);
- Retargeting applications from data access methods that pre-date RDBMS (networked databases, hierarchical databases, or (index) sequential files) to an RDBMS product (mainly Oracle or DB2).

3. Justifying migration

Other studies ([4], [9]) have investigated common 'drivers' for software transformation projects. Some [5] have gone further to analyse these drivers according to their justification by internal considerations (such as cost reduction or guaranteed operational continuity) or external considerations (such as eBusiness initiatives or more strategic 'future-proofing' of the applications).

Our experience with COBOL-heavy environments in Europe suggests that migration or transformation projects are mostly justified by a combination of these predictably recurring 'drivers' together, but that in a quarter of the cases a single overriding factor is sufficient to justify the project in its entirety. Y2K compliance, obviously, has disappeared as a driver.

Perhaps surprisingly for a business context, the driver of 'cost reduction' is rarely used to justify a migration project on its own. Examples of cost reduction drivers include eliminating

administrative overheads and extra technical support costs for running processes over separate, non-integrated systems; eliminating the need for middleware to connect proprietary systems with open ones; reducing maintenance overheads by adopting cheaper hardware or development platforms; or other economies made through platform consolidation.

Examples of overriding, singular drivers that do get used to justify migrations are the following:

- The technologies used present a physical technical barrier in terms of performance, (storage) volume, or the maximum number of concurrent users. Migration is seen as urgent when these technical barriers prevent the business' natural growth;
- The technologies used by the application are outmoded and the organisation is pressured by its clients to modernise them. ISV's are of course especially vulnerable to these influences, but this driver has also been found in the B2B insurance and services sectors;
- The migration of the application is a necessary step in some other process. A common example involves organisations implementing an ERP package to replace business-generic functionality in a legacy application, and that need business-specific application functionality to be migrated in order to retain the integration of the processes and data. In these cases, migration makes the implementation of the ERP system possible;
- The supplier of the technology has announced the termination of support. This can involve both hardware and software suppliers.

Perhaps equally surprisingly, in none (0 %) of the cases have any of the following been used either as primary or supporting drivers:

- Pressure from clients or suppliers to integrate supply chains;
- eBusiness;
- Migrating away from COBOL to a 'more modern' programming language.

In well over half of the cases, organisations that have real migration needs do not initially consider automated migration as a potential solution. The most common reasons for this are the following:

- An unawareness of the availability of tools that cater for their requirements in terms of source and target technologies supported, or the belief that the creation of a tool that fits their specific environment requirements is not feasible or cost-effective;
- A belief that manual redevelopment of their systems from scratch in newer technologies is desirable or feasible, or resignation to the belief that the only cost-effective solution is to outsource the redevelopment offshore;
- The perception that a project which delivers a 100 % functionally identical piece of software "does not take the company forward;"
- A prior negative experience with a migration tool that generated unmaintainable code.

Over 50 % of attempts to migrate applications with over .5MLOC through manual redevelopment are abandoned after two to four years as failures.

4. Wanted? 100 % migration

The figure of 100 % is often referred to in the justification and planning phases of migration projects, and this in two cases:

- How to justify a project that creates a 100 % functionally identical target system;
- The evaluation of the quality of a migration tool by measuring how close to 100 % of the objects or language statements in the original technologies it migrates automatically.

Deliberately creating a target system that is 100 % functionally identical to the original system is sometimes seen as a counterintuitive milestone. This limitation nevertheless offers several benefits, all of which an organisation typically comes to recognise during project execution:

- First and foremost, perhaps, it is incontestable. When organisations undertake a tools-assisted migration of their software applications, they rarely use tools of their own making. Not having the expertise or experience to build the tools needed, they look to licensing existing ones from a tool vendor. Normally, the tool vendor provides the services that go with the tool and can be given the responsibility for guaranteeing the new system works. A discrepancy in behaviour or in output is easily demonstrated and gives the organisation procuring the service added protection;
- It is the only way to avoid the difficult, expensive, and time-consuming process of making specifications; it prevents the danger of scope creep; and it facilitates the testing phases of the project;
- It gives the organisation a clear, easy-to-understand means of tracking the progress of the project;
- It relieves the strain on users to adapt to the new system and limits the entire change management process to the IT department;
- It is the fastest way for an organisation to transition from old technology to new, and it is the fastest way for the organisation to regain autonomous control and resume normal incremental maintenance activities for the system.

5. Migration Complexity

Regardless which life extending approach an organisation pursues for its applications, at some point 'analysis' takes place. For the sake of simplicity we will limit the discussion to the approaches of continued incremental maintenance (the 'do nothing' approach), replacement with COTS, re-engineering/re-writing, or automated migration. The exception to the analysis rule involves certain language, platform, or presentation extension technologies such as emulators, wrappers, or (cross) compilers that serve at the same time as a means and an end.

With the exception of migration, all of the strategies listed above have in common that analysis is functionality-driven. With COTS, a 'gap analysis' may be performed to highlight original application functionality that is not supported in the commercial product, and other analyses can be performed to plan change management when internal processes of the organisation have to be revised before the commercial software package can be used. With re-engineering and re-writing, code complexity, similarity, and redundancy can be analysed in addition to the functionality of the application.

Analysis plays an important role in the planning of a migration project too, however this analysis is not driven by a need to understand either existing or intended functionality. The focus of migration is in code and object translation, and is predominantly a purely technical exercise. Because of this, an understanding of the code's functional purpose is not needed.

Such analysis can be automated or done manually. Some tool vendors supply analysis tools as a companion part of their toolkits, which automate the process and provide a more mature solution. These analysis tools, much like the tools that do the actual migration, extract the information they need from the code itself. While there are many variants, a common denominator is the extraction of information pertaining to the size of the applications and their complexity. Vendors use this information to forecast the amount of work that the migration effort will entail and to draw up project plans. From a commercial perspective, vendors may also use this information to calculate the licence price that the migrating organisation must pay to use the migration tool.

Bearing the above in mind, automated migration is an exceptional part of application development for three reasons. First, since the calculation of the complexity and the number of lines of code in the original system is sufficient to calculate the effort required in terms of man-days, automated migration is exceptional since it uses a predictive LOC metric with accuracy. Second, automated migration is exceptional since functional or business analysis is not used to predict man-days of programming effort and function point analysis does not offer the project any direct benefits. And third, due to the purely technical nature of the exercise, automated migration is exceptional since the effort required in terms of man-days does not accelerate in function of the size of the applications, and as our experience shows, in some cases even decelerates.

The notion of complexity also warrants further clarification, since complexity in this context is not calculated on the basis of the relationships of the lines of code to each other, or from the code's structure or lack thereof. Complexity in migrations is an *indication of the number of occurrences in the source code of the original system where a statement or object does not have a one-to-one equivalent in the target technology*. These occurrences can be simple (for instance, a variable name used in the original application is a reserved word on the target environment) or complex.

An example of a frequently recurring, COBOL-related, cross-platform incompatibility of a high complexity involves the data access methods that are used on most legacy platforms. While most basic data types like numeric or alphanumeric are prevalent and equivalent in both legacy and modern platforms, composite data types often pose difficulties. COBOL makes it possible to access and store data at the record level through powerful low-level pointers, and many COBOL programs make use of this facility to store data in a single file with the individual data elements organised inconsistently. Modern RDBMS products restrict data access to the field level, and manage the structure of records so that each is guaranteed to have a consistent organisation. When a COBOL application uses a REDEFINES clause in a File Description, the lack of a one-to-one equivalent in the target RDBMS environment prevents an automated translation of the statement. Individual instances of this cross-platform incompatibility can sometimes be dealt with fully automatically. However, when a record is redefined with incompatible data types (for instance, to store alphanumeric data at a position in a record where previously numeric data was stored) the translation must be manually prepared before the automated translation process can continue.

The issue of cross-platform incompatibility leads to the question of whether it is possible to create perfect tools that automate the migration of 100 % of the objects and language statements in the original technologies. It is perhaps good to mention at this point that such ambitions are hardly the holy grail of the automated migration industry. Typically, averages of 95-99 % are achieved, and it is worth mentioning that a tool, compatible with 95 % of the objects and language statements of a development environment could automatically migrate 100 % of one application and only 80 % of another. At the same time, it is dangerous to compare tools on the basis of coverage percentages only. Certain cross-platform incompatibilities can be solved automatically with ease, but the solution may come at the cost of being very difficult to maintain. Except in circumstances where the target technologies have been built specifically with backwards-compatibility in mind, the likelihood of being able to automate the migration process of 100 % of the objects and language statements, and at the same time generate code that is easily maintained, is low to non-existent.

From the discussion on complexity and cross-platform incompatibility, it should be clear as well that the level of automation has a direct impact on cost. This is for two reasons: less automated translation means more man-hours to implement manual solutions; and less automated translation also means more time is spent on testing as humans tend to make more mistakes than software.

But how important is this factor, and how does this weigh against the drive of organisations to embrace 'more modern' languages and development environments? Surely a language such as Java is more modern than COBOL, but at the same time surely a COBOL-to-COBOL migration is cheaper than a COBOL migration coupled with a language conversion to Java, since there are more incompatibilities between the two languages? And surely there must be a perception that an application written in Java is better 'future-proofed' than one in COBOL, but how does this weigh against the notion that the Java program will be less intuitive for the original developers to maintain than if it were kept in COBOL, due to structural changes made to the code?

Actually, none of the organisations we have dealt with have pursued the automated migration of a large-scale information system with a language conversion from COBOL to Java. However, organisations that we have dealt with who pursue a migration of COBOL applications to, say, Oracle database technology have a choice, since a variety of tools are available that perform COBOL-to-COBOL retargeting, COBOL-to-PL/SQL conversion and retargeting, and COBOL-to-Java conversion and retargeting. When migrating COBOL applications from legacy platforms to an Oracle database, and given the choice to keep the applications in COBOL or to convert them to PL/SQL or Java, 95 % of the organisations opted to keep the applications in COBOL, and the other 5 % chose to convert to PL/SQL.

How each justified their decisions is also something of a surprise. The majority who chose to keep the COBOL did so out of cost considerations. The minority who chose conversion to the 'more modern' language, on the other hand, also did so out of cost considerations.

The cost argument used by the majority of organisations opting for COBOL-to-COBOL migration was that the cost to retrain teams of COBOL developers to the 'more modern' language was greater than the potential 'future proofing' benefits of having the code in PL/SQL or Java. Coupled with the fact that most organisations had other COBOL developers in their employ who worked on other applications, these organisations took this decision with the certainty that the number of available COBOL developers, for the time being at least, was higher than the number of available PL/SQL and Java developers. The cost argument used by the minority was a licensing issue, in which paying for COBOL runtime licences for the hundreds or thousands of users of the system was higher than the cost to retrain the COBOL developers.

This evidence suggests that organisations with large-scale software applications are not capricious with their migration choices, and stresses that the business angle weighs heavily in the major investment decisions taken around them.

6. Project dynamics and fluid systems

The approach to migration as explored in this paper is able to reduce or even eliminate many project elements that involve users, such as user retraining or the analysis of user requirements. Such economies are of course inherent to the deliberate limitations of the approach, which actively seeks to ignore these and a number of other issues.

On the other hand, it is rare to find a situation in which the issues, ignored by the approach, do not surface at some point during the project. This problem introduces a new issue that is common to any approach taken to migration, and involves the way in which it can balance the need for a system 'freeze' with the need of the existing system to evolve freely during the project's course.

In accordance with the strict enforcement of the 100 % equivalence rule, all modifications must be done on the existing system in production. In this case, the application of the rule protects the tool vendor, since any modification of the functionality pursued by the migrating organisation must be proven to work on the original system prior to being taken into consideration. Through the 100 % equivalence rule, then, business disruption is not only minimized as concerns the operational context of the system's use, but also as concerns its evolutionary maintenance throughout the transitory period.

To put the problem into perspective, it is necessary to consider two facts: First, the duration of an average migration project for applications as treated here is between five and six months. Second, despite organisations recognising the complications that modifications to the existing systems will introduce to the migration project, it has been necessary in 100 % of the cases for the original system to be modified at least once while migration is in progress. Such changes can be necessitated by law or by regulation; or can be warranted by other business needs.

The need for the original system to evolve freely during the course of a project makes it impossible for all practical purposes to impose any form of freeze on the code. The only freeze that does takes place regards the system in its totality, and is limited to the very last stage of the project in which the final conversion of the data from the old environment to the new one takes place. This phase normally takes place during a weekend when the system is otherwise not in use. Since 24/7 system availability has been necessary in 0 % of the cases, such a freeze has not been the cause of business disruption, and the overhead of implementing of a real-time switchover mechanism, while possible, has not yet been justified.

When assessing the impact of a modification on the existing system during the migration, there are two parameters that are the most important to consider. These are, first, whether or not the source concerned has been converted AND manual work has been done on the converted source; and second, whether or not the modification involves a change to the data structure.

The simplest scenario is if the modification does not involve a change in the data structure and no manual work has happened yet on the converted source. In this case, the source is merely converted again.

The scenario with a slightly higher complexity occurs when the modification does not involve a change to the data structure, but manual work has already been done on the converted source. This scenario introduces a version conflict, as illustrated in the figure below:

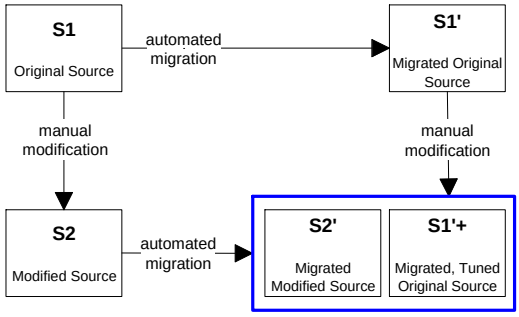


Figure 1: The version conflict in the migration of fluid systems

As shown in Figure 1, source S1 is converted to the new environment and manual work has been done, resulting in a production-ready candidate S1'+. When modifications are subsequently made to S1 on the original system, resulting in S2, the creation of S2'+ must result from the comparison of the differences between S1'+ and S2'. Very often, the modifications do not affect one another and S2'+ can be the result of the straightforward merge between S1'+ and S2'. In other cases, additional manual work and testing must be done.

Scenarios that involve the modification of the original system's data structure are significantly more complex. This is largely due to the method inherent to this form of migration, which combines incremental elements of 'chicken little' [4] during the construction and testing of the new system with a 'big bang' in the event of going live. As a result, the 'test data' being used in the tuning and testing of the migrated programs prior to going live plays an important role in the migration process, and the definition of the test data environment must always be kept up to date. For this reason, the creation of the test data environment is always one of the first steps done in any project.

When the data structure in the original system is modified, this implies that the test data on the target system together with the data dictionary and DDL statements must be updated. Any application sources that are affected by the change must be reconverted. This process can be the cause of numerous version conflicts, as depicted in Figure 1.

7. Migration impacts

Even when bearing in mind that the goal of migration is the creation of a 100 % functionally identical target system, and that doing so benefits the organisation since change management is limited to the IT department, change management in the IT department can be heavy nevertheless. While migration of course impacts the IT department, it is the transition to a new environment that causes the most disruption, and it is the automated migration approach that actually minimises the extent of it.

Or at least, it can. Our experience shows a clear correlation between the level of direct involvement of the organisation in the migration project and their overall level of satisfaction with the project's final outcome. This factor persists in all projects, and is not influenced by the involvement of third-party migration service providers. This factor is especially pronounced when the maintainers of the system play an active role in the performing of manual work. Maintainers who get involved are more autonomous and confident in their abilities to resume maintenance over the new system once the project is finished.

In environments where large, 15-20 year-old applications are maintained in-house, developers typically posses three critical competencies:

- A knowledge of the business;
- A knowledge of the application code and its structure;
- A knowledge of the development and deployment technologies used.

Armed with these three competencies, developers have the capacity to support the applications that support or enable the business. Automated migration makes it possible to economise and retain the first two of these, with both being actively used both during the course of the project and afterwards. As concerns the last point, the transition to new technologies can cause disruption since new skills must be acquired. There is rarely the luxury of time, since the migration projects of the organisations we advise normally take up to six months to complete.

Automated transformation and migration is arguably the best way for maintainers to acquire new skills and adapt to new technologies, and although organisations do not always see the benefits initially, real-world examples confirm it to be so. Some developer-related benefits of migration are the following:

- Since the bulk of the code is converted by a piece of software, the code is translated and generated consistently. This relates not only to code formatting conventions such as capitalisation and indentation, but also to the consistent translation of the statements the maintainers are already familiar with and the retention of comments. Subsequent application maintenance is easier since the code still 'belongs' to the developers;
- Learning the new environment is easier since developers can compare the code 'before and after;'
- The retraining of developers is never on the 'critical path' of the project, and developer retraining is never rushed as a result of it being a prerequisite for the project to begin, as is the case in fully manual redevelopment projects;
- Libraries and languages are pre-deployed by the migration tool. Through this, developers do not have to achieve reasonable professional proficiency in the target environment, and then go through a difficult process of agreeing on a development 'house standard' in the new technology before the project can start;
- By working with a tools vendor with extensive experience in both the source and target technologies, training materials and programs can be tailored to take advantage of the skills the developers already posses. Such targeted training is normally impossible to find externally. Our experience shows that training time of developers can be reduced by up to 50 % in comparison to following standard, vendor-approved, entry-level courses when the training can be tailored in this way.

The most lasting impact on the organisation of a migration project, of course, is that an application's anticipated lifespan is doubled, and that the applications preserved get a new lease of life. Especially if migration is partial and business-generic parts of the original legacy system are replaced by COTS, the migration to new technology of core applications can bring an organisation's appreciation of their uniqueness as a business into sharper relief.

This added realisation can impact the way that subsequent maintenance decisions of the system are handled, and mark the transition of business-specific application functionality to a new level of maturity in which an organisation makes a conscious choice to continue investing in its growth for many

years to come. The awkward position the system occupied prior to the migration as being simultaneously a business-enabling asset and a technical liability is thankfully put in the past.

Conclusion

In this paper, we have examined automated migration from a number of different angles, exploring the utility of the approach through the limitations that serve as its defining characteristics. The choice to deliberately limit the target system to feature 100 % functional and visual equivalence as a deliberate milestone on the road to ultimate modernisation is perhaps the most prominent feature of this approach. When applied consistently, this limitation can affect, as explained here, the way in which organisations justify implementing the approach, the relationship between complexity and the cost of the approach, the ability of systems to evolve freely during the transitory period, and the way in which the organisational context is impacted through the approach's application.

The fact that many organisations opt for an approach that deliberately limits the scope of the project deliverables opens a number of potential relatively unexplored research directions. It is our opinion that research is lacking, which considers the viability of achieving modernisation for larger software systems in terms of sequentially applied steps that are applied to a system on the whole. Such research has the potential to benefit organisations that are under pressure to achieve clearly understood business objectives and realise ROI in increasingly shorter timeframes.

Bibliography

[1] Aberdeen Group: Legacy Applications: From Cost Management to Transformation, March 2003.

[2] J Bézivin, S Gérard: A preliminary identification of MDA components. Date unknown.

[3] J Bisbal, D Lawless, B Wu, J Grimson: Legacy Information System Migration: A Brief Review of Problems, Solutions, and Research Issues. Technical Report TCD-CS-1999-38, Computer Science Department, Trinity College Dublin, May 1999

[4] M Brodie, M Stonebraker: Migrating Legacy Systems, Morgan Kaufmann Publishers, Inc. 1995.

[5] D Good: Legacy Transformation, Club de Investigación Tecnológica, August 2002.

[6] D Good: Proceedings of the Club de Investigación Tecnológica Legacy Transformation Workshop, San Jose Costa Rica. February 2003. Available at www.cit.co.cr/Presentations/DeclanGood.ppt

[7] M Olsem: STSC Reengineering Technology Report, Document No: TRF-RE-9510-000.04, Software Technology Support Center. October 1995.

[8] A van Deursen, B Elsinga, P Klint, R Tolido: From Legacy to Component: Software Renovation in Three Steps, Cap Gemini and the CWI. 2000

[9] I Warren: The Renaissance of Legacy Systems, Method Support for Software System Evolution, Springer Verlag.1999.

[10] B Wu, D Lawless, J Bisbal, R Richardson, J Grimson, V Wade, D O'Sullivan: The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems. Proceedings of the third IEEE Conference on Engineering of Complex Computer Systems (ICECCS97), September 8-12, 1997. pp.200-205, IEEE Computer Society.

[11] F Zoufaly: Issues and Challenges Facing Legacy Systems. November 1, 2002. Available at http://www.developer.com/mgmt/article.php/1492531.

Evolution of legacy systems : strategic and technological issues, based on a case study

Herman Tromp and Ghislain Hoffman
Department of Information Technology
Ghent University, Belgium
Herman.Tromp@UGent.be, Ghislain.Hoffman@UGent.be,

Abstract

The goal of this experience report is to highlight the strategic and managerial issues that are likely to be involved in any migration or evolution effort of large-scale legacy systems. The report is based on a specific case study of a large organisation in the Belgian health care and social security system. Major drivers for a legacy evolution effort are identified. Emphasis is put on the required management and planning view, rather than on the mere technological issues. Their constituent elements are discussed in some detail.

Introduction

In a companion paper [1] some first findings of the ARRIBA research project are described and directions for future research are outlined. While ARRIBA tries to define the more technological aspects of legacy mining, knowledge extraction and revitalisation in a generic and long term way, in this paper we report in more detail on specific experiences of a specific case. In [1] only a summary description of this case is presented, besides some other cases. We will also discuss the strategic and managerial aspects, rather than limiting the scope to pure IT issues.

While ARRIBA focuses on research towards techniques for reverse engineering and architectural knowledge extraction, the case we present here is mainly driven by the need to revitalise an IT infrastructure that is crucial for the organisation and is driven by the need to further evolve a large scale legacy application. Some of the most obvious drivers behind that need will be explained below, but it is important to note that they are widely different in nature. They include technology, economy of scale, strategy and market position.

This experience report is based on a spin-off project of the ARRIBA project. This project is a bilateral cooperation between the Department of Information Technology, Ghent University and the LCM ("Landsbond Christelijke Mutualiteiten"). The latter is the largest independent organisation, which is, as a part of the Belgian Social Security system, responsible for providing the redistribution of health care insurance allowances, both towards individuals and hospitals etc. Besides their legally regulated core mission, they also offer a number of welfare related services to their members. It should be noted that their operational and legal context is typically Belgian, which implies that no COTS software can be found on the international market to support their core business. Even if a standard ERP package is installed, it offers only a partial solution, for example in basic accounting operations. Particular legal requirements exclude the use of standard packages to cover all of them, mainly due to fundamental differences in information models and business processes.

Although the case study is about a large-scale migration effort, the best approach, as will be explained below, appears to be an evolutionary. The managerial problems associated with that approach will be discussed in some more detail further in this paper.

For reasons of confidentiality, some details in this experience report are made somewhat more generic, but the conclusions remain sufficiently based on real experience to be relevant for this workshop.

Drivers for legacy evolution

Several alternative definitions of what exactly a legacy system is can be envisaged (see for example [6] [7]). In those definitions, often the notion of "something valuable" is present, as well as the notion of "old, obsolete". It is clear that legacy systems are crucial for the operation of organisations, which are essential for our economical and welfare activities. If we want to identify, however, what the drivers and the needs for evolution are, we prefer to use a more pessimistic definition:

A legacy system is an operational system that has been designed, implemented and installed in a radically different environment than that imposed by the current IT strategy.

A careful analysis of the above definition allowed us to identify a number of most evident drivers for having the system migrate and evolve. A number of them are listed below and illustrated by the case.

- The corporate strategy gets redefined, e.g. from a traditional data processing model to a multi-channel, service oriented model. This goes together with the requirement to have up-to-date data, coming from multiple sources, online all the time. It must be noted that currently, data are often replicated (daily, weekly) at local offices, for historical reasons (lack of sufficiently performing communication infrastructure, unclear definition of data ownership, etc.). Historical data are often only available off-line, e.g. archived on tape.
- Legal requirements and regulations in Belgian health care insurance change often, and those changes hardly ever take into account the IT system characteristics. As an example of this, new legislation requires the use of archival data, which is currently only available on magnetic tape, in order to impose a limit on the maximum health care cost per household (a notion which is not strictly defined, by the way), depending on their total income, requiring an interface with the taxation services.
- Business processes are redefined when management and business structure is reorganised
- The total cost of ownership of current systems becomes prohibitive, due to the diversity of the systems and the cost of software maintenance. On top of that, due to a growing business volume and the data processing model used, performance becomes increasingly an issue, raising the question whether to invest either in more powerful, but expensive hardware or to migrate to a new hardware/software platform with a larger evolutionary capacity.
- In the case we were also confronted with an outright end-of-life situation (no proper data base system, phasing out of a line of hardware and system software, etc.), resulting in an absolute need to migrate to a new platform. It should be noted, however, that the timeframe in such a situation is still a few years, but not very much longer
- Obstacles for migration can clearly be identified
    - A corporate information model does not exist. That model is deeply hidden in a proprietary flat file system: essential corporate data are stored in a single file, which is accessed through a wrapper. That wrapper performs maps the logical view on the data to their physical structure, but also accounts low-level tasks such as hashing, garbage collection, memory mapping, etc. The original developer of that "data structure" has retired, which makes it extremely hard to recover the data model.
    - There is no well defined IT architecture

Other factors hindering evolution, as already mentioned in [1], were also found in this case. Just to mention a few:

- The predominance of COBOL code, which has severe implications
    - Knowledge of the code is getting lost (the experts are retiring)
    - Recent techniques for software reengineering [5] are often based on object oriented languages and it is not clear yet how they can be fully used in this environment
- The project driven nature of the development efforts in the organisation often prevents a uniform, organisation-wide view

- Recent technologies, based on standard ERP packages [11], EAI techniques [8] [9] [10] [12], data warehousing, etc are often not very well understood. This is mainly due to a significant gap between the business view on organisation needs on one hand, and the IT infrastructure on the other hand.

A feasible migration strategy and a management view should seriously address the above. In the next section, we will concentrate on those issues.

Management view and plans

In order to support a revitalisation effort in a large organisation, management must clearly formulate answers to the following questions

- What is the motivation for the effort? (Why do we do it?)
- What are the objectives of the migration effort? (Where do we want to get?)
- What are the basic postulates and constraints? (What do the environment and previous management decisions impose?)
- How do we measure success, both on the road and at the end? (What are the critical success factors?)
- What are the risks? How to assess and address those risks?
- What methodology do we adhere to? (How do we get there in a systematic way?)

In the following subsections, we will discuss those issues more specifically for the case at hand.

Motivation

A concise motivation is required to get the stakeholders' and management buy-in. In the case at hand, the following elements are at play:

- A technology drive is present, but should not be overestimated. The main issue with this respect is to make sure that the organisation develops and maintains a *strategic* technology, while keeping aligned with industry development, and employs the selected technologies appropriate to reach its long-term objectives
- The business driven motivation is the aim to remain the national leader in health care and related social services
- The current hardware platform (BS2000) is reaching the end of its useful lifecycle and must be replaced anyhow
- Migration involves some risks, but appears to be essential and inevitable for future evolution
- The strategic decision to start with a "REFAC" project (Reorganisation of the Financial, Administrative and Control Circuits) implies a complete revision of the health care information system and is the basis for a reengineering effort
- Migration to a new environment should reduce support requirements, enable faster response times in development needs and allow to reassign IT staff to support emerging new technologies
- Viewing data and information as an institutional asset will improve the quality of (management) reporting and allow staff to respond easily to rapidly expanding needs for information, as well as provide a service oriented environment.
- In this view, it is felt that a more structured one should replace the present proprietary data infrastructure. A relational data model seems appropriate, but performance remains a major issue and this could impose constraints on the feasibility of deploying a relational database system.

It should be clear that the above list is a mixture of technological, management and strategic issues. As an expression of interest and motivation, they provide a clear and necessary commitment from

management, which, by the way, consists mainly of non-IT professionals but rather of medically trained people.

Objectives

In this section of any master plan for migration and evolution, the desired outcome and objectives must be formulated, based on both the "motivation" section but taking into account the environment, as described in the "postulates and constraints" section.

In the case at hand, the objectives were phrased as follows:

- Set up a migration path from the present operating platform (BS2000) to a new one. This is not only a major technological challenge, but requires also a profound economical analysis.
- Define a new technical architecture to be developed and installed. Several options are open for consideration.
- Create an efficient and flexible application and data architecture, which is felt to be lacking currently
- Accompany this with a budgetary, human resources management and business reengineering plan
- Make the necessary budgets available. This point depends heavily on management and stakeholders' buy-in, as defined in the "Motivations" section

The above list is definitely not exhaustive, but turned out to be both sufficiently concise and elaborate to convince the stakeholders, including the users, that their interests are best served.

Postulates and constraints

This section lists constraints imposed by the present environment and by previous management decisions. It should be noted that a previous reengineering project has failed, at a considerable expense, so a clear understanding of those constraints must be stated. Some of the previous decisions can definitely be argued, but license costs of software packages already incurred must be given consideration in the final cost/benefit analysis.

- A big bang migration is not feasible, because of ongoing operational requirements and obligations.
- Migration must be gradual. This implies that more is needed than a mere refactoring exercise, since data migration, synchronisation and consistency are major issues
- In every migration step, data must remain synchronised. This involves inevitably some degree of replication in intermediate stages, but finally it should be avoided. A central data provider must be present, instead of replicating data asynchronously to local offices.
- A business process reengineering exercise is going on and it is not very clear how this can be synchronised with IT migration
- A corporate information model must be developed. In [2] is described how such a model can be extracted from existing persistent data structures *and* from COBOL code, but in this case much of the information cannot easily obtained from COBOL record structures and is embedded in the executable code. The central role of a corporate information model within an organisation is explained in [3].
- This corporate information model should be based on an existing relational database product, which was chosen a few years before, for mainly commercial reasons
- Real-time and on-line data access (24 x 7) becomes a stringent requirement for the future and service oriented environment
- A J2EE based thin client architecture through portal and application servers is deemed to be most appropriate, but few experience (estimated at 5 over 200 traditional COBOL developers) is available. This turns out to be a major challenge.

- Integration with newly developed customer relationship management and financial systems (already based on newer technologies) is required

All these put severe constraints on the feasibility of possible migration paths.

Critical success factors (CSF)

The success of an evolution effort has to be measurable. Therefore, a number of critical success factors (CSF's) must be defined. Those should not only be used to evaluate the final outcome, but should be used along the road to measure progress and to define decision points in the plan, where go/no-go decisions should be made.

In this particular case, the following CSF's were identified:

- An affordable but significant proof of concept must be delivered within a reasonable amount of time and effort (typically 4 months)
- Along the whole roadmap, quick wins must be identified to validate the migrating system, and those should offer a measurable business benefit
- A strategic implementation of a strategic application suite within a time-frame of two years is essential
- Support from the stakeholders must continuously be ensured
    - From management
    - From users, both corporate and individual

This list could easily be enlarged, but on the other hand reflects what was mentioned in the "Motivations" section.

Risk assessment

A number of risks are definitely present in a large-scale migration plan, and must clearly be identified in order to be controllable. A list of risks is indispensable in any plan, but is certainly preliminary. New risks are likely to pop up.

Possible risks in this case are:

- Unfeasibility. Before a proof of concept or even a pilot application is delivered, it remains uncertain whether the combination and integration of traditional and "new" technology is feasible. The difficulties involved in controlling transactional context in a mixed environment are, for example, discussed in [4].
- Especially the interaction between a COBOL environment and a Java environment is a major technological risk factor. See also [4].
- Complexity. Systems may tend to be overly complex, and complexity is a source of unmaintainability. Integration of systems of a different nature does not relieve this risk. Componentisation and loose coupling between constituents might bring a solution, but well-understood solutions are not available yet.
- Performance is already a problem in the "as-is" situation, mainly because of the data processing mode used. To control costs and guarantee operational flexibility, it remains a major issue in the new environment, especially while transitioning.
- Obsolescence of techniques. Even "new " technology tends to be obsolescent before it even matures.
- Lack of pragmatism, taking into account the current situation in terms of expertise, resources and technology available. Solutions have to be pragmatic. Solutions offered by academics often fail in this respect – we must admit that. We should clearly attend to what was presented before in the "Postulates and constraints" section.

- Unmanageable systems. This point is related to “complexity” as mentioned before. What is overly complex is also unmanageable. There is also a human resource issue here: do we have the people and staff to manage those environments?
- Cost control. This point might seem to be obvious. but cost estimates in legacy evolution are hard to obtain.
- Lack of appropriate (human) resources, both within the organisation as on the local market. This point was mentioned earlier. It is clear that finding the right people and skills is a major impediment.

In the present case, the risks were carefully evaluated and fallback positions were defined.

Methodology

A methodological action plan is required.

In the case at hand, a number of tracks were defined to plan the required actions. These are

1. Evaluation of the “as-is” situation in the current mainframe environment. Relevant action items are among others: evaluation of the existing COBOL code and underlying data structures
2. Determine the “to-be” business and application architecture
3. Determine the “to-be” technical and deployment architecture
4. Critical evaluation of the migration scenario’s between the “as-is” and “to-be” situations

The careful development of this methodology allowed (and still is allowing) the streamlining of the migration path. Although the division between the tracks seems, at least at first sight, to be rather artificial, it partially reflects the structure of the different teams in place and so it is a consequence of the organisational structures, which cannot be ignored.

Technological migration issues and solutions

Based on the above managerial and tactical considerations, several technical migration scenarios are examined at the moment. For company-confidential reasons, those cannot fully be exposed right now, but that can very soon be remedied and result in a more expanded version of this position paper. underpinned by facts and figures.
Possible solutions to the evolution and migration plan will definitely have to rely on a temporary and controlled form of mirroring of data between the current mainframe platform and the relational database to be deployed.

Conclusion

Any effort for migration or evolution of large-scale industrial software systems must be driven by a corporate strategy redefinition. Buy-in from management and end users requires a well-defined strategic management view and planning, based on a clear statement of motivation, objectives and of the constraints imposed by the current environment and continuous operational needs. The definition of measurable success factors and a precise risk assessment are also essential. An organization-wide methodology must be defined and supported by all stakeholders. As a final note, it should be pointed out that the case on which this paper is based is a not- profit organisation. It is likely that in a more commercial context, a more detailed cost-benefit analysis would be needed.

References

[1] I. Michiels, D. Deridder, H. Tromp, A. Zaidman, "Identifying ICT problems in legacy software: preliminary findings of the ARRIBA project", this workshop.

[2] J. Henrard, J-M. Hick, P. Thiran, J-L. Hainaut, "Strategies for data reengineering", Proc. WCRE02, IEEE Computer Society Press, 2002

[3] K.Vandenborre, P. Heinckiens, G. Hoffman, H. Tromp, "Coherent Enterprise Modelling in Practice", 13th European-Japanese Conference in information modelling and knowledge bases", Kitakyushu, Japan, 2003.

[4] D. Plakosh, S. Comella-Dorda, G.A. Lewis, P.R.H. Place, R.C.Seacord, "Maintaining transactional context: a model problem", Report CMU/SEI-2001-TR-012, Carnegie Mellon Software Engineering Institute, August 2001

[5] S. Ducasse, S.Demeyer and O.Nierstrasz, "Object-Oriented Reengineering Patterns", Morgan Kaufmann and Dpunkt, 2002.

[6] M.L. Brodie and M. Stonebraker, "Migrating Legacy Systems – Gateways, Interfaces and the Incremental Approach", Morgan Kaufmann Publishers, 1995.

[7] Aberdeen Group, "Legacy Applications: from cost management to transformation", Executive White Paper, March 2003, at http://www.aberdeen.com/2001/research:03038126.asp

[8] D.S.Linthicum, "Enterprise Application Integration", Addison-Wesley, 1999

[9] J.C.Lutz, "EAI Architecture patterns", EAI Journal, March 2000

[10] M. Themistocleous and Z. Irani, "Evaluating and adopting application integration: the case of a multinational petroleum company", Proc. 35th Hawaii International conference on system sciences, 2002.

[11] M. Themistocleous, Z. Irani, R.M. O'Keefe and R. Paul, "ERP problems and application integration: an empirical survey", Proc. 34th Hawaii International Conference on system sciences, 2001.

[12] M. Fowler, "Patterns of Enterprise Application Integration", Addison-Wesley Signature Series, 2002.

Supporting Software Maintenance and Evolution with Intentional source-code Views[1]

Kim Mens and Bernard Poll

Département d'Ingénierie Informatique, Université catholique de Louvain
Place Sainte-Barbe 2, B-1348 Louvain-la-Neuve, Belgium
E-mail: {Kim.Mens@, poll@student.}info.ucl.ac.be

Abstract. We propose the abstraction of *intentional source-code views* to codify essential information, about the architecture and implementation of a software system, that an engineer needs to better understand, maintain and evolve the system. We report on some experiments that investigate the usefulness of intentional source-code views in a variety of software maintenance, evolution and reengineering tasks, and present the results of these experiments in a pattern-like format.

1. Introduction

"A program that is used in a real world environment necessarily must change or become less useful in that environment." [Leh84]

Software systems are constantly being enhanced and adapted to accommodate to changes in their environment. Several studies have proven software maintenance and evolution to account for a large part of the total software life cycle cost, making software maintainability a major commercial and economic factor to deal with in the development of a software system.

Our research hypothesis is that many software maintenance and evolution problems are directly or indirectly caused by a documentation problem. Software documentation most often does not capture all information a software engineer needs to understand, maintain or evolve a software system: important implementation choices that were made; the original programmers' intentions; how the code maps to higher-level architectural descriptions; interactions between methods, classes and modules; specifications of each piece of code; and so on.

A lot of this information remains hidden in the engineers' minds. Some of it may be recovered by analysing the code and its comments, but often it cannot be retrieved at all. In addition, browsing the code to understand its underlying intentions or to reverse engineer its effective architecture is a non-trivial and time consuming process and generally produces an incomplete mental picture of the software only. When evolving the software, such an incomplete understanding of a software system can lead an engineer to unnecessarily increase its complexity, or to introduce undesired and erroneous inconsistencies.

In [MMW02a] we introduced the abstraction of *intentional source-code views* as a way to *"increase our ability to understand, modularise and browse the source code by grouping together source-code entities that address the same concern."* We claim that this abstraction can be used to capture much of the information about a software system's architecture and implementation that an engineer needs to better maintain and evolve a software system. This is mainly because the views are defined *intentionally*, i.e. as a logic query on the source code, rather than by explicitly enumerating all source-code entities involved. As such, an intentional source-code view's description typically does not need to be changed when the source code evolves. In addition, *extensional consistency* of such views, i.e. the fact that two alternative intentional descriptions of the same view should always produce exactly the same extension set, is proposed as a way of maintaining source code consistency.

We conducted some preliminary experiments to investigate how intentional source-code views can codify information that is essential to software maintainers and evolvers and to document, in a pattern-like format, how they can use this information in a variety of software maintenance, evolution and reengineering tasks. We will present two evolution-related usage patterns: *enforcing coding conventions* and *checking design consistency*.

This research is a step towards developing a technique and tool that, *without drastically changing the way in which software engineers work*, can:

- help engineers understand a software's architecture;
- contribute to keep an up-to-date software documentation;

[1] This position paper is a shortened version – though customized to the topic of the ELISA workshop – of the full paper [MPG03] that was accepted for publication and presentation at the main conference track (ICSM 2003). It was also submitted – with minor differences only – to the ECOOP 2003 workshop on Object-Oriented Reengineering, where it was published in a technical report [DDM03].

- improve developers' and maintainers' efficiency;
- help engineers take implementation choices;
- help engineers to conform to an established architecture;
- avoid an evolving software's architecture to become unnecessarily complex;
- help developers and maintainers following coding conventions and standards.

2. Intentional Views

Before continuing, we explain the essence of the intentional view model. For more details, see [MMW02a, MPG03].

> *An* intentional source-code view *is a set of related (static[2]) program entities (such as classes, instance variables, methods, method statements) that is specified by one or more alternative intentional descriptions (one of which is the 'default' intentional description). Each intentional description is an executable specification of the contained elements in the view. Such a description reflects the commonalities of the contained elements in the view, and as such, codifies a certain intention that is common to all these elements. We require that all alternative descriptions of a given view are 'extensionally consistent', in other words, after computation they should yield the same set of elements.*

The above definition highlights some key elements that turn intentional views into more than mere 'sets' of program entities:

Intentional. The sets are not defined by enumeration but are computed from a specification. This is useful when the software is evolved as the sets are 'updated' automatically: it suffices to recompute the specification. Intentional descriptions are also more understandable and concise.

Declarative. Preferably, to make them easier to read and understand, the executable specifications should be written in a declarative language. This is important as they codify essential knowledge on the programmers' assumptions and intentions.

Alternative descriptions. Some descriptions are more intuitive; others are more efficient to compute. As such it is useful to specify both. Also, sometimes there are different natural ways in which to codify a view, depending on the perspective taken.

Extensional consistency. The consistency constraint between different alternative descriptions allows us to assess the correctness of the view definition, as well as the consistency of the actual source code (e.g., consistent usage of certain conventions and assumptions in the source code).

Deviations. Although not mentioned in the definition, for each alternative we can specify positive and negative 'deviations', i.e. elements that do not satisfy the specification of the alternative but that should be included, and elements that do satisfy the specification but should not be included. These deviations indicate 'exceptions to the general rule' made by programmers. They also help in defining intentional views incrementally: you can start out with a rough rule that has some exceptions and refine it later to make it more precise.

Relations. By relating intentional views we codify high-level structural knowledge about the source code.

Negative information. By using logic negation in our intentional descriptions we can codify negative information too (all program entities that do *not* have a certain desired property), which is often very powerful.

Intentional views can help an evolver because they allow him to ensure that the code — either before or after an evolution step — has a certain structure or satisfies certain conventions. "Negative" views of all program entities that do not have a certain desired structure or do not satisfy a certain convention, are also useful for evolution purposes, as they group all entities that need to be modified (so that they do have the desired structure or satisfy the desired convention).

To help a software engineer define intentional views, we implemented a prototype tool called the Intentional View Browser [MPG03]. This tool supports the declaration of intentional views on top of the VisualWorks Smalltalk development environment. The Intentional View Browser also verifies extensional consistency.

3. Logic metaprogramming

The computational medium in which we specify our intentional source-code views is SOUL [MMW02b], a Prolog-like logic programming language. But SOUL is more than a mere logic programming language: it is a

[2] Although the definition itself does not really require that the program entities contained in intentional source code views be static, in our particular implementation of the intentional view model, we can only reason about static program entities.

metaprogramming language, which enables logic reasoning about an underlying base language.[3] In our case, this base language is the object-oriented programming language Smalltalk. In fact, SOUL was implemented entirely in Smalltalk, which made it quite easy to make it reason about Smalltalk (thanks to Smalltalk's strong reflective capabilities).

As a concrete example of an intentional source-code view (and of the capacity of SOUL to reason about its own underlying Smalltalk implementation), consider the definition of a view soulLogicTestMethods. When implementing the SOUL logic libraries, we followed a kind of unit testing approach where every logic predicate was tested separately. The view soulLogicTestMethods groups all methods that implement tests for SOUL logic predicates.

For readability purposes, we edited the logic declarations of our intentional views somewhat so that they resemble Prolog syntax more closely (except that Soul logic variables start with a question mark); we do assume that the reader is somewhat familiar with Prolog syntax.

```
view(soulLogicTestMethods,[extractedFromClasses,withSamePrefix]).

viewComment(soulLogicTestMethods,'This intentional view contains all methods that implement tests for logic
predicates.').

default(soulLogicTestMethods,withSamePrefix).
```

The above facts declare an intentional view soulLogicTestMethods with two alternatives extractedFromClasses and withSamePrefix, of which the latter is considered the default alternative. The alternative extractedFromClasses codifies the intention that a method is a *logic test method* if it belongs to a class that implements tests for logic predicates (which is verified by an auxiliary predicate soulLogicTestClass). An exception is made for private methods, which are only auxiliary methods that are used by the actual *logic test methods*.

```
intention(soulLogicTestMethods,extractedFromClasses,?MethodDefinition) :-
        soulLogicTestClass(?Class),
        classImplements(?Class,?MethodName),
        not(privateMethod(?Class,?MethodName)),
        methodDefinition(?Class,?MethodName,?MethodDefinition).
```

Now suppose that an auxiliary method exists that was not put in the private protocol, where it belongs. In that case, we can still exclude it to keep the alternative intentional descriptions consistent. Of course, this is a temporary fix and we should require the developer in charge to fix the error as soon as possible. E.g.,

```
exclude(soulLogicTestMethods,extractedFromClasses,?MethodDefinition) :-
        methodDefinition(Soul.SoulTests.ErrorHandlingTest,inspectorClassUsedFor:,
                ?MethodDefinition).
```

The other alternative withSamePrefix codifies the intention that all methods that implement tests for logic predicates have the same prefix[4] 'test' and belong to a subclass of a class LogicTests:

```
intention(soulLogicTestMethods,withSamePrefix,?MethodDefinition) :-
        hierarchy(Soul.SoulTests.LogicTests,?Class),
        classImplements(?Class,?MethodName),
        startsWith(?MethodName,test),
        methodDefinition(?Class,?MethodName,?MethodDefinition).
```

4. Potential usage patterns

Rather than just enumerating the results of our experiments, we decided to present them in a pattern-like format, thus broadening their scope of usability. The pattern style allows us to abstract away from the particular logic metaprogramming approach we used to codify various design issues. Being purpose-oriented, this kind of presentation enables an engineer to quickly identify what results he could reuse to aid in solving a relevant

[3] Although intentional source code views could be specified in any metaprogramming language, we chose a *logic* language because we felt that declaring them in a logic metaprogramming language made them more"intentional" and declarative.

[4] Note that the fact that all these methods have the same prefix 'test' is more than a naming convention. It is required by the SUnit application which will automatically run these test methods as "unit tests".

maintenance or evolution problem he is faced with. Each of our potential *usage patterns* consists of a name, a purpose indicating what task the intentional view was used for, a rationale explaining why this task is a relevant one, a solution describing how exactly we can use intentional views to help in achieving that task and a concrete example of such a solution.

We prefer to talk about our usage patterns as "potential" patterns because, as most of our patterns have just been discovered, they are still immature in the sense that we have not yet been able to check their validity on other case studies. We have also not yet elaborated other important aspects of these patterns such as possible alternative solutions, when (not) to apply the pattern and relationships among patterns. Although not elaborated upon in this paper, we are also planning to define a *pattern language* describing how these usage patterns coexist and interact. Such a pattern language can help an engineer to identify the set of patterns that address his concerns and to find out how to combine them to aid him in his software maintenance and evolution tasks.

We are currently using intentional views to maintain and evolve two applications. At this stage, we have defined six potential usage patterns, that each solve a relevant maintenance problem. Due to space limitations, we only show two of them, illustrated by a single example each. For more examples and patterns, see [MPG03].

Usage pattern 1: Enforcing coding conventions

Purpose. Verify the consistent use of certain coding conventions throughout the system.

Rationale. Programmers (and Smalltalk programmers in particular) use lots of coding conventions and 'best practice patterns' to codify their intentions [Bec97]. Unfortunately, consistent usage of such conventions and patterns strongly depends on the programmers' discipline, as it is difficult to verify that the conventions are actually respected throughout the software system (and remain respected after having evolved the software).

Example. Suppose we want to enforce the convention that every mutator method assigns a value to the corresponding instance variable. We can do this by defining an intentional view mutatorMethods with two alternatives. The extensional consistency constraint between the two alternatives takes care of the rest.

The first alternative codifies the Smalltalk naming convention that mutator methods always have the name of the instance variable that they modify, followed by a colon[5].

```
intention(mutatorMethods,byName,?M) :-
  mutatorMethod(?M,?).

mutatorMethod(?M,?V) :-
  instVar(?C,?V),
  equals(?N,{?V:}),
  classImplementsMethodNamed(?C,?N,?M).
```

The second alternative refines the first one with an extra clause which states that the method ?M actually assigns some value to the variable ?V. The predicate methodWithAssignment will traverse the entire method parse tree of the mutator method to search for such an assignment.

```
intention(mutatorMethods,byBody,?M) :-
  mutatorMethod(?M,?V),
  methodWithAssignment(?M,?V,?).
```

Extensional consistency of these two alternatives implies that all methods that follow the naming convention of mutator methods will actually assign the appropriate variable as well.

Solution. The extensional consistency constraint between the different alternative descriptions of an intentional view can be used to implicitly express an essential convention or assumption in the source code.

Enforcing such a convention will make the software cleaner and easier to understand, and thus easier to evolve. We can also use the extensional consistency constraint for evolution purposes. When the constraint is not satisfied, this implies that some entities do satisfy one alternative intentional description but not another. For example, it may be the case that we have a method with a mutator name, but which does not assign any value, or

[5] The expression `{?V:}` produces a string which is the concatenation of the string representation of the value contained in the logic variable `?V`, with a colon.

vice versa. Once we have discovered these faulty entities, it is easy to modify them so that the extensional consistency constraint will be satisfied.

Usage pattern 2: Checking Design Consistency

Purpose. Verify consistency of the system's source code with a higher-level design diagram.

Rationale. Without a means of ensuring that the source code of a software system is, and remains, consistent with a higher-level design diagram, the design diagram soon becomes outdated and looses its relevance as high-level documentation of the source code.

Solution. To verify whether every class in, for example, a UML class diagram corresponds to one in the source code and vice versa, we declare one intentional view with two alternative definitions. The first alternative groups all classes that have been defined in the diagram[6], the other groups *all* existing classes in (the relevant part of) the implementation. Inconsistencies may arise either when adding a class to the source code without updating the diagram or when evolving the diagram without updating the code. These inconsistencies will be detected automatically when verifying extensional consistency of the intentional view. The same applies for methods, instance variables and class variables. Due to space limitations, we only show the view defined for classes. Note that the inImplementation alternative is based on the convention that all classes of the diagram are implemented in the same namespace, but can be adapted easily when another convention would be used.

```
view(classesOfDiagram,[inDiagram,inImplementation]).

intention(classesOfDiagram,inDiagram,?ClassName) :-
        umlClass(Diagram,?ClassName,?,?,?).

intention(classesOfDiagram,inImplementation,?ClassName) :-
        namespaceForDiagram(?Namespace,Diagram),
        classNameInNamespace(?,?ClassName,?Namespace).
```

Example. We recently built a small proof-of-concept application for computing the invoices of a mobile phone operator. The source code for that application was partially generated from a UML class diagram description. The above solution allowed us to verify easily when the design diagram was out of sync with the implementation and when either (part of) the code needed to be regenerated, or the diagram needed to be updated.

Again this pattern can be used for evolution purposes, to modify the source code and/or diagram so that they become consistent (in case the existential consistency check would fail).

5. Discussion

In spite of the high declarative and intuitive nature of intentional views, one might argue that it still requires an above-average engineer to define intentional views. Therefore, we are currently investigating how to facilitate or automate the task of defining intentional views. This is also a crucial issue in order for the technique of intentional source-code views to be scalable and applicable in the context of large industrial software.

One possibility is to offer a simpler, decidable, but maybe less expressive, language in which to define the intentional views (as opposed to using a full-fledged logic programming language). Another way is to add tool support that offers some predefined templates for the most common kinds of intentional views, which only need to be parameterized with some concrete details. A third solution is to offer a tool that helps us in semi-automatically extracting intentional views from the source code or from an enumerated set of elements (or "extensional view"). For example, some of our colleagues are investigating the use of inductive logic reasoning to derive the logic rules describing an intentional view from a set of examples contained in an extensional view [TBKG03]. Of course, the question then remains how to come up with these extensional views. We are currently conducting some experiments to use the technique of conceptual analysis to automatically extract interesting extensional views from source code.

6. Related Work

Existing software engineering tools and formalisms provide only small, restricted sets of decomposition and composition mechanisms, that typically support only a single, "dominant" means of separating the concerns of

[6] In our experiment, the UML class diagram was expressed as a set of logic facts and could thus easily be reasoned about by the same logic (meta)programming language that we used to reason about source code entities.

importance in a software system. Tarr et al. [TOHJ99] propose Hyperslices and Hypermodules as a new paradigm for modeling and implementing software artifacts in a way that keeps separate all concerns of importance. In their approach, a program is defined in an N-dimensional space where each dimension is a different concern. e.g. objects, functionalities, properties. Each hyperslice defines a decomposition (a modularization) of the program according to one of its dimensions, considering methods as primitive, indivisible units.

The work on intentional source-code views is clearly similar in spirit to Tarr et al's technique of multi-dimensional separation of concerns [Men02], as it also acknowledges that it is essential to be able to separate the different concerns of importance. However, the abstraction of intentional source-code views is not targeted at modeling and implementing software systems but rather at understanding, maintaining and evolving software systems. In addition, as opposed to Tarr et al. we do *not* propose a *new* paradigm, but propose our approach as a technique that aids software engineers in their maintenance and evolution tasks, *without* drastically changing the way in which they work.

The same remark can be made about the relationship between Intentional Programming [CE2000, Chapter 11] and the work on intentional source-code views.

7. Conclusion

Evolving and maintaining software requires adequate documentation of its implementation. However, due to the software's constant evolution, this documentation is often absent, incomplete, or not synchronized with the implementation. We proposed intentional source-code views as an "active" documentation technique to alleviate this problem: intentional source-code views are defined as a logic query on the program and are thus defined intentionally rather than enumerating the collection of source-code entities involved. In this way, intentional source-code views provide support for software evolution: when the source code changes the intentional source-code views remain.

Although creating such views is not a trivial task, the support they may offer to future software maintainers may very well be worth the initial investment. Because the different ways in which intentional views may aid software maintainers and evolvers are documented in the form of usage patterns, an engineer will be able to quickly identify what particular pattern is useful for the particular maintenance or evolution task at hand. Two examples of how intentional source-code views can be used to support software evolution were presented: *enforcing coding conventions* and *checking design consistency*.

8. References

[Leh84] M. M. Lehman. Program evolution. Information Processing & Management, 20(1-2):19-36, 1984.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1997.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[DDM03] Serge Demeyer, Stéphane Ducasse and Kim mens (editors). *WOOR'03 – Proceedings of the 4th International Workshop on Object-Oriented Reengineering 2003*. Published as Technical Report 2003-07 by the Department of Mathematics & Computer Science, Universiteit Antwerpen.

[Men02] Kim Mens. Multiple Cross-Cutting Architectural Views. Position paper submitted to the Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000).

[MMW02a] Kim Mens, Tom Mens, Michel Wermelinger. Maintaining software through intentional source-code views, In *Proceedings of SEKE 2002*, pp. 289–296. ACM, 2002.

[MMW02b] Kim Mens, Isabel Michiels, Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 23(4):405–431. Elsevier, November 2002.

[MPG03] Kim Mens, Bernard Poll, Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of ICSM 2003*. IEEE, 2003. *(To be published.)*

[TBKG03] Tom Tourwé, Johan Brichau, Andy Kellens and Kris Gijbels. Induced intentional software views. Paper submitted to ESUG'03.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of ICSE 1999*, pages 107–119, 1999.

Identifying Problems with Legacy Software Preliminary Findings of the ARRIBA Project ⋆

Isabel Michiels[1], Dirk Deridder[1], Herman Tromp[2], and Andy Zaidman[3]

[1] Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan, 2 , 1050 Brussel, Belgium
{Isabel.Michiels,Dirk.Deridder}@vub.ac.be
[2] Universiteit Gent, INTEC, Sint-Pietersnieuwstraat 41
9000 Gent, Belgium - Herman.Tromp@UGent.be
[3] Universiteit Antwerpen, Lab On Re-Engineering, Middelheimlaan 1
B-2020 Antwerpen, Belgium - Andy.Zaidman@ua.ac.be

Abstract. The goal of this experience report is to identify some of the key problems of today's enterprises that have to deal with managing their large business critical software systems. Our motivation to do so is based on preliminary findings from the ARRIBA project. The work we present here form our preliminary conclusions of the first 6 months of the project, where we visited some of these enterprises, to identify their main needs of today.

Keywords: Legacy Systems, EAI, restructuring, COBOL

1 Introduction

The dynamics of modern business applications is characterized by a constant need for integration and restructuring and this at a much larger scale than ever before. This is often driven by the physical integration and restructuring of companies, which consequently results in a need to alter their ICT infrastructures to accomodate the changed business activities. Possible examples are the redefinition of a corporate strategy, a corporate take-over, a conversion of the existing infrastructure from a data processing model towards a service oriented model, etc ...
This continuous modification process will finally result in a situation where several software systems have to collaborate in a way that was never (or could never have been) anticipated in their original design.
Such large-scale software applications are often referred to as *legacy applications*. In this report we will adhere to the following definition [1] of a legacy application [13]:

> *A legacy system is an operational system that has been designed, implemented and installed in a radically different environment than imposed by the current ICT strategy*

⋆ This research is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT)

[1] Other definitions are in use [2]

When burdened with the task to enable the collaboration of these separate systems, having access to a rich collection of documentation (preferably also feedback from the original designers/ programmers of the system) is imperative. Unfortunately, in all but a few cases, this documentation remains non-existing or has become completely out of date due to evolution of the software system. Generally speaking, one could state that the only true description of the information structures and the implemented behaviour is locked up in the running software system itself. Hence to obtain a sufficient (active) set of documentation one will have to turn to analysing the available static (e.g. source code, data models) and dynamic (e.g. runtime event traces) artifacts of the system.
There are five ways to handle a legacy situation in which a change is imposed :

1. Develop a new system from scratch
2. Refactor - rewrite portions of the system preserving its existing behavior
3. Porting the system to another platform
4. Migration strategy with partial reuse of the existing system

As we have seen in our first findings, a solution is chosen by doing a combination of the four above methods.
In this report we provide a preliminary overview of a number of encountered problems when confronted with legacy software. We have based ourselves on the results of visiting three major Belgian enterprises in the context of a research project called ARRIBA. In the following section, we will briefly describe the ARRIBA project. Then we will present our first findings in section 3 and in section 4 we will point out future work for the project. We will then round up in section 5 with our conclusion.

2 The ARRIBA project

The ARRIBA project is a generic research project funded by the IWT, Flanders [2]. The project started in October 2002 and will allow 6 researchers to work on the project for 4 years [3].
The aim of ARRIBA is to provide a methodology and its associated lightweight tools in order to support the integration of disparate business applications that have not necessarily been designed to coexist. Inspiration comes from real concerns that are the result of an investigative effort on the part of some of the research partners in this consortium. The object of this investigation is the identification of mainstream ICT problems within a representative forum of Belgian enterprises (large and small) that rely on information technology for their critical business activities. Part of what we propose to investigate is covered by the newly named discipline of Enterprise Application Integration (EAI); another part is covered by re(verse) engineering; however, our ambitions reach further.
At the roots of the ARRIBA project are two driving forces. On the one hand we have a consortium of research groups that have been active in the field of software engineering

[2] Institute for the Promotion of Innovation by Science and Technology in FlandersIWT - http://www.iwt.be
[3] ARRIBA: Architectural Resources for the Restructuring and Integration of Business Applications, see http://arriba.vub.ac.be

and more particularly in re(verse) engineering, software evolution and software architectures[4]. These groups have a fairly long-standing history of cooperation and they feel confident that they can join forces and tackle the new and ambitious problem domain targeted in the ARRIBA proposal.
On the other hand, we have the already mentioned and recently created forum of Belgian enterprises interested in a joint initiative to identify generic problems and likewise generic solutions plaguing their ICT base[5]. This forum has the form of a foundation hosted by what could best be described as a collective spin-off of the five Flemish computer science departments.
The academic partners together with the user committee (the first providing the content of the ARRIBA project, the second providing the context) will guarantee the correct identification of the problem setting and the proper channeling of the results to the business world.
The user committee of the ARRIBA project currently consists of 7 Belgian enterprises. They form the steering group of the project: they regularly check if we tackle current ICT problems and during the evolution of the project they will see whether our results will be industrially applicable. The next section reports on the first findings based on visits of part of the user committee members.

3 First Findings

During the first six months of ARRIBA, we visited 3 major Belgian enterprises: the KBC group[6], Banksys [7] and LCM [13] [8]. As preparation for these visits, we prepared a question list according to [5]. One of these visits was organized in a workshop format, while the other two were more Q&A sessions based on presentations by the companies In a later phase, other visits to other companies are planned.
In what follows, we have organized our findings into common themes:

The Mainframe Syndrome

All of these organisations depend heavily for their back-office on proven technology and duplicate datacenters, which are essential for their critical business activities; this is an environment strictly used for controlling processes to be able to ensure operational

[4] There are 3 Flemish academic partners involved: Vrije Universiteit Brussel (VUB), Universiteit Gent (UG) and the Universiteit Antwerpen (UA) and two other European partners, UCL in Louvan-La-Neuve, Belgium and SCG, Berne, Switzerland. The latter two play a supporting role

[5] These Belgian enterprises are grouped in a *User Committee* currently consisting of 7 companies: Inno.com, KBC, LCM, Banksys, Toyota, KAVA and Pefa

[6] KBC is a large banking company that holds three major product factories: banking, insurance and marketing activities

[7] Banksys is one of the most important providers of the infrastructure for electronic financial transactions in Belgium

[8] Landsbond der Christelijke Mutualiteiten (LCM) is a large organisation responsible for the redistribution of health care allowances, and offers also a number of related social services

performance.
In the front-office environment and end-user environment, UNIX-like systems and J2EE application server systems are also used. They are not always considered to be fully reliable, and therefore less suited to support their essential business operations. This situation indicates that there is a serious resilience towards new and not yet proven technology (hardware as well as software). The integration with the existing mainframe environment also remains a very big issue [9, 10]. Previous efforts to migrate to a Microsoft technology-based system have proven to be unsuccessful, at least in the case of LCM [13].

Organisation and Human Resources

Most organisations have a pretty strict and project-based organization which is clearly reflected by the Human Resources setup. This adds considerably to their latency and inability to adapt. Take as an example LCM: they have about 200 COBOL developers, and (only) 4 or 5 Java-aware software engineers. It is clear that in such an environment there will be a lot of resistance towards new developments (the systems are functioning properly, aren't they, so why change anything?). The previous point is reflected in the structure of the organisation : separated business units, project driven work structure, etc. [4] . A central authority to control major revitalisation efforts, to enforce architectural consistency and provide a deployment policy is often missing or very difficult to install.

Coding Standards and Techniques

In general, it is estimated that between 60 and 80 % of today's operational code is still written in COBOL [3, 1]. Some C or Java code is also present, but only in small quantities. Knowledge about the systems is partly lost and only evident in the code itself, e.g. people have left the company, documentation is very poor (and out of sync with the current system) or not present. When validating the quality of these mainframe COBOL applications, usually the 80/20 rule will manifest itself: 80% of the coding problems are caused by 20% of the code (also known as *The Pareto Principle*) [8].
Regarding architectural issues, migration has been put forward as the main bottleneck of the restructuring process; however we found that companies experience that integration with new technologies is much more important (and also more difficult). Take as an example the introduction of new environments (like J2EE) for new applications: the real challenge here is how to let these connect or communicate with the other COBOL applications on the mainframe.

Data and Information

Large-scale software systems consequently also have to deal with large amounts of data. Unfortunately, in most legacy systems, the use of a Relational Database Management System (RDBMS) is scarce. Proprietary flat file systems are still in use, but migration to using an RDBMS system has received top priority.

In large organizations, a Corporate Data Model is hard to enforce. The reason for this is simple: there is no central ownership of data or information items in use by these companies. This often leads to a rapid growth of different information models, where every part of the organization has its own view on that same information, with differences in structure and even in the semantics of these information models. Take as an example the concept of a *customer*: it is interpreted differently in other business units within the company; a *customer* that buys something is very different from a *customer* that complains about the companies' delivered products. So the quality of the data models and the data itself, because of the lack of a responsible person, is far from guaranteed. Also, a consistent view on the information between the business units themselves [14] is missing. As a consequence, migrating the software and the information models becomes a real problem.
At the KBC for example, the use of a uniform data model cannot be enforced, but instead they enforce a uniform message model. This means that they clearly specify the syntax and semantics of messages that are sent between different software applications, not the form of the data itself. In practice, this approach has proven to give very satisfactory results.

Using Standard Packages

One of the possible solutions these companies bring forward to better structure their applications is using standard ERP packages. However, this causes several problems [12]. Experience proves that packages have a strong front-end (or presentation layer), but a weak back-end for performance. On the other hand, some applications require the use of certain packages since they implement international standards.
Security forms a large problem as well; it can be a conclusive reason for refusing the use of a package. Another drawback of using packages is that they are expensive and sometimes do not have the functionality that is really needed. Customizing these packages can be risky due to package updates; therefore a decision is made every time whether a package should be bought or written from scratch.
Another issue is that the view of the package on the business domain does not map directly to the real world, and the amount of work to be done for integrating these packages into the existing application is highly underestimated. Formulated otherwise: there is a semantic gap between the ”standard” package and its existing information model; and performing gap analysis is time-consuming.
Another open question that still remains is how to map the companies' business process model onto the ICT infrastructure of the predefined package.

Datawarehousing

Setting up datawarehousing activities is not a trivial thing to do: project-driven businesses (like the KBC) need to set-up a project first, mainly about collecting meta-data information. Since this data is cross-cutting different business units, these projects are difficult to 'sell', because it is difficult to find a single business case for them. After all, possible profit can only be shown after a while. Most extraction of meta-data is done by interviewing people: they are the most valuable sources of information. And although

most companies see the importance of datawarehousing, it is not really clear yet what they will do with all the meta-data information.

Enterprise Application Integration (EAI)

This rather new domain dates from the mid-nineties [7, 11]. According to Linthicum Enterprise Application Integration is [6]:

> *The unrestricted sharing of information between two or more enterprise applications. A set of technologies that allow the movement and exchange of information between different applications and business processes within and between organisations.*

At this moment there is a growing number of enterprises that try to use this, usually under the form of standard EAI tools (at the KBC they use eGate, Tibco and some EAI tools developed within the company). For KBC, they have been using this through a business case since 1998. Problems that arise now for KBC mainly come from handling different EAI tools at once: now it is almost impossible to go back to using only one tool throughout all units within the company. Instead, the use of the tools is being extended, according to the needs and applications, inside the growing domain of EAI.

The IT Development Process

The IT Development process is usually well-defined within a company policy and a lot of attention is paid to it. However, as mentioned before, it is not technology-driven, which has as a downside that projects that do not have a business case (that are hard to sell within the company), cannot be realized.
Developing and collecting documentation is, in some cases, part of the predefined software development process of the company. Unfortunately it is too often neglected for obvious reasons (e.g. time consuming, limited budget). So there is documentation available, but it is in most cases not up-to-date with the current software. So the source code and the information models are often the only reliable source of documentation.

4 Future Work

Based on our first findings, we conclude that the first step for restructuring legacy applications is to understand and analyze the source code (we will call this *Code Mining*). This can be accomplished by analyzing static as well as dynamic information and taking into account the data and information models as well.
The second step could then be to identify lightweight tools that can, using the results of the analysis of the first step, automatically extract architectural information, documentation or domain knowledge out of the source code and data models. A last step could then be to incorporate changes into the extracted artifacts and propagate these back into the code (forward engineering).
Future tracks will emphasize more on COBOL and its environment and on how to use dynamic information as well:

Emphasis on COBOL code and its environment Since the companies we presented before are willing to let us experiment on their code, we will concentrate in a first phase on studying COBOL code and its environment. We would like to apply some of the already known tools (that were developed in the labs of one of the academic partners), like SOUL [9] or CodeCrawler [10]. Since these tools were not developed specifically for COBOL, we first have to see how we can adapt them to use them within this context.
We have started to work on transforming COBOL into a more portable platform: we intend to use XML as a portable format for source code representation. We can then manipulate XML documents inside other language platforms. In a second phase (forward engineering), we could try to manipulate this XML representation (either directly on the DOM model, either through XSLT) and retransform it back to COBOL to actually restructure the code.
In the near future, we would also like to investigate in which way we can reuse techniques developed for object-oriented systems, like code metrics, code refactoring,... for restructuring (and enhancing code quality) non-OO legacy systems.

Using Dynamic Information in the context of reverse engineering, static analysis is the term used for a reengineering effort based solely on the information that can be found in the source code of the software. In many cases this analysis is computationally very intensive and doesn't give the whole picture. Dynamic analysis uses information collected during the execution of the program. The information we collect is called an event trace and consists of a list of method invocations, procedure calls, object instantiations, etc. A clear advantage of using dynamic analysis is that the information you have is always correct with respect to the execution of the program, but a clear disadvantage is the amount of information you have to wade through. Research in this direction will revolve around finding event sequences that logically belong together in the execution of the program, i.e. a clustering operation. These clusters can then be abstracted to patterns that point to key functionality in the software.

5 Conclusion

In this experience report, we have identified some of our preliminary findings of the ARRIBA project, which aims at providing lightweight methodologies and tools for the integration of software entities that have not necessarily been designed to cooperate.
During the first phase of the project we visited 3 out of 7 enterprises that are part of the project's user committee, and we presented some surprising commonalities found in their current ICT restructuring schemes.
Finally, we ended by pointing out our future work for this ARRIBA project, with as next intermediate goal to experiment with some mainframe applications (witten in COBOL) and applying some already known lightweight tools to see what we can achieve.
In the near future, we will continue with the company visits.

[9] Smalltalk Open Unification Language - http://prog.vub.ac.be/research/DMP/soul/soul2.html
[10] see http://www.iam.unibe.ch/ lanza/CodeCrawler/ codecrawler.html

References

1. Aberdeen Group. Legacy applications: From cost management to transformation, 2003. Executive White Paper from Aberdeen Group, March 2003. Can be found at http://www.aberdeen.com/2001/research/03038126.asp.
2. M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems - Gateways, Interfaces and the Incremental Approach*. Morgan Kaufmann Publishers, 1995.
3. G. D. Brown. Cobol: The failure that wasn't. COBOLReport.com - http://www.csis.ul.ie/COBOL/course/.
4. J. O. Coplien. *Pattern Languages of Program Design*, volume 1, chapter 14 - A Development Process Generative Pattern Language. Addison-Wesley, May 1995.
5. S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.
6. D. S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.
7. J. C. Lutz. EAI architecture patterns. In *EAI Journal*, March 2000.
8. V. Pareto. The pareto principle or the 80:20 rule. http://www.public.asu.edu/ dmuthua/pareto's_principle.html.
9. D. Plakosh, S. Comella-Dorda, G. A. Lewis, P. R. H. Place, and R. C. Seacord. Maintaining transactional context: A model problem. Technical report, SEI, august 2001. CMU/SEI-2001-TR-012 - ESC-TR-2001-012.
10. M. Stonebraker and J. M. Hellerstein. Content integration for e-business. In *SIGMOD Conference*, 2001.
11. M. Themistocleous and Z. Irani. Evaluating and adopting application integration: The case of a multinational petroleum company. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.
12. M. Themistocleous, Z. Irani, R. M. O'Keefe, and R. Paul. Erp problems and application integration issues: An empirical survey. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.
13. H. Tromp and G. Hoffman. Evolution of legacy systems, strategic and technological issues, based on a case. Paper also accepted to the workshop on Evolution of Large-Scale Industrial Software Applications (ELISA), 23 September 2003, ICSM 2003.
14. K. Vandenborre, P. Heinckiens, G. Hoffman, and H. Tromp. Coherent enterprise information modelling in practice. In *Proceedings of 13th European-Japanese Conference on Information Modeling and Knowledge Bases, Kitakyushu, Japan, June*, 2003.

A Case for Establishing Evolutionary Policies and their Support Mechanisms, with Examples †

Nazim H. Madhavji
University of Western Ontario, Canada
madhavji@csd.uwo.ca

Josée Tassé
University of New Brunswick, Canada
jtasse@unbsj.ca

Abstract

An important trait of a mature discipline is that, amongst other things, practitioners have specific criteria to judge the appropriateness of the different courses of action to take under a given circumstance, or whether a given task has been well-accomplished. These criteria may be in the form of templates, checklists, rules-of-thumb, constraints, policies and laws, which have resulted from many years of experience with repeated application of these in different situations. There is data to support that software evolution practices are far from mature. Thus, in this position statement, we make a case for establishing a (i) comprehensive set of evolutionary policies and (ii) their support mechanisms, to guide development[1] in the context of the instituted policies. A benefit of utilising established policies and their support mechanisms is that the sustainability of the evolving systems would likely be increased.

1. Introduction

While the overall process maturity in software organisation continues to improve according to the SEI's Year 2002 Year End update [1], there are still a staggering 60% of the 1,345 organisations assessed worldwide (appraised *and* reported since 1998) that have been calibrated at Level 1 or 2 on the software Capability Maturity Model (CMM) [2] and another approx. 25% at Level 3. The first two levels denote *chaotic* and *repeatable* practices, respectively, while Level 3 denotes *defined* processes in an organisation. Both technically and numerically, majority of the organisations are far from the 15% organisations that are at the, desired, higher levels of maturity (Level 4 -- *managed* and Level 5 -- *optimising*). Overall, therefore, the worldwide picture of software development can be considered quite gloomy.

Moreover, because most significant software projects in industry are in evolutionary stages (i.e., beyond the first release of a software system), we can assume that the software projects assessed were typically *not* new development projects. Also, whilst in general there are many factors that contribute to the overall low process maturity rating in software projects, there is no reason to believe that *software-evolution-related* factors (e.g., ability to control size growth, or amount of regression testing conducted in proportion to the degree of code change, etc.) were not amongst them. Software *evolution* community, both research and practice, thus has every reason to be concerned about the state of the art and of practice in software evolution.

Also, the Standish Group's CHAOS study [3] of 23,000 applications in US companies over five years (1994-1998) shows that, while more and more projects are succeeding, by 1998 approx. 28% were failing outright and another 46% were significantly challenged on the quality and delivery fronts. This is corroborated by data from another source [4] that also indicates that approx. 30% of the large projects get cancelled, and that the probability of a system of size 1 million lines of code (MLOC) or some 10,000 Function Points (FP) getting cancelled is approximately 50%. Not only this, large systems are notorious for: drastically overshooting schedules and budgets; severe reduction in requirements, features or functions after project start; not delivering what was promised; major reliability and performance problems following delivery; and many other issues [5].

Add to this abysmal record the approximately 8% annual growth (new + changed), though in migration projects (hardware or software based), volatile environments, or in early evolutionary life, the growth can be significantly higher (25-100%) [4]. This, therefore, raises a challenge as to how to increase the life expectancy of, say, a 10,000 function point system from the current average of 10-15 years.

2. Position

There are many lines of attack in attempting to solve software evolution problems. In this position statement, however, we make a case for establishing a (i)

† This work is supported, in part, by research grants from NSERC (Natural Science and Engineering Research Council of Canada).

[1] In this paper, by "development" we mean *evolutionary* development unless indicated otherwise.

comprehensive set of *evolutionary policies*[2] and (ii) their *support mechanisms*, to guide development[3] in the context of the instituted policies.

A rationale for this strategy is that, in a mature discipline, amongst other things, practitioners have specific *criteria* to judge the appropriateness of the different courses of action to take under a given circumstance, or whether a given task has been well-accomplished. These criteria may be in the form of templates, checklists, rules-of-thumb, constraints, policies and laws, etc., which have resulted from many years of experience with repeated application of these in different situations.

In the building industry, for example, single-glazed or ¼" double-glazed windows would be considered inadequate for the deep wintry conditions of Quebec (typically in the range -20 to -30 °C); whereas, they would be considered acceptable-to-comfortable for the mild winters of New Zealand. Such knowledge is often a result of past mistakes. For example, when early British settlers emigrated to New Zealand, the orientation of many houses did not maximise solar access in the principal rooms which, in the years to come, precipitated house remodelling.

In the field of software *evolution*, however, while progress has no doubt been made over the last thirty-odd years, exemplified by Lehman's laws [6], the general principle of "design for change" [7], or by numerous other empirical studies (some of which are cited later in the section on discussion) it is our contention that, as a community, our rate of progress in adopting and developing evolutionary policies and their support mechanisms has been undeniably slow[4]. For example, the Trial Version 1.00 of The Guide to the Software Engineering Body of Knowledge (SWEBOK) [8] – specifically Chapter 6 (Software Maintenance) and Chapter 10 (Software Engineering Tools and Methods) -- neither mentions policies for evolving software nor their technological or methodological support as a critical issue.

Thus, in the absence of a *concerted*[5] effort by the software evolution community, developers have often resorted to use, manually of course, *ad hoc* policies and rules of thumb, such as:

> *If the number of files edited for a given change is ≤ six then self-reviews would suffice; otherwise, independent inspection would be conducted.* [9].

While an argument in favour of such practice is "better this than none", it does little to further the discipline as a whole. Consequently, even in a single *large* project, let alone *across* projects, divisions or organisations, one may find inequity in the application of specific policies. The net result is an imbalance in software quality in different parts of even the same system; integration delays due to hold ups, or feature or test reduction to cope with integration and release schedules; higher evolutionary costs; and ultimately, user dissatisfaction.

Ad hoc and esoteric practices in a given project almost certainly imply a lack of a comprehensive (or practically viable) set of policies concerning different aspects of software evolution. Much remains to be done, therefore, in defining *detailed* policies to guide, monitor and verify project-specific actions in all areas of software evolution (e.g., from release planning, detailed analysis, to release implementation, and involving numerous types of software artefacts).

From the preceding description, it should be evident that the total number of policies required to comprehensively satisfy the needs of a software evolution project would be *quite large*. There is thus a danger that such a large set of policies could become the heart of a bureaucratic machine reeked with policy management problems, which would defeat the purpose of institutionalising policies in the first place.

To avoid this danger, but also, in fact, to apply policies effectively, there is a need for technological support to design, codify, organise and evolve policies and verify development against these policies. In our work thus far, we have concentrated mainly on the last of these. Detection of development-violation against the policies would help fix product or process problems at their earliest; whereas, any "positive" feedback would help build stronger confidence in the development team. Collectively, thus, a significant benefit of utilising established policies and their support mechanisms is that the sustainability of the evolving systems' quality would likely be increased.

---

[2] An evolutionary policy is defined as a statement of rule, guiding principle, strategy, plan, course of action, procedure, or constraint, to follow during the process of software evolution.

[3] In fact, we also need mechanisms to ensure *continued* relevance, comprehensiveness and soundness of the enacted policies. But we choose not to delve into policy management and evolution issues in this position statement so as not to lose focus on development issues, which are clearly of first order importance.

[4] While one may argue that this slow pace is due to the lack of a general theory of software evolution, we contend that there are nuggets hidden in numerous empirical studies and in general practice awaiting discovery and their synthesis into formalized policies that can be supported by automated means. A prime purpose of this position statement is to demonstrate a humble beginning in this direction.

[5] While it is not our intention to give here a particular blueprint for such a concerted effort, examples exist in other disciplines where such effort has resulted in benchmarks, body of knowledge, standards, Open Source software, etc., which have proved invaluable for experimentation, learning, and business.

3. Examples

In this section, we give two brief examples of policies derived from third-party empirical work [10, 13]. These examples deal with pertinent issues in software evolution, such as: re-engineering change-prone modules, and consistency between code and documentation. Some more examples can be found in a companion paper [12].

3.1 Example 1: Re-engineering change-prone modules

Mattsson and Bosch [10] have proposed an approach to identify those modules of a system that require re-engineering. Proactively maintaining the software (an object-oriented framework in their case) by restructuring the change-prone modules could "simplify the incorporation of future requirements″. In their approach, the change-prone modules from past releases are identified based on their size, change rate and growth rate.

Once it has been decided which modules need to be re-engineered during the development of a particular release, an important issue then is to ensure that all the identified modules do in fact go through the re-engineering process. However straightforward this may appear by itself, such monitoring -- basically carried out manually today – is leaden with the risk of losing track of the tasks involved amidst project pressures.

In an automated system, however, a policy such as the following could be defined:

Policy:

$\forall$ c $\in$ {p $\in$ TypedEntSet("Component") | p.name $\in$ <*list of components*> } •
$\exists$ r(a,m,t) $\in$TypedRelSet("activity consumes component") • a.name = "re-engineering"

where, "<list of components>" denotes the list of modules to be re-engineered. The policy says that for each component in the given list, there should be an activity called "re-engineering" that consumes (or operates on) the component.

This policy would be used to verify the development *plan*, such as that shown in Figure 1. This plan shows two versions of the same system, called V-elicit[6], (see the two double circles) and the new-release development process (see the hierarchy of boxes representing the process activities). Version 5 (V5) of the system consists of the components (see ellipses linked to V5): `visualization_V2`, `policies_V1`, `generator_V2`, `base_code_V5` and `view_matching_V1`. This system is to be updated to version 6 (V6), whose planned components (also shown by ellipses) are likewise linked by its arrows. The new-release development process (*model*) consists of the activities: `make_changes`, `re-engineering` and `testing`. For simplicity, no further activity breakdown is shown here.

Let us now assume that the Mattsson-Bosch approach identified two components for re-engineering: `view_matching_V1` and `generator_V2` (from version 5). The *"<list of components>"* in the policy description above would thus be replaced by these two

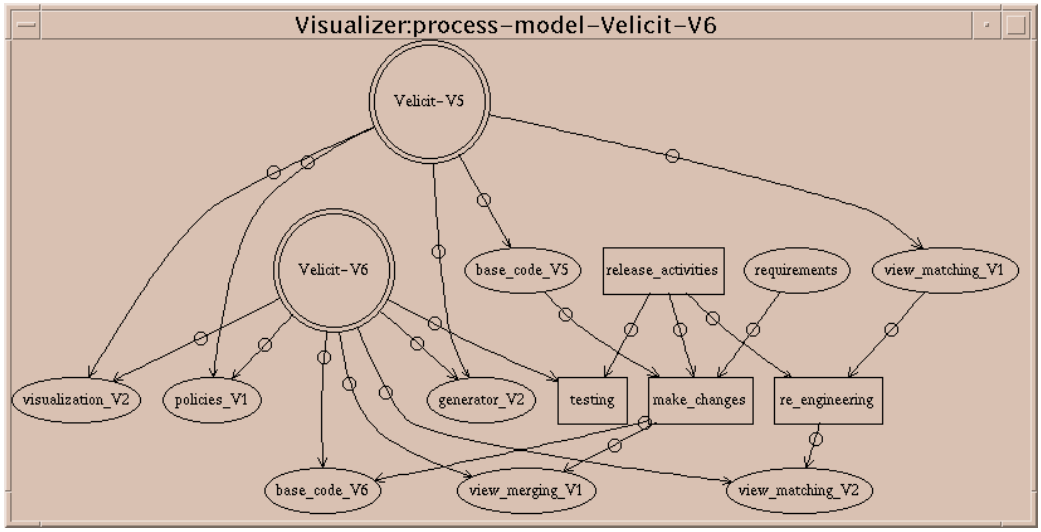


Figure 1 - Overall plan for the development of the sixth version of the V-elicit system.

[6] V-elicit is a system for eliciting models of processes or products [11]. Its operational details are not relevant to this paper.

component names[7].

The policy checking mechanism[8], described in [12], accepts two inputs (a policy and a model) and produces feedback as to whether or not the model complies with the policy and, if not, identifies the offending elements and relationships of the model.

Evidently, the plan in Figure 1 is not correct. Specifically, the component `generator_V2` (identified for re-engineering) is mistakenly left out from the re-engineering effort (i.e., this component is not an input to the "re-engineering" activity box in Figure 1). Such mistakes do occur when building prescriptive models in the planning phase, even in moderate sized projects. This is why it is quite important to verify the planned process - against the prescribed policies -- prior to its execution, in order to prevent evolution errors.

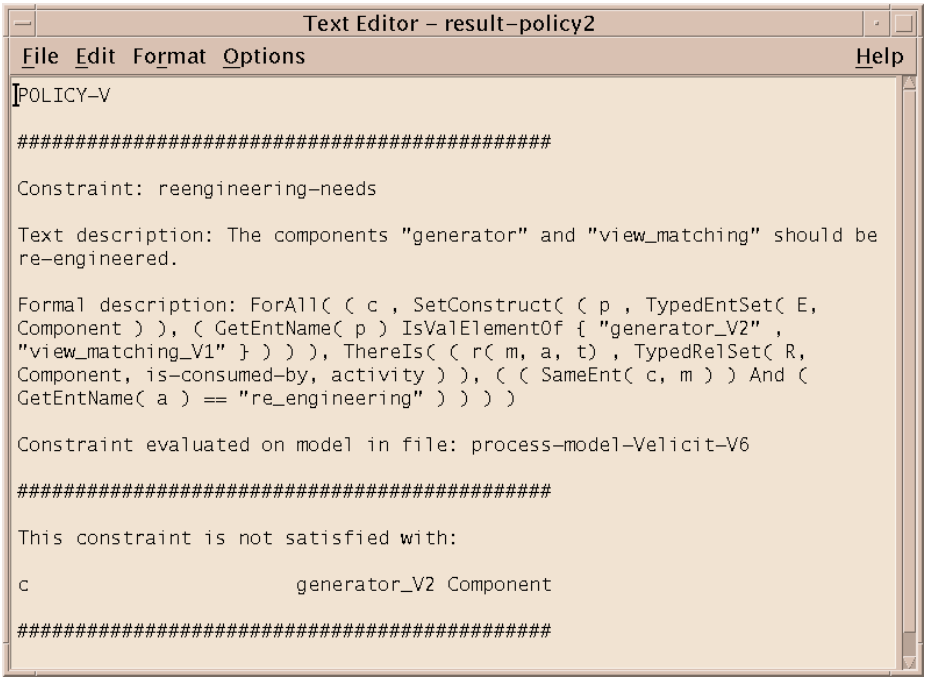


Figure 2 - Verifying the plan for re-engineering change-prone modules.

Figure 2 shows the result of verifying the plan against the described policy. The top part of the figure describes the policy informally and then formally. The bottom part lists the violations -- that is, those components that were supposed to be re-engineered but have not been included in the plan.

[7]An advanced form of this policy could automatically detect the components that should be re-engineered, for example, components with a change rate higher than a certain threshold.

[8] This mechanism is relevant here in concept, not so much in its details.

Such *automated* checking is much preferable to hand-checking the plans, and its value is particularly felt: in large or complex systems; when many individuals are involved in the project; when quality is at stake; and when time is at a premium. Also, the policy checking mechanism can be used in either *prescriptive* or *descriptive* contexts. For example, in the former context, as described above, it is used to ensure that the plan is complete prior to process enactment. In the latter context, it can be used to monitor a project's progress by verifying the process-trace against the policy at a desired time in the project.

3.2 Example 2: Code-documentation consistency

A case study by Tryggeseth [13] shows that the availability of valid documentation during software evolution increases system understandability and productivity. However, maintainers and evolvers often document their work by means of memos, which are not always integrated into the master documentation [7]. Over time, therefore, the documentation gets increasingly out of date to the point that the documentation is no longer trusted or used. Often, this triggers costly and intensive reverse-engineering of the system to recover the "lost" design, architectural, requirements or other software artefacts.

A preventative approach would ensure that with any new development or changes the documentation and implementation are congruent with each other. For example, in an object-oriented system one may want to verify that the code implements exactly the class diagram in the design document (i.e., no classes missing or added, and all attributes and method interfaces properly implemented). This can be achieved by comparing the class diagram from the design document against that generated by reverse-engineering the new or updated code, guided by a policy that specifies those entities and relationships that should be similar in the two diagrams. For example, the following policy verifies whether all the classes in the code are included in the design document.

Policy:

∀ c1 ∈ {c ∈ TypedEntSet("Class") | c.source = "code" } •
∃ c2 ∈{d ∈ TypedEntSet("Class") | d.source = "documentation" }•
c1.name = c2.name

Figure 3 (bottom part) then shows that the *code* class `line-on-invoice` does not match the class diagram from the *documentation*. Likewise, policies can be written to verify in more detail whether related classes have the same attributes and functions (including parameters and return values).

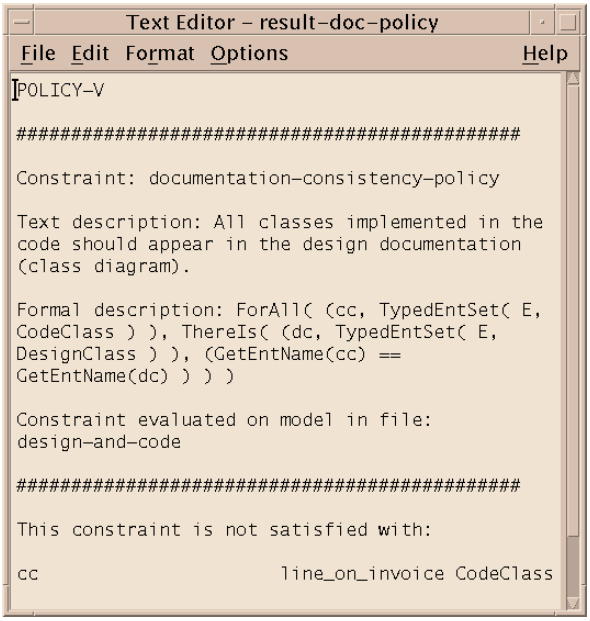


Figure 3 - Verifying consistency between documentation and code.

3.3 Discussion and Closing Remarks

The described two examples are illustrative in the basic idea of evolutionary policies and their supporting mechanism, though, needless to state, much further work is necessary to make this an industrial-scale reality. As a step in this direction, we have derived, codifed in logic and, in some cases, pseudo-codified for preliminary assessment, a number of other policies interpreted from empirical studies or experiential works of others, e.g.: 35 policies from Davis' 201 principles of software development [14]; 42 policies from Lehman's laws of evolution [6]; Munson's proportional regression testing policy [15]; Humphrey's optimal value of "*Appraisal-to-failure ratio*" [16]; and Ramanujan et al.'s "*standard for variable naming*" [17]. What this does suggest is that evolutionary policies are numerous, if implicitly buried in their rudimentary forms in the literature or in specific practices.

Time is thus ripe to dig into such literature, all the while conducting more empirical studies to establish evolutionary facts; use the findings to design suitable evolutionary policies; experiment with such policies to assess their validity in case studies and in practice; and investigate into policy-design and support mechanisms (see [12], for example, where we describe a mechanism to verify evolutionary software artefacts and processes against instituted policies). The overall objective of this work is to improve software evolution practice and to improve the quality-sustaining power of software systems. This is no mean task, however, and therefore to make significant progress, it would require a concerted, community, action as opposed to isolated efforts of a few individual researchers and practitioners.

ACKNOWLEDGEMENTS

We are thankful to the three anonymous referees, whose comments have helped us to improve this position paper.

4. References

[1] The SEI, "Software CMM CBA IPI and SPA Appraisal Results 2002 Year End Update", April 2003, available from: www.sei.cmu.edu/sema/profile.html (accessed: May 2003).

[2] M.C. Paulk, C.V. Weber and B. Curtis, "The Capability Maturity Model: Guidelines for Improving the Software Process", Addison Wesley Professional, 1995.

[3] "CHAOS: A recipe for success", The Standish Group International, Inc., 1999.

[4] C. Jones, "Applied Software Measurement – Assuring Productivity and Quality", 2nd edition, McGraw Hill, 1996.

[5] W. Wayt Gibbs, "Software's Chronic Crisis", Scientific American, September 1994, pp72-81.

[6] M. M. Lehman and J. F. Ramil, "Rules and Tools for Software Evolution Planning and Management", Annals of Soft. Eng., Vol. 11, 2001, pp. 15-44.

[7] D.L. Parnas, "Software Aging", Proc. Of the 16th International Conference on Software Engineering, Sorento Italy, May 1994, IEEE Press, pp. 279-287.

[8] SWEBOK -- Guide to the Software Engineering Body of Knowledge, Trial Version 1.00, May 2001, IEEE Computer Society.

[9] H. Dayani-Fard, Personal communication, 2002.

[10] M. Mattsson and J. Bosch, "Observations on the Evolution of an Industrial OO Framework", Proc. of the International Conference on Software Maintenance 1999, pp. 139-145.

[11] J. Tassé and N. H. Madhavji, "View-Based Process Elicitation: a User's Perspective", Software Process Improvement and Practice, vol.6 no.3, Sept. 2001, pp. 125-139.

[12] N.H. Madhavji and J. Tassé, "Policy-guided Software Evolution", Proc. of the International Conference on Software Maintenance, September 2003, Amsterdam (To appear).

[13] E. Triggeseth, "Report from an Experiment: Impact of Documentation on Maintenance", Empirical Software Engineering, vol.2 no.2, Kluwer Academic Press, 1997, pp.201-207.

[14] A. Davis, "201 Principles of Software Development", McGraw Hill, 1995.

[15] J. C. Munson, "Measuring Software Evolution", chapter submitted for consideration in "Software Evolution" (eds.) Madhavji, N.H., Lehman, M.M., Ramil, J.F. and Perry, D., Wiley (pending).

[16] W. S. Humphrey, "Using a Defined and Measured Personal Software Process", IEEE Software, vol.13 no.3, May 1996, pp. 77-88.

[17] S. Ramanujan, R. W. Scamell, J. R. Shah, "An Experimental Investigation of the Impact of Individual, Program, and Organizational Characteristics on Software Maintenance Effort", Journal of Systems and Software, vol.54 no.2, October 2000, pp. 137-157.